UNIVERSITY OF COPENHAGEN
DEPARTMENT OF COMPUTER SCIENCE

Master's Thesis

Thorbjørn Bülow Bringgaard

Efficient Big Integer Arithmetic Using GPGPU

With focus on implementations of exact addition, division, and multiplication in CUDA C++ and Futhark

Supervisor: Cosmin Eugen Oancea

May 31, 2024

Abstract

Exact big integer arithmetic is a fundamental component of numerous scientific fields, and therefore, required to be efficient. One way to increase efficiency is by acceleration on GPGPU, calling for parallel arithmetic algorithms. This thesis examines parallel algorithms for addition, multiplication, and division, with the premise of fitting in a CUDA block, and consequently, suited for medium-sized big integers. The algorithms are implemented in the high-level languages C++ and Futhark. The addition algorithm boils down to a prefix sum, which runs efficiently in both implementations. The multiplication algorithm is the classical quadratic method, parallelized by orchestrating the convolutions in a way that balances the sequential work per thread and minimizes synchronization. The C++ implementation exhibits good performance, while the Futhark implementation leaves room for improvement. The division algorithm is based on finding multiplicative inverses without leaving the domain of big integers. To do so, a variety of big integer operators and routines are defined, including shifts, comparisons, and signed subtraction using the prefix sum approach of addition. The algorithm parameterizes over the methods involved for big integer arithmetic, and its efficiency directly mirrors the given multiplication method. In addition to conveying the algorithm, as well as adapting it to big integers, supplementary implementations have been produced. This includes a validating and inefficient sequential implementation in C, and a partially validating and semi-efficient parallel implementation in Futhark.

Contents

1 Introduction

Integers are commonly represented as either 32- or 64-bit in both hardware and software. They are an essential part of computing and we expect most programs to utilize integer arithmetic in accomplishing a diverse range of tasks. However, some applications require numbers that are too big to fit a 64-bit integer. One solution is resorting to floating-point arithmetic, but that is both imprecise and inefficient compared to integer arithmetic.

Exact arithmetic with big integers (also called multiple precision integers or big numbers) is the very foundation of numerous fields in computer science, other formal sciences, natural sciences, and industry. Evident examples includes cryptography and algebra. Big integers can span hundreds, thousands, or even millions of bits, necessitating the exact arithmetic to be efficient in the size of the integers. A widespread implementation for such arithmetic is the GNU Multiple Precision Arithmetic Library (GMP) written in C and assembly [11]. One approach to further accelerate the performance is utilizing massively parallel hardware such as General Purpose Graphics Processing Units (GPGPUs).

In order to efficiently use GPGPUs, the underlying algorithms have to be adapted and parallelized. Addition has shown to be very efficiently computable by a scan operator [5, 7]. Multiplication classically runs in quadratic time [14]. The classical approach adapted to GPGPU is found to be efficient for small- and medium-sized big integers [8, 19]. Fast Fourier Transform (FFT) based multiplication algorithms are known to be asymptotically faster [14]. Due to the overhead of FFT, such approaches are most efficient on GPGPU for large-sized big integers in comparison to the classical approach [6, 8, 19].

Division is the hardest of the basic arithmetics. It traditionally involves a long division algorithm that iteratively finds one correct digit [14]. With the number of iterations linear in the input size, this algorithm is a poor fit for GPGPU. Another common division approach is by multiplicative inverses. Watt has shown an algorithm to efficiently compute exact division by finding such an inverse, without leaving the original domain [21]. Its complexity mirrors that of its multiplication method, over which can be parameterized, and the number of iterations is logarithmic in the input size, yielding a more suitable algorithm for GPGPU.

This thesis focuses on efficient parallel implementations of exact big integer arithmetic for GPGPU. It presents the algorithms for an efficient addition, classical multiplication, and Watt's exact division by whole shifted inverse. Algorithmic, parallelization, and optimization efforts are kept general, but the implementations are narrowed to the Compute Unified Device Architecture (CUDA) platform through the programming languages C++ and Futhark. Both are high-level languages, but operates vastly different. C++ allows low-level command over primitives and fine-grained memory control, while interfacing directly with the CUDA runtime API to produce GPGPU executable code [1, 20]. Futhark is a functional array programming language that is designed around parallel basic blocks, making programs more elegant and less dependant on hardware specifications, in exchange for loosing some of the fine-grained and low-level control [9, 12].

The arithmetics are implemented at CUDA block-level, and hence, aimed at medium-sized big integers, ranging roughly from a few hundred to a few hundred thousand bits. Each algorithm includes optimizations to further enhance the performance at block-level – or performance in general. The results show that both the produced addition and multiplication methods are competitive performance-wise, but the performance gap between C++ and Futhark implementations grows with the complexity of the algorithms and applied optimizations.

The produced implementations of division are not as highly optimized or efficient as for the other arithmetics. However, this thesis is (to our knowledge) the first to recognise and use Watt's division algorithm (outside of Watt's own work), and in turn, first to parallelize it.

The contributions of this thesis are:

- A description of efficient parallel big integer addition and classical multiplication algorithms, on top of gradual degrees of optimizations over the shape of the inputs, accompanied by implementations at CUDA block-level in C++ and Futhark.
- A benchmark driven performance evaluation of the produced addition and classical multiplication implementations against a state of the art CUDA library.
- A presentation of the high-level intuition and specialization for big integers of Watt's algorithm for exact division by whole shifted inverse, including a revision that extends the algorithm to an otherwise unconsidered cornercase.
- An inefficient sequential prototype of Watt's algorithm in a low-level language (C).
- A parallelization effort of Watt's algorithm that culminates in a partially valid and semi-efficient Futhark implementation, entailing efficient parallel operators to shift, compare, and subtract big integers.

The structure of this thesis is as follows: Section 2 presents other work related to the subject of this thesis. Section 3 details the practicalities of the developed software suite. Section 4 provides the background information on GPGPU, CUDA C++, and Futhark, assumed throughout this thesis. Section 5 regards the representation of big integers on a machine. Section 6 outlines the overarching strategy of the implementations. Sections 7, 8, and 9 presents the algorithms, optimizations and implementations of addition, multiplication, and division, respectively. Section 10 presents the methodology and results of validation testing. Section 11 benchmarks and evaluates the performance of the addition and multiplication implementations, while giving a detailed description of the methodology and performance metrics. Section 12 concludes the work of this thesis and lists directions for future work.

2 Related Work

Two works are particularly related to this. The first is "GPU Implementations for Midsize Integer Addition and Multiplication" by Oancea and Watt [19]. It has been the developed concurrently with this thesis and served as an initial inspiration (with some of their CUDA C++ setup files as a starting point for our code basis[1]), but the two has been developed independently. It shares a similar approach to addition and classical multiplication as this thesis. However, where this thesis focus on division, their work focus on FTT multiplication. They found that FFT multiplication becomes faster than classical multiplication for big integers of size greater than 2^{15} bits. Their approach to FFT involves finding a finite prime-field that allows the Discrete Fourier Transform (DFT) to stay in the domain of integers, while simultaneously use bases that almost maps to a machine word size one-to-one.

The other particularly related work is the state of the art CUDA library called "Cooperative Groups Big Numbers" (CGBN), published by NVlabs [18]. CGBN is aimed at integers in the range of 2^5 bits to 2^{15} bits (i.e. small- to medium-sized integers), where each integer is processed in a cooperative group of either 4, 8, 16, or 32 threads. Cooperative groups are collections of threads assigned with special intra-communication properties [1]. As evident by the group sizes, CGBN optimizes their arithmetics for warp-level processing. In comparison to the approach of this thesis (i.e. block-level processing of arithmetics), their approach minimizes latency by storing intermediate results in local memory – rather than shared memory – and advocates the usage of fast warp-level instructions. In turn, this allows consecutive arithmetics to fuse seamlessly, and to run very fast for smaller-sized integers. However, their approach impose constraints such as being hardware-dependent (i.e. the fast warp-level instructions are specific to the proprietary CUDA platform), offer no scalability to integers above 2^{15} bits (e.g. large-sized integers will exhaust local memory and registers), and requiring the size of integers to be evenly divisible by 32.

Other related work includes a classical and FFT multiplication by Dieguez et al. [8], where the classical multiplication takes a divide-and-conquer approach s.t. convolutions are tiled over CUDA blocks. This has the benefit of increasing the amount of parallelism within a block, but at the cost of blocks having to integrate partial convolution results and carries using atomic operations.

To propagate the carries, Dieguez et al. use the hierarchical carry look-ahead scheme that Emmart and Weems use in their big integer addition [10]. This addition scheme is structured around propagating at block-level in a bottom-top-bottom fashion: First the digits are added. Then carries are propagated in threads, then warps, then block, afterwards to be distributed back in warps, followed by threads. The blocks can then overflow to the following chunk of digits, which would then repeat the process. In comparison to their work, we recognize the carry propagation as a scan.

[1] `https://github.com/coancea/midint-arithmetic`

A more distant related work is Cuda Multiple Precision Arithmetic Library (CAMPARY) by Joldes et al. [13]. It aims at small-sized integers up to a few hundred bits of precision, and use unevaluated sums of floating-points numbers to represent the integers internally. Hence, it relies on floating-point arithmetic, rather than exact arithmetic. The idea behind the number representation is to compute the exact error of a floating-point, and then store the rounded floating-point and the exact error in two different floating-points. This decomposes their big integer arithmetic operators to a series of hardware-supported floating-point arithmetic operations, while checking the errors. They also support division, based on a similar Newton-Raphson approach as the division algorithm we use [21]. The algorithms regarding errors are computationally demanding and the limiting factor for integer sizes.

3 Software Structure

Various big integer arithmetic implementations have been developed as part of this thesis, publicly available at the GitHub repository `https://github.com/tossenxD/big-int`.

The repository is structured as follows: The directories `cuda`, `futhark`, and `prototypes` contains CUDA C++, Futhark, and C code, respectively. Each directory includes a `README` with detailed explanations of the setup and a `Makefile` to replicate the results of this thesis.

CUDA C++ The files `ker-add.cu.h` and `ker-mul.cu.h` contains the kernels for addition and multiplication. The arithmetics are gradually optimized over three and five versions, respectively, kept in the files for the sake of comparison. The kernels are called from the main file `main.cu`, which serves to generate inputs, check for errors, test the arithmetics, and run benchmarks. The heading of the main file defines numerous parameters specifying integer base, kernel version, what to run, etc., which can manually be adjusted – along with a detailed explanation of each parameter and kernel version.

The `Makefile` offers two ways to run the main file – with either a normal or small amount of total work. They are invoked by `$ make` and `$ make small`, respectively. The main file prints benchmark results in the form of metrics defined in section 11.2 (, and possibly prints the results of validation tests too, if validation is enabled).

Files to run CGBN is in subdirectory `cgbn`, and have a similar setup to the primary files.

Futhark The files `add.fut`, `sub.fut`, `mul.fut`, and `div.fut` contains Futhark programs with big integer addition, subtraction, multiplication, and division. The addition and multiplication has four and three versions, respectively, similarly to the CUDA kernels. The big integer base type can be configured in the file `helper.fut`. While the implementations compile for all base types, the division program is nonsensical for bases other than `u16` due to the base restrictions explained in section 9.2.

The subdirectories `test` and `bench` contains the Futhark test and benchmark programs. Benchmarks are straightforward, and can be run by `$ make add-uX` or `$ make mul-uX`, where `X` is the bits of the specified base type (64 is default and 16 is not supported). The tests includes a C file that is tricky to call – using the `Makefile` is recommended. They can be run with `$ make test-uX`, where `X` is the bits of the base type (64 is default). The benchmarks prints runtimes and input size, and the tests prints the count of valid testcases.

Sequential C Prototype The file `div.c` contains a sequential prototype for big integer division, that may be of interest to run experiments on. It can be called either with random inputs (`$ make random`) or a fixed input (`$ make fixed`), which tests against GMP. Both the number of random inputs and the given fixed input can be adjusted in the `Makefile`. The default number of random inputs to test with is 1000, and the `README` contains information on how they are generated. The random tests will print the inputs, output, and correct output of failed tests, and report the number of valid tests.

4 Preliminaries

Common processors in a computer includes a Central Processing Unit (CPU) and a Graphics Processing Unit (GPU). The CPU typically executes general purpose routines, while the GPU is specialized in massive parallel processing of the numerous vectors that constitutes computer graphics. However, the GPU is not confined to processing graphics and can act as a General Purpose GPU (GPGPU) to execute general parallel routines.

Compared to the CPU, GPGPU has the benefits of higher instruction throughput and higher bandwidth (achieved by various design differences such as increased core count), but has the main drawback of requiring substantially more effort to properly utilize. For our purposes, GPGPU provides a significant performance speedup and scalability in exchange for parallel algorithms and programs of greater complexity.

This section is structured as follows: In 4.1 we introduce the GPGPU architecture and parallel execution model. In 4.2 we give an overview of the CUDA C++ environment, which allows interfacing with low-level GPU primitives in the high-level language C++. In 4.3 we give an overview of the high-level programming language Futhark that can compile to GPU executable code without the need to manually manage low-level primitives.

4.1 GPGPU Architecture

There are two main interfaces for a GPU: OpenCL and CUDA, the former being an API for an open set of standards and instructions, and the latter a proprietary platform for NVIDIA GPUs consisting of specialized instructions. This thesis focus on CUDA, yet the algorithmic aspects are kept general and allows for OpenCL operability, along with the software developed in Futhark compiling to both models.

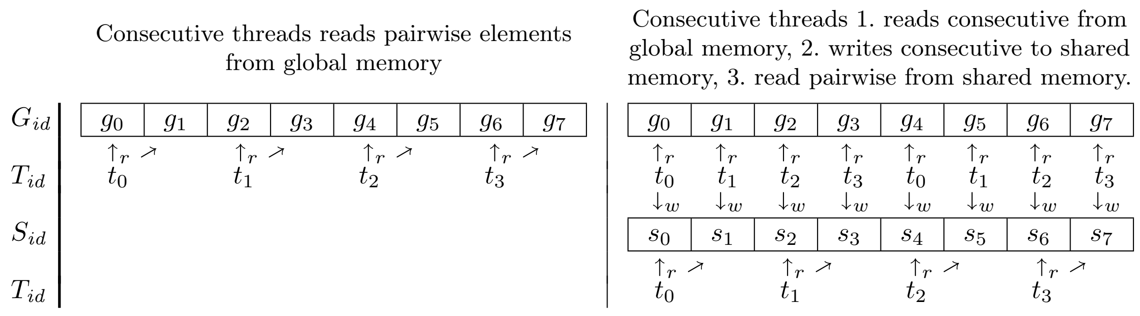

Consecutive threads reads pairwise elements from global memory
Consecutive threads 1. reads consecutive from global memory, 2. writes consecutive to shared memory, 3. read pairwise from shared memory.
G_{id}
T_{id}
S_{id}
T_{id}
g_0 g_1 g_2 g_3 g_4 g_5 g_6 g_7
t_0 t_1 t_2 t_3
s_0 s_1 s_2 s_3 s_4 s_5 s_6 s_7
$\uparrow_r$ $\downarrow_w$

Figure 1: Example of coalesced global memory access using a shared memory buffer – G denotes global memory cells, S shared memory cells, T the threads, and r-w indicates read-write transactions. Left-hand-side shows uncoalesced global memory access, while right-hand-side shows coalesced access in 3 steps.

GPGPU is parallel by design and consists of up to thousands of multithreaded cores. The architecture, according to NVIDIA's own CUDA C++ programming guide [1], is as follows: Threads are grouped in 32, where groups are called *warps*. A warp executes one instruction at a time in lockstep, called Single Instruction, Multiple Threads (SIMT). A program, called a *kernel*, is executed on a *grid* of *blocks* (also called a thread block) with a specified amount of threads per block. The number of threads in a block (block size) and the number of blocks in a grid (block count) must be statically known prior to kernel execution, with a maximum block size of 1024 threads. Each warp, block, and grid has its own memory called *local*, *shared*, and *global* memory, respectively, with the former being the smallest and fastest, and the latter the largest and slowest.

SIMT allows writing parallel programs of minimal overhead w.r.t. synchronization and memory latency. It urges to keep procedures closest to warp-level, utilizing faster memory, lockstep execution, and optimizing for locality of reference within a warp. This introduce the idea of *coalesced memory accesses*, where consecutive threads access consecutive memory locations, resulting in as few memory transactions as possible, and necessary for maintaining high bandwidth while accessing global memory. However, consecutive threads may not be supposed to access consecutive elements. In such case, it is faster to use shared memory as a buffer s.t. transactions between shared and global memory are coalesced, and the original uncoalesced transactions are between shared and local memory. Figure 1 illustrates this approach with an example. While we may not gain much speed in the shown example, it becomes a significant latency reduction with many more threads and elements per threads.

An important side-effect of SIMT is no divergent control flow (branching) within warps. I.e. suppose only 1 thread in a warp choose an `if`-branch; then, the remaining 31 threads cannot proceed execution before the first thread has finished, and instead, executes the `if`-branch instructions on dummy data.

Thus, in order to construct efficient programs for GPGPU that exhibit high throughput and bandwidth, they must be inherently parallel, contain minimal amount of branching, use fast memory, run as close to warp-level as possible, and access global memory coalesced.

4.2 The CUDA C++ Programming Interface

In this thesis, we interface with CUDA by means of a C++ language extension that compiles a program to both a CPU and GPU executable portion using C and the CUDA driver API, as described in the CUDA C++ guide [1]. The CPU running the initial program is called the *host* and the target GPU is called the *device*. The extension exports functions in the *alloc*-family, that allows the host to preallocate, read, and write memory of the device. Device functions (i.e. kernels) are called from the host, and take parameters in the form of pointers to memory pre-transferred to the device.

The extension exports some function execution- and variable memory- space specifiers. These infer to the compiler how to treat functions and memory (e.g. host, kernel, or device function, shared or local memory, etc.). The extension also exports some device functions, notably the function `__syncthreads()`. It creates a barrier preventing threads from further code execution until all threads within the block has reached that barrier.

In order to write parametric kernels in e.g. the type or size of big integers, we can use C++ templates. They provide generic programming over type and value parameters that are concretized at compile time, as written by Stroustrup in [20]. Templates especially prove to be a strong tool when combined with other high-level C++ constructs, such as classes. (On a side note, the CUDA design has no explicit error handling on the device. Instead, CUDA API calls returns error codes, which the host must then explicitly check and handle.)

Lastly, threads are implicit: Kernels launch with an *execution configuration* of the form "$\texttt{ker} <<< \texttt{grid}, \texttt{block} >>> (\texttt{params})$", where `grid` and `block` specifies the grid and block dimensions, respectively, and are of type `dim3`, denoting the specifications in 3D. The spawned kernel has access to a set of special objects, notably `threadIdx`, `blockIdx`, and `blockDim` containing the index of the currently executing thread, the index of the block that it belongs to, and block size, respectively. Thus, the executable kernel code is run on each of the specified threads, and spatial awareness within a thread is achieved explicitly through the special objects – as demonstrated in Example 1.

From now on, we denote a CUDA C++ program simply as a CUDA program and assume:

- Kernel dimensions to be 1D (as they are in Example 1 above).
- The existence of sensible host functions to execute and error-check kernels.
- A small library of common subroutines such as accessing global memory coalesced.

These assumptions allow us to focus on the implementation of algorithms.

Example 1 (a simple CUDA C++ program)**.** The following program is a simple example of the general routines involved with parallel programming through the CUDA C++ extension.

```
• Kernel function that increments m integers of the input array by one.
template<int m> __global__ void incrKernel(int* input, int* output) {
  // find the global thread ID of the currently executing thread
  const unsigned int global_id = blockDim.x*blockIdx.x + threadIdx.x;
  // if its ID does not exceed the input size, it executes the operation
  if (global_id < m)
    output[global_id] = input[global_id] + 1;
}
• Host function to execute the above kernel assuming array h_mem contains m integers.
...
  // allocate and transfer memory to device
  int *d_mem_in, *d_mem_out;
  cudaMalloc((void**) &d_mem, m * sizeof(int));
  cudaMemcpy(d_mem_in, h_mem, m * sizeof(int), cudaMemcpyHostToDevice);
  // execute the kernel below with ⌈m/512⌉ blocks of 512 threads each
  dim3 block (512, 1, 1);
  dim3 grid (1 + ((m - 1)/512), 1, 1);
  incrKernel<m><<< block, grid >>>(d_mem_in, d_mem_out);
  // fetch results
  cudaMemcpy(h_mem, d_mem_out, m * sizeof(int), cudaMemcpyDeviceToHost);
...
```

4.3 The Futhark Programming Language

Futhark is a high-level pure functional language with emphasis on data parallel array programming [9, 12]. It is hardware-agnostic and can compile to both sequential or parallel C, OpenCL, and CUDA code [3]. The fundamental design of Futhark revolves around *Second-Order Array Combinators* (SOACs). They are array-functions that are easy to reason about and define in terms sequential semantics, while still being able to compile to (efficient) parallel code.

Asymptotic analysis of parallel programs is separated into *work* and *depth*. Work refers to the total amount of computations in the program – known as the traditional basis of time complexity analysis. However, in the parallel domain, work is distributed amongst threads, and so it may not be an accurate representation of runtimes. The depth (also called span) is the amount of sequential work within a thread, given infinitely many threads.

We now present the most fundamental SOACs, which we use throughout this thesis to analyse parallelism.[2] We assume their function inputs are $O(F_w(n))$ and $O(F_d(n))$.

First we have the combinator `map` of work $O(n \cdot F_w(n))$ and depth $O(F_d(n))$. It distributes

[2]Other combinators exists, but these are the most crucial to this thesis. An overview of the array combinators and functions can be found in the Futhark standard library [2].

a function over an array (known from other functional languages such as Haskell [15]) and can inherently be executed in parallel. The type signature and semantics is:

$$\texttt{map} : (\tau \to \tau') \to [\tau] \to [\tau'] \tag{1}$$

$$\texttt{map}\ f\ [a_0,\ a_1, \dots,\ a_{n-1}] \coloneqq [f\ a_0,\ f\ a_1, \dots,\ f\ a_{n-1}] \tag{2}$$

Next we have the combinator `reduce` of work $O(n \cdot F_w(n))$ and depth $O(\log n \cdot F_w(n))$. The semantics is akin to a `fold` from other functional languages, but with one big difference: The operator must be associative and have a left-associative neutral element, allowing the array to be accumulated in $O(\log n)$ steps. The type signature and semantics is:

$$\texttt{reduce} : (\tau \to \tau \to \tau) \to \tau \to [\tau] \to \tau \tag{3}$$

$$\texttt{reduce} \oslash e\ [a_0,\ a_1, \dots,\ a_{n-1}] \coloneqq e \oslash a_0 \oslash a_1 \oslash \dots \oslash a_{n-1} \tag{4}$$

Similarly, we have the combinator `scan` with work $O(n \cdot F_w(n))$ and depth $O(\log n \cdot F_w(n))$. The semantics corresponds to an accumulated reduction over the input array (also called a prefix sum), and so the associativity and neutral element restrictions of the operator applies as well. The type signature and semantics is:

$$\texttt{scan} : (\tau \to \tau \to \tau) \to \tau \to [\tau] \to [\tau] \tag{5}$$

$$\texttt{scan} \oslash e\ [a_0,\ a_1, \dots,\ a_{n-1}] \coloneqq [e \oslash a_0,\ e \oslash a_0 \oslash a_1, \dots,\ e \oslash a_0 \oslash a_1 \oslash \dots \oslash a_{n-1}] \tag{6}$$

Lastly we have the combinator `scatter` with work $O(n)$ and depth $O(1)$. It takes a destination, index, and value array, and performs an in-place distribution of the values over the destination array according to the indices. The indices and values must have same shape. Values are ignored for indices that are out-of-bound. The type signature is:

$$\texttt{scatter} : [\tau] \to [\texttt{i64}] \to [\tau] \to [\tau] \tag{7}$$

One of the strengths of the Futhark compiler is its ability to *fuse* chains of `map-reduce` compositions. This allow us to write nice and clean code, straightforwardly model parallel algorithms, and let the compiler generate more complex and optimized GPGPU code that use less intermediate values and instructions.

While Futhark is more clean and succinct than CUDA, it contains compromises that allows Futhark to compile to GPGPU code, making it restricted in comparison to other high-level functional languages such as Haskell. E.g. Futhark allows nested arrays as data structure, but all data must be flat on a GPGPU. Hence, the nested arrays must be regular in order for the compiler to know how to flatten them for execution. The shape of arrays must be statically inferred to the compiler too – primarily part of the type signature of functions, but can be manually inferred. Example 2 shows the difference in CUDA and Futhark clearly.

Overall, Futhark is meant to balance usability, power of abstractions, and parallel efficiency.

Example 2 (a simply Futhark program)**.** The following program is a Futhark translation of the CUDA program of Example 1 that increments the integers in a given array by one:

```
def incrFut [m] (input: [m]i32) : [m]i32 = map (+ 1) input
```

5 Representation of Big Integers

Big integers, also known as big numbers or multiple-precision integers, are integers that exceeds the word size on a machine. A common system for reasoning about them, e.g. used by Knuth in [14], is the positional number system:

Definition 1 (positional number system)**.** An integer $u \in \mathbb{N}$ can be expressed in base $B \in \mathbb{N} > 1$ with m digits $u_{i\in\{0,1,\ldots,m-1\}} \in \{0, 1, \ldots, B-1\}$ by the sum:

$$u = \sum_{i=0}^{m-1} u_i \cdot B^i \tag{8}$$

By choosing the size of a machine word as the base B (also called the radix), the positional system maps directly to an array data structure of unsigned words. I.e. for an unsigned word type `uint` of `b` bits, we get that a big integer u of m digits (also called limbs) in base $2^{\texttt{b}}$ is represented by an array of type `uint` and size m in little endian s.t. $u[0] = u_0$, $u[1] = u_1$, etc. Furthermore, in this representation, u can be viewed as binary with $\texttt{b} \cdot m$ bits; suppose a word is 64-bits and the size of the array is 4 – then u is a 256-bit integer where, say, the 67th bit of u is the third least significant bit of the second digit $u[1]$.

We have to make an important choice between arbitrary precision and exact precision integer representation. Arbitrary precision, like the name suggests, means that the precision is not bounded by software, and so we never loose precision from arithmetic operations (as long as we have enough memory to house the result). This is e.g. implemented in Haskell and in GMP [15, 11]. It provides a great deal of abstraction, as there can occur no overflows. In other words, it allows the programmer to work without worrying about the sufficiency of the underlying data structure.

However, the cost of this abstraction is the requirement for dynamic memory allocation to handle overflows. This may be a slight inefficiency on a CPU architecture and be well worth the level of abstraction, but dynamic memory management is problematic on a GPU because of the memory hierarchy discussed in section 4.1. Hence, arbitrary precision is an unsuitable representation for parallel computing.

Exact precision integers are bounded by a specified size, making them more suitable for a GPU architecture. The size can still be arbitrarily specified and changed during runtime (by means of allocating and copying to a new big integer), but the exact size remains known.

Thus, the chosen representation for big integers is exact precision arrays. In turn, our arithmetic preservers the dimensions between input and output, and require that the input dimension matches. This setup allows us to fully utilize GPGPU capabilities.

So far, we have only considered unsigned integers. There exists multiple representations of signed integers on a binary machine, the most common being *two's complement.* This representation does not work well with multiple precision, as extracting a negative number requires a bitwise negation. Instead, we can use *sign-magnitude* representation, where a signed integer is represented by an absolute value (magnitude) along with an indicator (sign). The downside of this representation is that 0 and -0 are two different numbers. This representation is also used in other high-efficiency libraries, such as GMP [11].

We use unsigned integers as the internal representation, as this is more efficient. In cases where signs are required, we extent by sign-magnitude. Furthermore, we only consider unsigned arithmetic (with wrap-around on overflow), as signed arithmetic is trivial to define using their unsigned counterpart.

Lastly, by fixing the representation of big integers to a data structure of arrays of unsigned machine words, we lay a foundation that allows us to efficiently compute arithmetics on GPGPU (and a CPU for that matter). However, also fixing the type of machine word (i.e. the base) does not increase performance or efficiency – instead, the arithmetics becomes more strict, hardware dependent, and nonextensible.

As a mechanism to keep the arithmetics generic over the base type in implementation, we utilize templates, classes, and type declarations to create an abstract big integer interface.[3] The interface is as follows:

Name	Description	Type	Example
`uint_t`	The base	unsigned integer	`uint32_t`
`ubig_t`	Double the base	unsigned integer	`uint64_t`
`qint_t`	Quadruple the base	unsigned integer	`unsigned __int128`
`carry_t`	Type of carries	unsigned integer	`uint32_t`
`bits`	Number of bits in base	integer	32
`HIGHEST`	Max number in base	unsigned integer	4294967295

Note that `ubig_t` and `qint_t` may not exist. Hence, implementations that depend on those types are only correct for some choices of base type. Overall, we assume the base is either 32- or 64-bit words, as these are the two most common word sizes, and, unless stated otherwise, the implementations type checks for both bases.

[3]Futhark has an extensive module system that supports generic programming in the same manner as C++ templates. However, it requires a lot of effort to wrap the whole implementation inside a module with generic base type. As a simple means to generality we use type declarations – allowing us to focus on the performance aspects rather than the intricacies of functional programming.

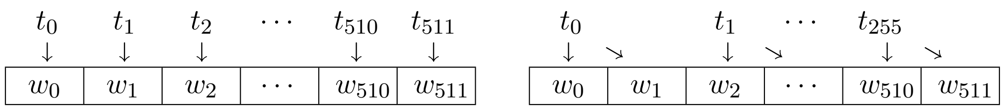


Figure 2: Example of sequentialization within parallel threads to reduce CUDA block size.

6 Overview of Implementation Strategy

For the arithmetics to be efficient, they must be implemented at a CUDA block-level, thus, minimizing the communication overhead by utilizing the faster block-level shared memory for intermediate results. In CUDA programs, this is a manual process achieved by the execution configuration and the methods deployed to index over kernel parameters. However, for Futhark, the compiler ultimately determines how to map it. Hence, we use the compiler attribute `#[only_intra]` when batch processing arithmetics in Futhark, telling the compiler to only map the arithmetics at block-level (intra-block).

In principle, any arithmetic function only requires global memory access to read the inputs and write the output. In Futhark, this can be a bit involved, discussed further in section 7.3. In CUDA however, we always assume the following kernel structure: Fetch inputs from global memory coalesced; execute the function body; write results to global memory coalesced. From now on, we refer to the function body when discussing kernels.

Given we optimize for block-level, the arithmetic operators are confined to run on at most 1024 threads. Hence, the implementation is aimed at medium-sized big integers, as when the input length creates block-sizes exceeding 1024, the operators are not able to run. This is an artificial barrier, as we can introduce sequentialization within each thread to allow bigger integer sizes. E.g. consider the example given in Figure 2. In the left illustration, each of the 512 digits of the big integer is handled by a separate thread, giving a block size of 512. In the right illustration, each thread handles two digits sequentially, giving a block size of 256 instead.

However, when the sequentialization factor within parallel threads increases, the amount of parallelism within the program decreases. Furthermore, sequentially handling many digits per thread may exhaust the local memory. Thus, the bigger the integers are, the less efficient the implementation becomes (for sizes that otherwise would exceed CUDA block limits). This only holds up to a certain input size, because the block-level shared memory also grows with the integer size, resulting in an out-of-memory error for integers too big.

Likewise, we can also have big integer sizes that are too small to efficiently compute in parallel. Suppose that each thread process a digit of the input, and the input consists of 16 digits – then we have 16 threads processing and 16 threads idling. This is not a problem when we only compute arithmetics on a single big integer, but effectively halves throughput when batch processing enough integers to consume the whole device.

Again, this is an artificial barrier, as we can combine (*flatten*) big integers and handle them *segmented* within a block. E.g. combining two integers per block in the example above is enough to fulfill a warp. This is not necessarily double the performance, since flattening, unflattening, and segmented operations comes with a cost, but the cost is overshadowed by the benefits (no idling threads) when processing many instances over smaller integers.

To summarize, the arithmetics are implemented at CUDA block-level aimed at medium-sized integers. They must increase the sequentialization factor within threads to allow bigger input integers (up to a limit). The throughput may improve from integrating segmented operations, processing multiple instances of integers per CUDA block.

We take inspiration from the strategy behind the GMP library: They write multiple parameterized versions of each arithmetic function (including multiple algorithms), and dynamically (or by tuning) choose the one performing best based on the input size [11]. While we do not implement multiple algorithms, we implement multiple versions of the same algorithm with a gradual degree of optimization – e.g. with and without the ability to process multiple instances per block.

We expect the most optimized version to consistently perform best, but run experiments and benchmarks over the size of integers on all versions. It reveals the performance gain of each optimization and possibly find versions most suited for specific inputs. Since performance is the main concern, we assume that the size and number of integers per block exactly divides the block dimensions for kernels exhibiting sequentialization and segmentation.[4]

7 Addition

Addition is the simplest of the basic arithmetic operators. It is a cornerstone that can be used to define all other operations - an essential part of big integer arithmetic.

Former work has been carried out for a parallel addition operator in Futhark, which will serve as a stepping stone for the algorithm and implementation of our addition [5]. In essence, big integer addition formalizes to a `map-scan` composition, where the `scan` handles carry propagation, making it run efficiently on a GPGPU [7].

This section is structured as follows: In 7.1 we present a parallel algorithm to compute big integer addition. In 7.2 and 7.3 we discuss how to efficiently implement the algorithm in CUDA and Futhark, respectively. Lastly, in 7.4 we show how to define subtraction from the addition algorithm.

[4]Assumption does not restrict the usefulness of the implementations since inputs can be padded to fit.

Input: u and v of size m base B
Output: w of size m in base B

```
c = 0
for i in (0..m-1)
    w[i] = u[i] + v[i] + c
    c = if overflow then 1 else 0
```

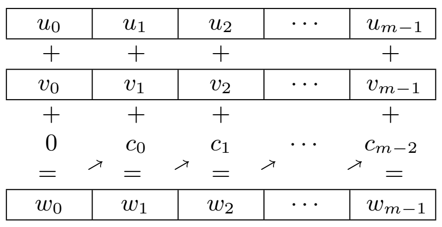


Figure 3: Pseudo-code and illustration of sequential addition algorithm.

7.1 Algorithm

From the Definition 1 we derive the following addition definition.

Definition 2 (big integer addition)**.** The addition of two big integers u and v of size m and base B is the sum of their added digits:

$$u + v = \sum_{i=0}^{m-1} (u_i + v_i) B^i \tag{9}$$

I.e. we compute each digit of the result simply by adding the corresponding two input digits. However, these inner additions may overflow the base, resulting in a carry being added to the following digit. In turn, this digit may now overflow, and so we need to add yet another carry, and so forth. E.g. consider the addition $199 + 1$ of decimal base; first we add 1 to 9 giving 0 and a carry, then we add 0 and the carry to 9 giving 0 and another carry, which we then add to 1, resulting in the number 200: $\begin{array}{cccc} & \overset{1}{0} & \overset{1}{0} & 1 \\ + & 1 & 9 & 9 \\ \hline & 2 & 0 & 0 \end{array}$

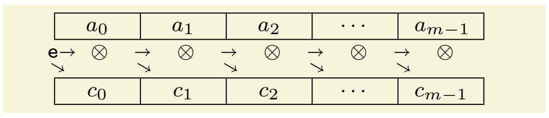
A sequential algorithm, and illustration thereof, is given in Figure 3. It is immediate from the figure that the sum of digits can be independently computed, and thus, trivially parallelizable. However, the carries are truly dependent on all sums and carries before it, as illustrated in the figure, but we know they can be computed using a `scan` [7].

Each addition may overflow once at most, and so we need to figure out the conditions for an overflow. In the former work by Olesen, Topalovic and Restelli-Nielsen, they found that this happens when I) the addition already overflowed, or II) it results in the maximum representable integer, and the addition just before it overflowed [5]. They found that these conditions can efficiently be checked in parallel as a prefix sum.[5]

Thus, the big integer addition parallelizes to a `map-scan` function composition. Figure 4 lists and illustrates a generalized parallel addition algorithm. The algorithm has three steps:

[5]Specifically an exclusive prefix-sum, which is easy to see on the arrows in Figure 3.

```
Input: $u$ and $v$ of size $m$ base $B$
Output: $w$ of size $m$ in base $B$
Use: $f$ to extract carry from augmented carry

(r, a) = map2 ⊕ u v
c = scan_exc ⊗ e a
w = map2 (λ x y → x + f y) r c
```


a_0 a_1 a_2 $\cdots$ a_{m-1}
e→ ⊗ → ⊗ → ⊗ → ⊗ → ⊗
c_0 c_1 c_2 $\cdots$ c_{m-1}

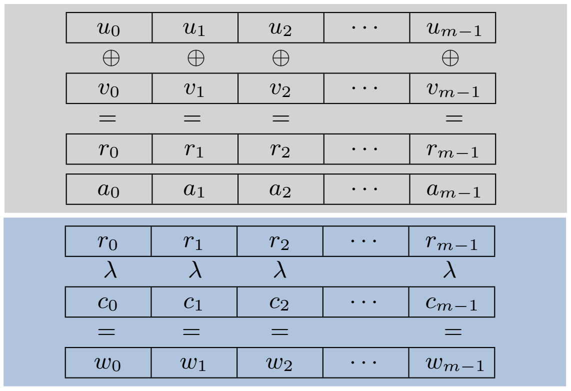

u_0 u_1 u_2 $\cdots$ u_{m-1}
⊕ ⊕ ⊕ ⊕
v_0 v_1 v_2 $\cdots$ v_{m-1}
= = = =
r_0 r_1 r_2 $\cdots$ r_{m-1}
a_0 a_1 a_2 $\cdots$ a_{m-1}
r_0 r_1 r_2 $\cdots$ r_{m-1}
λ λ λ λ
c_0 c_1 c_2 $\cdots$ c_{m-1}
= = = =
w_0 w_1 w_2 $\cdots$ w_{m-1}

Figure 4: Pseudo-code and illustration of parallel addition algorithm.

1. Compute the inner sums (`r`) and the augmented carries (`a`) by mapping operator $\oplus : \tau \to \tau \to (\tau,\ \tau')$ over the inputs. The augmented carries corresponds to the two conditions from [5].

2. Propagate the carries by computing the exclusive prefix sum over the augmented carries (`c`) using operator $\otimes : \tau' \to \tau' \to \tau'$ with left-associative neutral element $\texttt{e} : \tau'$.

3. Compress the propagated augmented carries using $f : \tau' \to \tau$ and distribute them over the inner sums.

Now, we must figure out what $\oplus$, $\otimes$, `e` and f is. Olesen et al. found $\oplus$ to augment the carries as a pair of boolean values, encoding the two conditions straightforward. I.e. for digits x and y of type `uint` in base B with wrap-around semantics on overflow, we have:

$$x \oplus y \coloneqq (x + y,\ (x + y < x,\ x + y == B - 1)) \tag{10}$$

The first element of the tuple encodes the actual overflows (condition I), and the second element is the augmented data needed for scanning (condition II). Hence, the compression function $f\ :\ (\texttt{bool},\ \texttt{bool}) \to \texttt{uint}$ is the projection π_1 followed by a type conversion.

The operator $\otimes$ then computes the prefix sum on the augmented carries s.t. if both the sum itself and the one just before it, is the maximum integer, then they remain the maximum integer combined.[6] Overflows are determined by the aforementioned conditions. I.e. for some pairs of booleans $x = (ov_x,\ mx_x)$ and $y = (ov_y,\ mx_y)$, we define:

$$x \otimes y \coloneqq (ov_x \land mx_y \lor ov_y,\ mx_x \land mx_y) \tag{11}$$

Olesen et al. has proven in [5] that $\otimes$ is associative with neutral element `e`:

$$\texttt{e} \coloneqq (\texttt{False},\ \texttt{True}) \tag{12}$$

[6]E.g. in decimal base, 9 is maximum of one digit and 99 remains maximum of two digits.

While this approach is intuitive and straightforward to implement in Futhark, hardware does not support boolean values natively - they are syntactic sugar for zero-and-nonzero integers. Thus, a low-level implementation will have to use integers, and so we might as well use bitwise operations over logical ones, as these are faster.

In turn, instead of using pairs of integers, with each pair using one indicator bit as boolean value, we combine them to one integer with two indicator bits. Not only does it halve the memory usage w.r.t. the prefix sum, it also increases memory utilization of threads, as each thread only have to fetch and write once. The formal definition of the optimizations is:

Definition 3 (*b*itwise-optimized parallel *add*ition algorithm (*badd*))**.** The bitwise-optimized operators of the algorithm in Figure 4, where the least and second least significant bit indicates integer overflow and maximum, respectively, are:

$$x \oplus y \coloneqq (r,\ \texttt{uint}(r < x) \mid (\texttt{uint}(r == B-1) \ll 1)), \quad \textbf{where } r = x + y \tag{13}$$

$$x \otimes y \coloneqq (((x \mathbin{\&} (y \gg 1)) \mid y) \mathbin{\&} 1) \mid (x \mathbin{\&} y \mathbin{\&} 2) \tag{14}$$

$$\texttt{e} \coloneqq 2 \tag{15}$$

$$f \coloneqq (\lambda\ x \to x \mathbin{\&} 1) \tag{16}$$

Equation (13) is a straightforward conversion of equation (10), with the pair being replaced by the shift- and bitwise or-operator, and Equation (16) extracts the first indicator bit.

Equation (14) is a conversion of equation (11), where I) the pair is replaced with bitwise or-operator, II) the first clause is checked in the least significant bit and zeroes out the second bit with "& 1", and III) the second clause is checked in the second least significant bit and zeroes out the first bit with "& 2". Associativity naturally still holds. Likewise, the neutral element `e` of Equation (15) is the corresponding indicator bits of Equation (12), and so `e` remains a left-associative neutral element of $\otimes$ after the optimizations.

For good measure, proof of associativity and neutral element are in Appendix A.

Thus, when using the *badd* algorithm presented in Definition 3, we get that both $\oplus$, $\otimes$ and `f` are $O(1)$. Therefore, the `maps` exhibit work $O(m)$ and depth $O(1)$, and the `scan` work $O(m)$ and depth $O(\log m)$. Hence, the total work is $O(m)$ and the depth $O(\log m)$.

As a final note, we can stay in the lane of bitwise optimizations when it comes to handling multiple instances per CUDA block. Extending the algorithm to work on a set of flattened instances is trivial using a *segmented* scan [7]. It is a scan with a flag array (indicating where the flattened segments begins) s.t. if we are at the beginning of a segment, the neutral element is used as the first operand rather than the left sum. We can implement it as a normal scan, but with an extended operator that checks for segment beginnings [9].

However, just like we combined the overflow and maximum indicator, we can also combine them with the flag indicator. Thus, we use the third least significant bit to indicate segment beginnings. Setting the flags as part of the augmented carries is a trivial extension to operator $\oplus$, and we get the segmented scan operator $\otimes$ (with neutral element remaining 2):

$$x \otimes y \coloneqq x_f \mid y_f \mid \begin{cases} y_v & \textbf{if } y_f \neq 0 \\ (((x_v \mathbin{\&} (y_v \gg 1)) \mid y_v) \mathbin{\&} 1) \mid (x_v \mathbin{\&} y_v \mathbin{\&} 2) & \textbf{otherwise} \end{cases} \tag{17}$$
$$\textbf{where } \; x_f \coloneqq (x \mathbin{\&} 4), \quad x_v \coloneqq (x \mathbin{\&} 3), \quad y_f \coloneqq (y \mathbin{\&} 4), \quad y_v \coloneqq (y \mathbin{\&} 3)$$

7.2 CUDA Implementation

In this section we introduce three versions of addition in CUDA. The first version (`V1`) is the fundamental implementation that follows the algorithm presented in Figure 4 closely.

The second version (`V2`) attempts to efficiently sequentialize the parallelism in excess: Since the operation runs in sub-linear depth (with two of the three steps being a constant depth), we expect the threads to have an excess amount of parallelism spent on communication (waiting for memory and synchronizations). Increasing the sequentialization factor decrease the amount of communication, and in this case, increase the performance. This optimization comes with the added benefit of handling integers of size greater than a CUDA block.

The third version (`V3`) can handle multiple instances per block, giving better performance for many arithmetic instances with small integer sizes, as mentioned in section 6.

We keep the sequentialization factor (q) and number of instances per block (ipb) parametric using C++ templates. First of all, this allows us to handle all input sizes generically rather than relying on multiple hand-written versions, but more importantly, it allows us to experiment with optimal parameters. A purpose of the CUDA implementation is to determine how much performance is possible to achieve using our designated algorithms. Hence, experimentation in CUDA directly influence how we approach the implementations in Futhark.

Generic ipb is trivial, because it only serves to determine segment beginnings of segmented algorithms. A generic sequentialization factor can impedes performance, as we achieve this by introducing `loop`-constructs invariant to q. However, a benefit of high-level languages like C++ is compiler directives, allowing us to unroll loops as if they were hand-written [1].

Overall, we determine q, ipb, and kernel dimensions with the code of Listing 1 below. The sequentialization factor of `V2` and `V3` is by default $q = 4$, but if the size of the inputs still exceeds a CUDA block, we use the rule $(a + b - 1)/b = \lceil a/b \rceil$ to round up to nearest sequentialization factor that fits. Likewise, for `V3` we write $ipb = \lceil 256/(m/q) \rceil$, which rounds the number of threads per block up to 256.[7]

[7]Experimentally, we found a minimum of 256 threads per block and a sequentialization factor of 4 to be

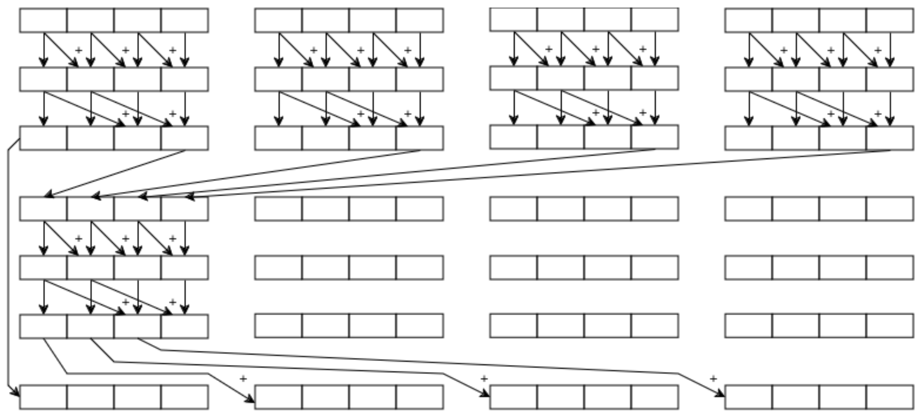


Figure 5: Illustration of a warp-level inclusive scan with warp-size 4 and 16 threads

Listing 1: CUDA addition parameters and kernel dimensions for version v with size m and *num_instances*.

```
const uint32_t q = (v == 1) ? 1 : (m/4 <= 1024) ? 4 : (m+1024-1) / 1024;
const uint32_t ipb = (v == 3) ? (256 + m/q - 1) / (m/q) : 1;
dim3 block(ipb*(m/q), 1, 1);
dim3 grid (num_instances/ipb, 1, 1);
```

Steps 1. and 3. of *badd* are implemented straightforward in all three versions (because the `map`s are implicit in CUDA). Step 2. on the other hand, requires more thought:

For the `scan` function, we use a work-efficient warp-level inclusive scan [16]. The idea is to minimize communication by exploiting that warps execute in lockstep. Figure 5 contains an illustration of this algorithm. It works in three steps with synchronization in-between:

I Each warp scans its elements in log *warpsize* iterations.

II The end-of-warp results are put in the first warp, synchronized, and then scanned.

III The results of II are fetched, synchronized, and distributed over the results I.

To convert an inclusive to an exclusive scan, we shift the result to the right by 1 and insert the neutral element at the left-most position. Step 2. of *badd* use this warp-level exclusive scan with the operator $\otimes$ and neutral element `e` defined in Equations (14), (15), and (17).

What we have so far is enough to write `V1`, but `V2` and `V3` requires some extra attention. We cannot directly use the warp-level scan, because this works for one element per thread and we have q. Instead, let us reuse the *scan* → *end-of-result* → *scan* → *distribute* idea from the warp-level scan algorithm on a register-level basis:

the most efficient kernel parameters for addition.

Let each thread compute the inclusive prefix sum of its q elements sequentially, and place only the last sum in shared memory. Then, run a warp-level exclusive scan over the end-of-register-level sums in shared memory. Lastly, each thread sequentially distribute the results of the warp-level scan over the result of the register-level scan.

Using this idea, we are now able to write all three versions. For brevity, we only show the main kernel body of `V3`, but both `V1` and `V2` are principally implemented the same way. Listing 2 contains this kernel, where the input/output elements of each thread are read coalesced beforehand and written coalesced afterwards to/from global memory.

Listing 2: CUDA *badd* `V3` implementation body from file `ker-add.cu.h` (slightly edited), where registers `ass`, `bss`, and `rss` contains q input/output digits, respectively, of type `uint_t` with base class `Base`, segmented scan operator class `SegCarryProp` over carry type `carry_t`, and shared memory buffer `shmem`.

```
const bool new_segm = threadIdx.x % (m/q) == 0;

// 1. compute result, carry, register-level scan, and segment flags
uint_t css[q];
carry_t acc =
    new_segm ? SegCarryProp<Base>::setFlag(SegCarryProp<Base>::identity())
             : SegCarryProp<Base>::identity();
#pragma unroll
for(int i=0; i<q; i++) {
    rss[i] = ass[i] + bss[i];
    css[i] = ((carry_t) (rss[i] < ass[i]))
             | (((carry_t) (rss[i] == Base::HIGHEST)) << 1);
    acc = SegCarryProp<Base>::apply(acc, css[i]);
}
shmem[threadIdx.x] = acc;
__syncthreads();

// 2. propagate carries
acc = scanExcBlock< SegCarryProp<Base> >(shmem, threadIdx.x);
acc = new_segm ? SegCarryProp<Base>::identity() : acc;

// 3. add carries to results
#pragma unroll
for(int i=0; i<q; i++) {
    rss[i] += (acc & 1);
    acc = SegCarryProp<Base>::apply(acc, css[i]);
}
```

To reflect on the implementation; C++ is verbose by design (as a C-like language), and modelling the *badd* algorithm requires care and thought. Due to the nature of the CUDA extension, we furthermore have to reason about the semantics in parallel, e.g. in the way we compute segment beginnings. However, structuring the implementation around the high-level C++ features of classes and templates, not only makes it generic and readable, but also extensible in terms of optimizations. Once the fundamental version is designed with proper usage of C++ constructs, the optimizations comes as natural extensions, while still allowing fine-grained control over the important aspects, such as memory usage.

7.3 Futhark Implementation

In this section, we introduce four versions of addition in Futhark. The first version (`V0`) is the original version formulated by Olesen et al. [5], included for completeness. The second version (`V1`) is the straightforward *badd* algorithm.

The third version (`V2`) introduces a fixed sequentialization factor of 4, which we found to be the optimal factor from experimenting with the CUDA implementation (see section 7.2). In CUDA, it is arbitrary to make this factor generic using C++ templates and compiler directives, without loss of performance. In Futhark however, while we have type parameters and loop unroll attributes, we do not have the fine-grained control of C++ that allows us to e.g. decide when and how to store intermediate results in shared memory. We found that parameterizing the sequentialization factor is significantly slower than fixing it. In turn, integers with size $m > 4096$ does not fit in a CUDA block, and requires manually increasing the factor. Another approach could be to write a parameterized version for integers with $m > 4096$, and search for tricks that optimizes the compilers generated output. For simplicity, we fix the factor at 4.

The fourth version (`V3`) introduces a parameterized number of instances per block (ipb) and the segmented *badd*. This factor is fine to vary, since it is only used to determine boolean values of the segmented semantics.

Version `V0` and `V1` are defined straightforward, and due to Futharks high-level design based on parallelism, they look almost identical to the pseudocode of Figure 4. E.g. Listing 3 contains the implementation of `V1`:

Listing 3: Futhark *badd* `V1` using base `ui` and carry type `ct` (from file `add.fut` slightly edited).

```
def carryProp (c1: ct) (c2: ct) : ct =
  (c1 & c2 & 2) | (((c1 & (c2 >> 1)) | c2) & 1)

def carryPropE : ct = 2

def carryAug (r : ui) (a : ui) : ct =
  (boolToCt (r < a)) | ((boolToCt (r == HIGHEST)) << 1)

def baddV1 [m] (us: [m]ui) (vs: [m]ui) : [m]ui =
  -- 1. compute sums and carries
  let (ws, cs) = map2 (\ u v -> let w = u+v in (w, carryAug w u)) us vs
                 |> unzip
  -- 2. propagate carries
  let pcs = scanExc carryProp carryPropE cs
  -- 3. add carries to sums
  in map2 (\ w c -> w + fromCt (c & 1)) ws pcs
```

While versions `V0` and `V1` feeds into the strengths of Futhark (simple abstract functions that run comparatively well), versions `V2` and `V3` reveals the weaknesses. As mentioned, they include a fixed sequentialization factor of 4, so we want to compute four sums for each thread. It is therefore crucial that we pre-fetch the inputs to either shared or register memory, greatly reducing the amount of reads from global memory (if fetched coalesced). However, while easily done in our CUDA implementation, we are at the mercy of the Futhark compiler to perform this optimization, and as of writing this thesis, it does not.

Instead, we may exploit that the compiler stores intermediate arrays in shared memory to e.g. read input u into w in a coalesced fashion. Normally, we avoid such redundant computations because the compiler constructs and copies to a new arrays when resolving array operations (, since data is immutable as per usual for functional languages). However, the Futhark compiler use a memory optimization strategy called array short-circuiting, where it performs a short-circuit analysis that reveals whether it can skip constructing intermediate arrays and execute the operation in-place instead [17].

Thus, when we construct the read operation in a particular way, the compiler will pre-allocate the inputs in shared memory when fused with the surrounding functions, and in turn, generate the read operation with no-ops. We use the code of Listing 4 below, which utilizes short-circuiting to read the inputs into shared memory coalesced and with minimum overhead. I.e. the `zips`, `unzips` and concatenations of the code becomes no-ops, and `ush` and `vsh` is in-place read from `us` and `vs` (occupying the same memory region).

Listing 4: Futhark code snippet reading `us` and `vs` of size `ipb`·`m` from global to shared memory coalesced utilizing array short-circuiting (from file `add.fut` slightly edited).

```
let cp2sh (i : i32) = #[unsafe]
  let str = i32.i64 (ipb*m)
  in ((us[i], us[str + i], us[2*str + i], us[3*str + i])
     ,(vs[i], vs[str + i], vs[2*str + i], vs[3*str + i]))

let (uss, vss) = (0..<ipb*m) |> map i32.i64 |> map cp2sh |> unzip
let (u1s, u2s, u3s, u4s) = unzip4 uss
let (v1s, v2s, v3s, v4s) = unzip4 vss
let ush = u1s ++ u2s ++ u3s ++ u4s
let vsh = v1s ++ v2s ++ v3s ++ v4s
```

The compiler attribute `#[unsafe]` tells the compiler not to perform any memory bound checks or optimizations. We generally add this attribute when a sequentialization factor is involved, just to be certain that the compiler takes our array indexing at face value.

Now, let us discuss the implementation of `V2` and `V3`. Their type signature includes the shape parameters, enforcing the memory layout of the inputs to be aligned:

```
def baddV2 [m] (us: [4*m]ui) (vs: [4*m]ui) : [4*m]ui
def baddV3 [ipb][m] (us: [ipb*(4*m)]ui) (vs: [ipb*(4*m)]ui) : [ipb*(4*m)]ui
```

The functions are defined s.t. they first copy from global to shared memory (as described above), and then run the function body (corresponding to the *badd* algorithm). For brevity, we focus on `V3` since they are principally identical. The function body sequentializes the excess parallelism using tuples of four elements as a manually unrolled loop. It exhibit the same sequentialized register-level prefix sum approach as the CUDA `V3` implementation of Listing 2. The function body is in Listing 5 below.

Listing 5: Futhark *badd* `V3` main function body using base `ui` and carry type `ct` from file `add.fut`.

```
let baddV3Run (us: []ui) (vs: []ui) : []ui = #[unsafe]
  -- 1. compute sums, carries, flags, and register-level prefix sum
  let (ws, cs, accs) = unzip3 <| imap (0..<ipb*m)
    (\ i -> let (u1, u2, u3, u4) = (us[i*4],us[i*4+1],us[i*4+2],us[i*4+3])
            let (v1, v2, v3, v4) = (vs[i*4],vs[i*4+1],vs[i*4+2],vs[i*4+3])
            let (w1, w2, w3, w4) = (u1 + v1, u2 + v2, u3 + v3, u4 + v4)
            let (c1, c2, c3, c4) = (carryAug w1 u1, carryAug w2 u2,
                                    carryAug w3 u3, carryAug w4 u4)
            let c1 = (boolToCt (i % m == 0)) << 2 | c1
            let acc = carryProp c1 <| carryProp c2 <| carryProp c3 c4
            in ((w1, w2, w3, w4), (c1, c2, c3, c4), acc))

  -- 2. propagate carries
  let pcs = scanExc carryPropSeg carryPropE accs

  -- 3. distribute carries over register-level prefix sum, and add to sum
  let (wi1s, wi2s, wi3s, wi4s) = unzip4 <| imap4 ws cs pcs (0..<ipb*m)
    (\ (w1, w2, w3, w4) (c1, c2, c3, _) acc1 i ->
       let acc1 = if i % m == 0 then carryPropE else acc1
       let acc2 = carryProp acc1 c1
       let acc3 = carryProp acc2 c2
       let acc4 = carryProp acc3 c3
       in ((w1 + fromCt (acc1 & 1), i*4),  (w2 + fromCt (acc2 & 1),i*4+1),
           (w3 + fromCt (acc3 & 1), i*4+2),(w4 + fromCt (acc4 & 1),i*4+3)))
  let (ws, inds) = unzip <| wi1s ++ wi2s ++ wi3s ++ wi4s
  in scatter (replicate (ipb*(4*m)) 0) inds ws
```

While `V2` and `V3` are still somewhat succinct, it is clear that manipulating the compiler for certain optimizations introduces a lot more code compared to the fundamentals of `V1`. This can especially become cumbersome given the hand-written nature of the sequentialization factor. On the other side, most of the Futhark code is still modelled closely to the algorithm, and thus, making it easy to follow, reason about the semantics, and spot potential errors.

7.4 Subtraction

Subtraction is also a basic of arithmetic. While we do not use signs in our underlying representation, we can define subtraction using the addition. In fact, it is easy to see that the complement of each subroutine of the addition gives us subtraction:

Definition 4 (big integer subtraction)**.** Subtraction of two big integers can be obtained from the generic algorithm of Figure 4, where line 3 subtracts rather than adding the carry, and operators $\otimes$, $\mathtt{e}$, and f remains the same as in *badd*. The operator $\oplus$ then becomes:

$$x \oplus y \coloneqq (x - y > x) \mid ((x - y = 0) \ll 1) \tag{18}$$

This definition has the usual unsigned integer wrap-around semantics. If we are interested in the difference, we can check which input is the largest (by a `map-reduce` composition), and then subtract and return the sign accordingly.

Since it is the same algorithm as addition, we does not benchmark or discuss it further.

8 Classical Multiplication

Multiplication is next of the basic arithmetic operators. It is more complex than addition and subtraction, classically requiring each digit of the multiplicand to be multiplied by all digits of the multiplier, which leading to asymptotically quadratic work. Other well known algorithms of sub-quadratic work exists, namely Karatsuba, Toom-Cook, and FFT-based multiplication, all mentioned by Knuth and part of GMP [11, 14]. Karatsuba and Toom-Cook are viewed as practical algorithms, but are usually modelled by recursion making them less straightforward for GPGPU. FFT multiplication is an involved process consisting of multiple data structure conversions (and possibly leaving the domain of integers), but has shown to run fast on GPGPU [6, 19]. While classical multiplication has quadratic work, it is expected to run faster for smaller-sized integers due to the complexities involved with the sub-quadratic algorithms. This thesis focus on the classical approach.

This section is structured as follows: In 8.1 we present a classical quadratic multiplication algorithm that convolutes the digits of the operands, and we discuss how to parallelize it. In 8.2 and 8.3 we present our CUDA and Futhark implementations, respectively, and any further optimizations related to the implementations. Lastly, in 8.4 we cover the special case of multiplication by a single precision factor, and present an algorithm that shares the behaviour and efficiency of an addition operation.

8.1 Algorithm

The classical multiplication algorithm handles one digit of the multiplier at a time. E.g. in decimal, $42 \cdot 21$ becomes $1 \cdot 42 \cdot 10^0 + 2 \cdot 42 \cdot 10^1$. For our big integers, we have:

Definition 5 (classical multiplication)**.** Multiplying integer $u \in \mathbb{N}$ by $v \in \mathbb{N}$ in the positional number system, with base B and m digits, is classically decomposed to:

$$u \cdot v = \sum_{i=0}^{m-1} u_i \left(\sum_{j=0}^{m-1} v_j \cdot B^j \right) B^i \tag{19}$$

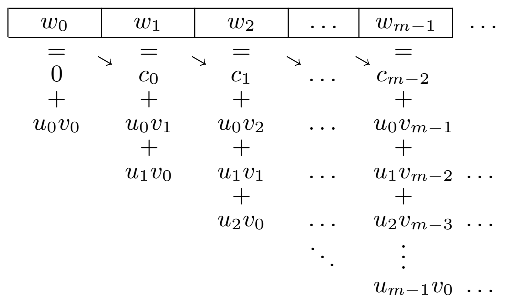


Figure 6: Visualization of classical multiplication as stated in Definition 5 with the outer sum as rows.

The definition is illustrated in Figure 6. We make two observations:

– The multiplicand v appears inside the outer sum of the definition in (19).

– Only the first m sums of the illustration contributes to the result in our setting.[8]

The former tells us that we cannot compute the outer sum using register arithmetic. However, from the latter, we can redefine the terms of the outer sum by exploiting the tiling of digits that is apparent in the illustration. We refer to the columns as *convolutions*, e.g. the convolution of w_1 is the sum $c_0 + u_0v_1 + u_1v_0$, and each arithmetic of a convolution fits in register. Hence, we arrive at the following definition:

Definition 6 (tiling of classical multiplication by convolutions preserving input shape)**.** Assuming the product of Definition 5 is truncated to m digits, it can be redefined to:

$$u \cdot v = \sum_{k=0}^{m-1} \left(\sum_{\substack{0 \leq i,j < m \\ i+j=k}} u_i \cdot v_j \right) B^k \tag{20}$$

A sequential algorithm is straightforward to construct from this definition. In contrast, we make three new observations to parallelize it:

– The products across all convolution are unique, i.e. the $O(m^2)$ products of the inner sum in Equation (20) have to be computed.

– The number of terms constituting a convolution (i.e. the work) is linear in k.

– The overflows can be propagated as a separate step after computing the convolutions (e.g. using *badd*).

[8]The result of multiplication is up to twice the size of the inputs. In order to not loose precision in our setting, a memory region of double size should be preallocated before the multiplication function call.

For simplicity, let us assume we have m threads available.[9] The first observation tells us that each thread must compute $O(m)$ inner products with no parallelism. It would seem natural to let thread $t_{l\in\{0,..,m-1\}}$ compute convolution l, but from the second observation, this gives an unbalanced amount of sequential work amongst threads. However, we also derive from the second observation that the combined work of convolution 0 and convolution $m-1$ is $m+1$, and is equal to that of convolution 1 and $m-2$, and to that of 2 and $m-3$, etc.[10] Thus, by introducing a fixed sequentialization factor of 2, we can balance the work amongst threads, s.t. thread $t_{l\in 0,..,(m/2)-1}$ computes convolution l and convolution $m-1-l$.

Regarding the third observation; in order to propagate the overflows in a separate sweep, we must keep track of them while computing the convolutions. It is a known rule that; given any two factors, their product fit in their combined size. E.g. in decimal, 9 (size 1) times 99 (size 2) is 891 (size 3). By this rule, each product of a convolution fits in two machine words. We denote the least significant word as the *low* part and the most significant as the *high* part. Now, each convolution is the sum of $O(m)$ low and high parts. Adding the low parts overflows to the high part, and adding the high part overflows to the *carry* part – yet another machine word (assuming the base is big enough to hold the carry part). Hence, we need three words to keep track of each convolution.

Thus, we derive the following parallel algorithm with work $O(m^2)$ and depth $O(m)$:

Definition 7 (work balanced parallel algorithm for classical *mul*tiplication by *conv*olution (*convmul*))**.** We parallelize Definition 5 according to the pseudocode in Figure 7. Lines 1-18 describes the main function that, given a thread index $t \in \{0, .., (m/2) - 1\}$, computes a low, high, and carry part of both convolution t and convolution $m-1-t$. Lines 20-25 calls the main function on each thread and adds the convolution parts according to following memory layout, where color and superscript denotes the thread that has computed the part, and subscript denotes the convolution that the part originates from:

l_0^0	l_1^1	l_2^2	l_3^3	$\dots$	l_{m-4}^3	l_{m-3}^2	l_{m-2}^1	l_{m-1}^0
$+$	$+$	$+$	$+$		$+$	$+$	$+$	$+$
0	h_0^0	h_1^1	h_2^2	$\dots$	h_{m-5}^4	h_{m-4}^3	h_{m-3}^2	h_{m-2}^1
$+$	$+$	$+$	$+$		$+$	$+$	$+$	$+$
0	0	c_0^0	c_1^1	$\dots$	c_{m-6}^5	c_{m-5}^4	c_{m-4}^3	c_{m-3}^2

While we cannot optimize the algorithm asymptotically, we can eliminate one of the additions, decreasing the amount of communication. The intuition is to compute two lower and two upper convolutions per thread – four in total. This allows to prematurely compute half of the additions before the parts leave register memory (i.e. in the function `CONV` of Figure 7), and thus, necessitating only one addition after running the convolutions. The optimization not only reduces the communication, but also reduces shared memory

[9]Since we parallelize at block-level, we have $m/q \leq 1024$ threads, where q is the sequentialization factor.
[10]Easy to see on the pattern visualized in Figure 6

```
fun CONV t = -- Parameter ‘t‘ represents current index of the m/2 threads
    k1 = t
    k2 = m - 1 - k1              -- The indices ‘k1‘ and ‘k2‘ represents the
    l1, l2, h1, h2, c1, c2 = 0 -- upper and lower ‘k‘ handled by thread ‘t‘

    for i in (0..k1)         -- The indices ‘i‘ and ‘j‘ are computed
        j = k1 - i           -- straightforward w.r.t. Equation (20)
        l1 += u[i] $*_{low}$ v[j]
        h1 += u[i] $*_{high}$ v[j] + overflow$_{l1}$
        c1 += overflow$_{h1}$

    for i in (0..k2)
        j = k2 - i
        l2 += u[i] $*_{low}$ v[j]
        h2 += u[i] $*_{high}$ v[j] + overflow$_{l2}$
        c2 += overflow$_{h2}$

    return (l1, l2, h1, h2, c1, c2)

(l1, l2, h1, h2, c1, c2) = map CONV (0..(m/2)-1)
l = concat l1 (reverse l2)            -- The second half of the
h = shift 1 (concat h1 (reverse h2)) -- convolutions are computed in
c = shift 2 (concat c1 (reverse c2)) -- reverse order due to work balancing
r = ADD l h
w = ADD r c
```

Figure 7: Pseudocode of work balanced parallel algorithm for classical multiplication by convolution of big integers. `Input:` u and v of size m. `Output:` w of size m. `Use:` Big integer addition function `ADD`.

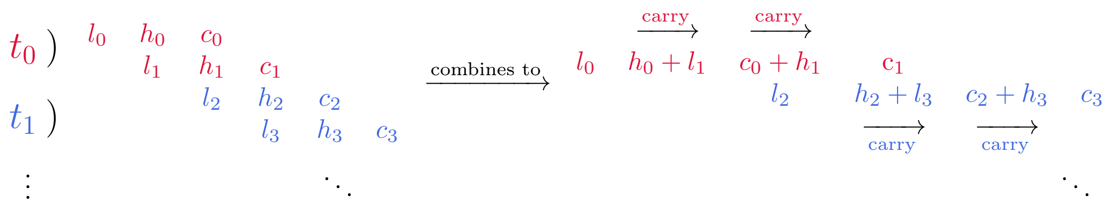


Figure 8: Illustration of optimized memory layout of *convmul* with sequentialization factor of 4, where the left-hand-side shows the low, high, and carry parts of the first four convolutions, and the right-hand-side shows how to combine six parts to four on a thread-level.

usage by better utilizing registers. The downside is that it further increase the amount of sequential work within a thread by a factor of 2. Hence, whether the optimization will pay off depends on the batch size and number of available threads – or if the inputs are too big to fit a CUDA block with sequentialization factor of 2.

Figure 8 contains an illustration of the tiled convolution layout of this optimization w.r.t. threads. Now, compared to the pseudocode of Figure 7, a thread computes a total of twelve parts in the convolution function `conv`, but combines them to eight parts before returning – four for the lower convolution and four for the upper. In turn, the memory layout is trickier to construct, requiring threads to alternate in writing pairs of their parts to the additive arrays. E.g. with four threads we get that the final parallel addition of the convolution-results corresponds to the following memory layout, where color and superscript denotes the thread that has computed the part, and subscript denotes the part number:

t^0_{00}	t^0_{01}	t^1_{00}	t^1_{01}	t^2_{00}	t^2_{01}	t^3_{00}	t^3_{01}	t^3_{10}	t^3_{11}	t^2_{10}	t^2_{11}	t^1_{10}	t^1_{11}	t^0_{10}	t^0_{11}
+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
0	0	t^0_{02}	t^0_{03}	t^1_{02}	t^1_{03}	t^2_{02}	t^2_{03}	t^3_{02}	t^3_{03}	t^3_{12}	t^3_{13}	t^2_{12}	t^2_{13}	t^1_{12}	t^1_{13}

The optimized memory layout perfectly partitions into a parallel addition operator with sequentialization factor of 4, and thus, yielding both the aforementioned upsides and a faster addition.[11] With a sequentialization factor of $q = 4$ rather than $q = 2$, it can also process integers at CUDA block-level of up to double the size. To go beyond $q = 4$, it is speculated wiser to stack this optimizations rather than combining all the consecutive sequential parts, as more combinations reduce the effectiveness of the final parallel addition.

8.2 CUDA Implementation

In this section, we introduce five CUDA versions. The first version (`V1`) implement the *convmul* algorithm straightforward, with a sequentialization factor of 2. The second version (`V2`) introduces a data type optimization w.r.t. doubling the word-size for keeping track of convolutions inside loops. This optimization effectively minimizes the amount of work in loops, by delaying the propagation of overflows until after the loop has run.

The third version (`V3`) introduces multiple instances per block to the approach of `V2`. The fourth version (`V4`) implements a sequentialization factor of 4 according to the optimization that combines consecutive convolutions, and the fifth version (`V5`) expands `V4` with multiple instances per block.

We decide the CUDA kernel dimensions and parameters with the code in Listing 6. Akin to Listing 1 regarding addition, but with the block size rounded up to 128 rather than 256 (which we experimentally found to be faster). The sequentialization factor does not grow

[11]Experimentally, we found addition with sequentialization factor of 4 to be the fastest (see section 7).

beyond 4, but, as mentioned at the end of section 8.1, one could implement a version with a sequentialization factor that is a multiple of 4. Hence the CUDA implementations are limited to integers of size $m \leq 4096$.

Listing 6: CUDA multiplication parameters and dimensions for version v with size m and $num\_instances$.

```
const uint32_t q = (v >= 1 && v <= 3) ? 2 : 4;
const uint32_t ipb = (v == 3 || v == 5) ? (128 + (m/q) - 1) / (m/q) : 1;
dim3 block(ipb*(m/q), 1, 1);
dim3 grid (num_instances/ipb, 1, 1);
```

This section is structured into three paragraphs that describes the implementation of convolutions, memory layouts, and multiple instances per block, respectively. Combined, they establish how the five CUDA kernels are implemented.

Convolutions. The implementations use the type `ubig_t` of the generic base class – a type that use twice as many bits as the base `uint_t`. We use the type to compute the inner products of the convolutions, but it also allows the optimization of `V2`: The idea is to postpone the combining of the low and high parts of products until after the sequential loop. Then, the carries can be propagated once, rather than i times for convolution k_i. The first two convolutions of Listings 7 and 8 shows the difference in storing intermediate values of the convolution in type `uint_t` (by a low, high, and carry part) and in type `ubig_t` (by a low and high part only).

The third convolution of Listing 8 shows how to compute and combine two consecutive convolutions, used in the optimized version with sequentialization factor of 4. The CUDA listing also reveals another upside of this optimization; by combining the two consecutive convolutions into one loop-construct, we can reuse a digit of the multiplier u, and thus, reduce memory usage further.

The three presented convolutions are used to define `V1`, `V2`, and `V4`, respectively.

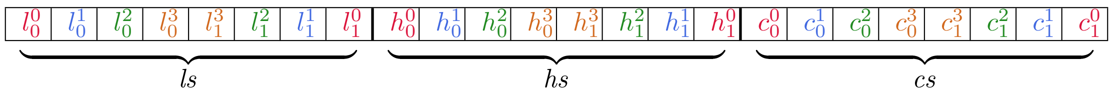
Listing 7: The three convolutions used in the CUDA multiplication kernels of file `ker-mul.cu.h`. The listing continues in Listing 8 on the next page.

```
• Convolution over index k as described in Definition 7
    for (int i=0; i<=k; i++) {
        // compute high and low part of product
        int j = k - i;
        ubig_t uv = ((ubig_t) u[i]) * ((ubig_t) v[j]);
        uint_t l = (uint_t) uv;
        uint_t h = (uint_t) (uv >> Base::bits);
        // update lows, highs, and carries
        ls += l;
        hs += h + (ls < l);
        cs += hs < (h + (ls < l));
    }
Continued in Listing 8 on next page ...
```

Listing 8: The three convolutions used in the CUDA multiplication kernels of file `ker-mul.cu.h`. Continuation of Listing 7 on the previous page.

```
... Continuation of Listing 7 from previous page.

• Convolution over index k optimized for data type ubig_t
    ubig_t l = 0;
    ubig_t h = 0;
    for (int i=0; i<=k; i++) {
        // compute high and low part of product
        int j = k - i;
        ubig_t uv = ((ubig_t) u[i]) * ((ubig_t) v[j]);
        l += uv & ((ubig_t) Base::HIGHEST);
        h += uv >> Base::bits;
    }
    // update lows, highs, and carries
    ls = (uint_t) l;
    hs = ((uint_t) h) + ((uint_t) (l >> Base::bits));
    cs = ((uint_t) (h >> Base::bits)) + (hs < ((uint_t) h));

• Convolution over index k and k+1 optimized for ubig_t and sequentialization factor of 4
    ubig_t l1 = 0; ubig_t h1 = 0;
    ubig_t l2 = 0; ubig_t h2 = 0;
    for (int i=0; i<=k; i++) {
        // compute high and low part of product
        int j = k - i;
        ubig_t uv1 = ((ubig_t) u[i]) * ((ubig_t) v[j]);
        ubig_t uv2 = ((ubig_t) u[i]) * ((ubig_t) v[j+1]);
        l1 += uv1 & ((ubig_t) Base::HIGHEST);
        h1 += uv1 >> Base::bits;
        l2 += uv2 & ((ubig_t) Base::HIGHEST);
        h2 += uv2 >> Base::bits;
    }
    // remaining computation where i = k+1
    ubig_t uv = ((ubig_t) u[k+1]) * ((ubig_t) v[0]);
    l2 += uv & ((ubig_t) Base::HIGHEST);
    h2 += uv >> Base::bits;
    // update lows, highs, and carries
    ls1 = (uint_t) l1;
    hs1 = ((uint_t) h1) + ((uint_t) (l1 >> Base::bits));
    cs1 = ((uint_t) (h1 >> Base::bits)) + (hs1 < ((uint_t) h1));
    ls2 = (uint_t) l2;
    hs2 = ((uint_t) h2) + ((uint_t) (l2 >> Base::bits));
    cs2 = ((uint_t) (h2 >> Base::bits)) + (hs2 < ((uint_t) h2));
    // combine the lows, highs and carries
    ls = ls1;
    hsls = hs1 + ls2;
    cshs = cs1 + hs2 + (hsls < hs1);
    cscs = cs2 + (cshs < hs2);
```

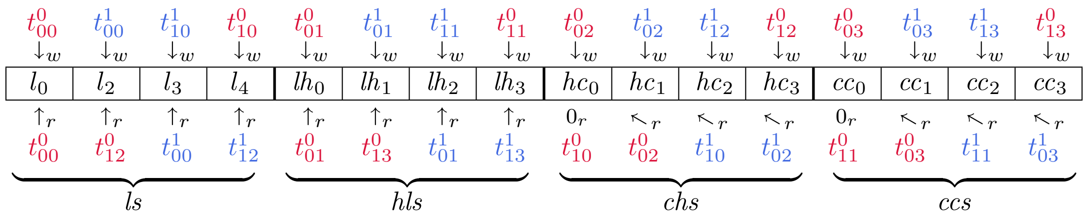
Memory layout. Constructing the memory layout of *convmul* with sequentialization factor of $q = 2$ (Definition 7) is straightforward; the shared memory is partitioned in three, and each thread write their two convolution parts to each of the three memory partitions. The writes are coalesced by design. E.g. for eight digits and four threads, where color and superscript denotes thread, and subscript denotes convolution, the writes are:


l^0_0 l^1_0 l^2_0 l^3_0 l^3_1 l^2_1 l^1_1 l^0_1
h^0_0 h^1_0 h^2_0 h^3_0 h^3_1 h^2_1 h^1_1 h^0_1
c^0_0 c^1_0 c^2_0 c^3_0 c^3_1 c^2_1 c^1_1 c^0_1
ls
hs
cs

Now, reading and adding the three sub-arrays in memory is trivial using *badd* with $q = 2$.

On the contrary, we get a complicated memory layout (in order to read and write coalesced) when optimizing *convmul* for $q = 4$. Since we compute four parts per convolution, we partition the shared memory in four and write the convolution results to the partitions, coalesced. Reading from the first two partitions is straightforward, but reading from the last two partitions is offset by -1 s.t. if the index is out-of-bound of the partition (i.e. the first thread is reading), the constant 0 is read instead.

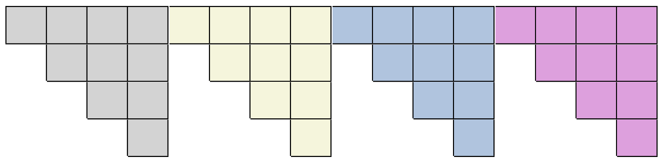
E.g. for eight digits and two threads, we get a shared memory buffer of size 16 and partition it to the four buffers *ls*, *hls*, *chs*, and *ccs*. Now, in order to add the convolution parts and access memory coalesced, the reading and writing of shared memory follows this pattern, where color and superscript denotes thread and subscript denotes convolution (register):


t^0_{00} t^1_{00} t^1_{10} t^0_{10} t^0_{01} t^1_{01} t^1_{11} t^0_{11} t^0_{02} t^1_{02} t^1_{12} t^0_{12} t^0_{03} t^1_{03} t^1_{13} t^0_{13}
$\downarrow_w$
l_0 l_2 l_3 l_4 lh_0 lh_1 lh_2 lh_3 hc_0 hc_1 hc_2 hc_3 cc_0 cc_1 cc_2 cc_3
$\uparrow_r$ 0_r $\nwarrow_r$
t^0_{00} t^0_{12} t^1_{00} t^1_{12} t^0_{01} t^0_{13} t^1_{01} t^1_{13} t^0_{10} t^0_{02} t^1_{10} t^1_{02} t^0_{11} t^0_{03} t^1_{11} t^1_{03}
ls
hls
chs
ccs

Afterwards, the registers of threads contains four pairs of elements to be added, and thus, can straightforward use *badd* with sequentialization factor of 4 and values pre-fetched to registers – corresponding to the addition at the end of section 8.1. Listing 20 in Appendix B contains the CUDA code for reading and writing in this pattern.

The two memory layouts are used to define `V1` & `V2` and `V4`, respectively.

Multiple instances per block. When handling multiple arithmetic instances in a CUDA block at the same time, the symmetrical pattern remains w.r.t. the work of convolutions. E.g. for four big integer instances that each consists of four digits, where each instance is denoted by a color, we have the following convolution pattern:



Hence, the work-balancing pattern still applies (i.e. indexing from both ends in a thread).

Thus, we use the same convolution pattern to define `V3` and `V5` as we use to define `V2` and `V4`, respectively. In turn, the memory layouts from `V2` and `V4` applies to `V3` and `V5` as well. However, we have to be careful when reading from the memory buffer of the second additive array, as shifting the layout is no longer sufficient to get the correct tiling. Instead, we compute the thread index w.r.t. segments s.t. the first thread of a segment fetch 0 rather than indexing to the previous segment. The aforementioned CUDA code for the memory layout of `V4` (Appendix B Listing 20) includes this segment-beginning check.

8.3 Futhark Implementation

The implementation of *convmul* in Futhark has one significant change from CUDA: Futhark does not support 128-bit integers, thus, there may not be a double-size type for the base type, and hence, we cannot apply the optimization of postponing the carry propagation. In turn, we have three degrees of optimization in Futhark: The first version (`V1`) implements the algorithm with sequentialization factor of 2, the second (`V2`) with a factor of 4, and the third (`V3`) allows segmented multiplication. The type signature of the three versions guarantees that the input memory layout matches the algorithm:

```
def convMulV1 [m] (us: [2*m]ui) (vs: [2*m]ui) : [2*m]ui
def convMulV2 [m] (us: [4*m]ui) (vs: [4*m]ui) : [4*m]ui
def convMulV3 [ipb][m] (us:[ipb*(4*m)]ui) (vs:[ipb*(4*m)]ui): [ipb*(4*m)]ui
```

First we present how to compute a convolution iteration. Since there may not be a double type, we instead use the function `mul_hi` to compute the high part of a multiplication. Listing 9 defines this, where the low, high, and carry parts are represented as a triple:

Listing 9: Futhark function to compute a convolution iteration from file `mul.fut`.

```
def iterate (l: ui, h: ui, c: ui) (u: ui) (v: ui) : (ui, ui, ui) =
  -- compute the low and high part of result
  let lr = u * v
  let hr = mulHigh u v
  -- update l, h and c
  let ln = l + lr
  let hn = h + hr + (fromBool (ln < l))
  let cn = c + (fromBool (hn < h))
  in (ln, hn, cn)
```

It is now straightforward to define `V1` with `iterate` and the `loop`-construct of Futhark. It looks almost identical to the pseudocode of *convmul* (Figure 7), except that it copies to shared memory beforehand using the technique explained in Listing 4 of section 7.3. Hence, `V1` is not listed. Note; our Futhark implementations of *badd* does not include a version with sequentialization factor of 2. The general version is applied (i.e. `V1`), and thus, it is in the hand of the Futhark compiler to efficiently map it to a CUDA block.

To define the two remaining versions, we introduce the helper `combine` of Listing 10. It combines six consecutive parts so they become four (illustrated in Figure 8 of section 8.1):

Listing 10: Futhark function to combine the parts of two consecutive convolutions from file `mul.fut`.

```
def combine (l0:ui, h0:ui, c0:ui) (l1:ui, h1:ui, c1:ui) : (ui,ui,ui,ui) =
  let h0l1 = h0 + l1
  let h1c0c = h1 + c0 + (fromBool (h0l1 < h0))
  let c1c = c1 + (fromBool (h1c0c < h1))
  in (l0, h0l1, h1c0c, c1c)
```

For brevity, we only show `V3` since `V2` is practically identical. The convolution body – shown in Listing 11 – is a straightforward adaptation of the CUDA convolution body:

Listing 11: Futhark *convmul* `V3` convolution function on thread-basis in from file `mul.fut` (slightly edited).

```
let CONV (us: []ui) (vs: []ui) (tid: i64)
    : ( (ui, ui, ui, ui), (ui, ui, ui, ui) ) = #[unsafe]
  -- iterate over the lower two convolutions
  let k1 = tid * 2
  let k1_start = (k1 / (4*m)) * (4*m)
  let (lhc1, lhc2) : ( (ui, ui, ui), (ui, ui, ui) ) =
    loop (lhc1,lhc2) = ((0, 0, 0), (0, 0, 0)) for i<(k1 + 1 - k1_start) do
    let j = k1 - i
    let u = us[i + k1_start]
    let lhc1 = iterate lhc1 u vs[j]
    let lhc2 = iterate lhc2 u vs[j+1]
    in (lhc1, lhc2)
  let lhc2 = iterate lhc2 us[k1+1] vs[k1_start]
  -- iterate over the upper two convolutions
  let k2 = ipb*4*m - 1 - k1
  let k2_start = (k2 / (4*m)) * (4*m)
  let (lhc3, lhc4) : ( (ui, ui, ui), (ui, ui, ui) ) = ... -- as above
  -- combine to eight parts (four lower, four upper)
  in (combine lhc1 lhc2, combine lhc3 lhc4)
```

Now, preparing the convolution parts in shared memory for addition is not a trivial task. Our solution shown in Listing 12 is sub-optimal, but illustrates the process. It consists of five steps, where step 4. is the nontrivial portion. Step 3. writes the four kinds of parts to four partitions of shared memory, alike to the CUDA implementation. Unlike the CUDA implementation, the additive arrays must now be constructed with a `scatter`, requiring to compute indices corresponding to the pattern at the end of section 8.1. The indices are computed in lines 166-174, and is the reason that our implementation is sub-optimal. Firstly, we implement with $ipb \cdot m$ threads in mind, but we map over double that amount of threads. Secondly, the function that is mapped branches on the thread-ID being odd or even.[12] The optimal solution entails fusing the computation of indices with the `CONV` function, s.t. each convolution part gets tagged with a corresponding index to its place in the additive arrays.

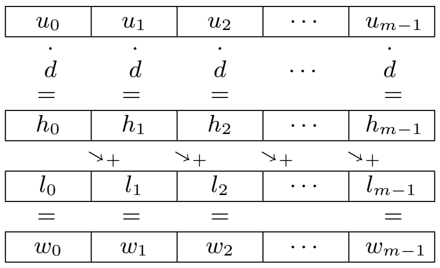
Listing 12: Futhark *convmul* `V3` adding the convolution parts from file `mul.fut` (slightly edited).

```
-- 1. copy to shared memory coalesced
...
-- 2. find the upper and lower four parts for each thread
let (lhcs1, lhcs2) = map (CONV ush vsh) (0..<ipb*m) |> unzip
let lhcs = lhcs1 ++ (reverse lhcs2)

-- 3. map the convolution result to memory
let (ls, hls, chs, ccs) = unzip4 lhcs
let lhcss = ls ++ hls ++ chs ++ ccs :> [8*ipb*m]ui

-- 4. compute indices and retrieve convolution result from memory
let (inds1, inds2) = unzip <| imap (0..<2*ipb*m)
                (\ i -> let off = i * 2
                        let isOdd = bool.i64 (i % 2)
                        let inds = if isOdd && ((off + 2) % (4*m) == 0)
                                   then (off, off+1, -1, -1)
                                   else (off, off+1, off+2, off+3)
                        let disc = (-1, -1, -1, -1)
                        in if isOdd
                           then ( inds, disc )
                           else ( disc, inds ))

let (inds11, inds12, inds13, inds14) = unzip4 inds1
let indss1 = inds11 ++ inds12 ++ inds13 ++ inds14 :> [8*ipb*m]i64

let (inds21, inds22, inds23, inds24) = unzip4 inds2
let indss2 = inds21 ++ inds22 ++ inds23 ++ inds24 :> [8*ipb*m]i64

let lhcss1 = scatter (replicate (ipb*(4*m)) 0) indss1 lhcss
let lhcss2 = scatter (replicate (ipb*(4*m)) 0) indss2 lhcss

-- 5. add the convolution parts
in baddV3 lhcss1 lhcss2
```

The Futhark implementation manages to stay close to the pseudocode of *convmul*. It is less optimized than the CUDA implementation, due to insufficient 128-bit integer support.

[12]Thirdly, the `reverse` of line 158 is unnecessary since the purpose of the array is to be scattered.

```
Input: $u$ of size $m$ base $B$ and digit $d$
Output: $w$ of size $m$ in base $B$
Use: Function ADD for adding big ints

l = map (*_low d) u
h = map (*_high d) u
h = shift 1 h
w = ADD l h
```



u_0 u_1 u_2 $\cdots$ u_{m-1}
d d d $\cdots$ d
h_0 h_1 h_2 $\cdots$ h_{m-1}
$+$
l_0 l_1 l_2 $\cdots$ l_{m-1}
w_0 w_1 w_2 $\cdots$ w_{m-1}

Figure 9: Pseudocode and illustration of parallel algorithm for multiplication by single precision factor.

Constructing the memory layout for the final addition, requires contemplating the indexing of the scheme while computing convolutions – neglected in the presented implementation.

8.4 Single Precision Factor

Integer multiplication has some special cases; we have multiplication by 0, by 1 and by the radix. Respectively, this is equivalent to 0, identity and right-shift (e.g. $4 \cdot 10^1$ in decimal is equivalent to $4 \ll 1$, and $4 \cdot 10^2$ to $4 \ll 2$, etc.). Multiplication with big integers has another special case; multiplication by a single precision factor.

This case can be computed in parallel in three steps, using parallel addition:

I Multiply each digit of the big integer by the single precision factor, resulting in two arrays – one with the low parts and one with the high parts of the multiplication.

II Shift the array with the high parts by 1.

III Add the two arrays using parallel addition (e.g. *badd*).

Figure 9 contains an illustration and the pseudocode of this algorithm.

We could integrate this special case (along with the other special cases) into the classical multiplication implementation. However, such an integration raises two concerns: First, we expect multiple precision factors to have the dominant occurrence over single precision (because we are in the domain of big integers), and therefore it is not worth the cost of the extra computational steps required to check for special cases. Second, it introduces branching into the multiplication, which is especially inefficient for batch processing with multiple instances per block (as diverging branches within a block gives unbalanced work amongst threads).

Moreover, it is essentially a parallel addition, so we do not benchmark or discuss it further.

9 Division

The last of the basic arithmetics is division, and it is far more complicated than the others. We say that the division is *exact* when the divisor precisely divides the dividend. When the division is not exact, the result is ordinarily represented by a fraction – yet we do not desire to leave the domain of integrals. Instead, we use the notion of *quotient* and *remainder* commonly present in integer semantics:

$$u \texttt{ quo } v := \lfloor u/v \rfloor \tag{21}$$

$$u \texttt{ rem } v := u - (u \texttt{ quo } v) \cdot v \tag{22}$$

Usually, the quotient is associated with the division operator, and the remainder with the modulus operator. Our division computes both values and returns them as a pair.

A common way to reason about division is to multiply the dividend with the inverse of the divisor, also called the reciprocal. GMP use reciprocals for division by single precision, but since reciprocals are expensive to compute, their basecase division use Knuth's algorithm for long division, also known as *grade-school* division [11, 14]. This algorithm is inherently sequential: It iteratively produces the result by finding one correct digit per iteration, and thus, hints at a linear depth unfit for GPGPU. As a result, we focus on algorithms using the inverse of the divisor.

A well-known algorithm to find reciprocals is the Newton-Raphson method. It takes an initial approximation, and through a series of iterative steps (also called Newton iterations), refines the precision of the approximation. While it runs a number of inherently sequential iterations too, the precision of the approximation roughly doubles at each iteration, hinting at a logarithmic depth and making it more suitable for GPGPU than long division.

However, it presents a new hurdle w.r.t. the internal representation of both the inverse and the intermediate approximations in iteration steps. One solution is to use fractions internally and convert back to integers before returning the output. The downsides to this solution is the potential loss of precision when using floating points (i.e. it will no longer be an exact arithmetic) and the overhead of converting between representations.

Instead, we use Watt's efficient and exact arithmetic algorithm for computing quotients, presented in [21]. The prerequisite of the algorithm is the existence of a shift operation for the chosen data type – which is an efficient operation for integers – and a multiplication method given as an argument. The intuition is to utilize the shift operation in order to avoid the domain change associated with computing the inverse, by instead computing what is referred to as the *whole shifted inverse*.

This section is structured as follows: In 9.1 we give a detailed introduction to the algorithm formulated by Watt in [21]. We also present a revision to the algorithm regarding an unconsidered corner case, and specialize it to the domain of big integers. In 9.2 we further discuss its adaptability w.r.t. big integers, and raise concerns and solutions based on a sequential prototype of the algorithm written in C. It serves to fully understand the complexities involved in adapting the algorithm to big integers. Lastly, in 9.3 we give an overview on how to parallelize the algorithm based on Futhark.

9.1 Algorithm

Before discussing the exact division algorithm, we introduce a new notation: The *precision* of a big integer (denoted by p) refers to the number of digits without leading zeroes. E.g. the integer $[1,2,0]$ has size $m = 3$ and precision $p = 2$.

We now give the intuition behind the algorithm defined by Watt in [21], and refer to the original paper for proofs and a generic version. The foundation of the algorithm is a whole shift operator and a whole shifted inverse operator:

Definition 8 (whole shift and whole shifted inverse of big integers)**.** We define a $n \in \mathbb{Z}$ whole shift and $n' \in \mathbb{N}$ whole shifted inverse of big integers $u \in \mathbb{N}$ and $v \in \mathbb{N}^+$ in base B as:

$$\texttt{shift}_n\ u := \lfloor u \cdot B^n \rfloor \qquad \texttt{shinv}_{n'}\ v := \lfloor B^{n'}/v \rfloor \tag{23}$$

A shift in our array-oriented representation behaves similarly to a binary system arithmetic shift, e.g. $\texttt{shift}_1\ [1,2,3] = [0,1,2]$ and $\texttt{shift}_{-1}\ [1,2,3] = [2,3,0]$. The shifted inverse behaves as an inverse that has been shifted, which follows from the its definition:

$$\texttt{shinv}_{n'}\ v = \lfloor B^{n'}/v \rfloor = \lfloor \frac{1}{v} \cdot B^{n'} \rfloor = \texttt{shift}_{n'}\ \frac{1}{v} \tag{24}$$

Now, suppose the fractional part of v consists of m' digits – then, given it is shifted by some $n' > m'$, the shifted inverse essentially behaves as a fractional representation that has been shifted into our domain. E.g. decimal system, the inverse of the number 4 is 0.25, which has a fractional part of precision 2, and so picking $n' = 2$ gives us $\texttt{shinv}_2\ 4 = \texttt{shift}_2\ 0.25 = 25$, which does not require fractions to represent.

Picking such a n' is not trivial, as is evident from recurring digits (e.g. $1/3$). However, for division, it is good enough to pick $n' = h$ where h is the precision of the dividend, proven by Watt in [21]. E.g. the decimal division 10 `quo` 3, the dividend has precision 2 and so we find the shifted inverse $\texttt{shinv}_2\ 3 = \texttt{shift}_2\ 1/3 = 33$. Then we multiply $33 \cdot 10 = 330$. Now, since the inverse was shifted by 2, we must shift it by -2 (i.e. it was multiplied with B^2, so we multiply it by B^{-2}), and get $\texttt{shift}_{-2}\ 330 = 3$.

Before we define how to compute the shifted inverse, let us consider Example 3, showing the intuition behind the above process for big integer.

Example 3 (big integer division by whole shifted inverse)**.** Suppose we have the integer $u = [1, 2, 3]$ in base $B = 2^{32}$ and wish to divide it by the integer $v = [1, 1, 0]$ in base B. Since u is of precision $p = 3$, we shift the inverse of v by 3 (using a calculator for now):

$$\texttt{shinv}_3\ [1, 1, 0] = [0, 4294967295, 0] \tag{25}$$

We can then multiply u with the shifted inverse (expanding the result size), and get:

$$[1, 2, 3] \cdot [0, 4294967295, 0] = [0, 4294967295, 4294967294, 4294967294, 2] \tag{26}$$

Now, since the fractional part of the inverse was "shifted into our domain" by 3, we shift the result by -3 (and compress it to the original size):

$$\texttt{shift}_{-3}\ [0, 4294967295, 4294967294, 4294967294, 2] = [4294967294, 2, 0] \tag{27}$$

The correct answer is $[4294967295, 2, 0]$, so the method is off by one.

In the given example, the result is not precise – which is an accommodated byproduct of using the whole shifted inverse: The method approximates the result by a difference of one at most, allowing us to mechanically adjust the result. The formal definition given by Watt in Theorem 1 of his paper [21] is as follows:

Definition 9 (quotient of big integers by whole shifted inverse)**.** For the base B big integers u and v and some $h \in \mathbb{N}$ s.t. $u \leq B^h$, the quotient of u by v is:

$$u \ \texttt{quo}\ v = \texttt{shift}_{-h}\ (u \cdot \texttt{shinv}_h\ v) + \delta, \quad \text{where } \delta \in \{0, 1\} \tag{28}$$

The challenge now lies in computing `shinv`. The remaining of this section is partitioned in two parts. Part 9.1.1 details the algorithm to compute the whole shifted inverse (`shinv`) of a big integer, and part 9.1.2 defines a division algorithm using the whole shifted inverse.

9.1.1 Computing the whole shifted inverse

The algorithm given by Watt (Algorithm 1 in [21]) computes the whole shifted inverse of an integer, by using a Newton iteration over integers and specialized to the base, s.t. it can use shifts rather than fractions. We adapt this algorithm to our big integer representation, and make a revision to it. The complete algorithm is listed in Figures 10 and 11. It computes $\texttt{shinv}_h\ v$ for v in base B, under the assumption that $B^k \leq v < B^{k+1}$. We contribute to the algorithm with the extra assumption that $k \neq 1$. We show how to find and enforce these assumptions in part 9.1.2. We now present each function step by step.

```
• Computes shinv_h v, i.e. ⌊B^h/v⌋ (Definition 8), where v is a base B big integer s.t.
    B^k ≤ v < B^(k+1) and k ≠ 1, and quo_digit and *_digit denotes a single precision operand.
fun SHINV h k v =
  -- I Handle special cases
  if v            <  B   then return B^h quo_digit v[0]
  if v            >  B^h then return 0
  if (v *_digit 2) >  B^h then return 1
  if v            == B^k then return B^(h−k)
  -- II Find a two digit initial approximation w for the shifted inverse
  V = v[k-2] + (v[k-1] * B) + (v[k] * B^2)
  w = ((B^4 - V) / V) + 1
  -- III Refine until sufficient (see below), which it may already be
  if h - k <= 2 then return shift (h - k - 2) w
  return REFINE v w h k 2

• Refines the approximation w of the h-shifted inverse of v, currently consisting of l
    correct leading digits, where the total computational work is reduced by using shorter
    iterates and divisor prefixes.  In the function, g denotes the amount of guard digits
    to account for the inexactness of using shorter iterates and divisor prefixes.  n is
    number of digits to grow the correct part of the approximation in each iteration,
    w.r.t.  shorter iterates.  s is the amount of digits to scale v with w.r.t.  divisor
    prefixes, in order to obtain a prefix that has at most g digits of error.  The Newton
    step is then, for all but the last iteration:
```

$$w = w \cdot B^l + \lfloor w \cdot (B^l - B^{-k+s-g} \cdot \lfloor v \cdot B^{-s} \rfloor \cdot w) \rfloor$$

In the last iteration, it is instead:

$$w = w \cdot B^{h-k+1-l} + \lfloor w \cdot (B^{h-k+1-l} - B^{h-2k+1-2l+s-g} \cdot \lfloor v \cdot B^{-s} \rfloor \cdot w) \rfloor$$

```
fun REFINE v w h k l =
  -- guard digits to account for the inexactness of using shorter iterates
     and divisor prefixes (at most B from being correct, so 2 guard digits)
  g = 2
  w = shift g w
  while h - k > l do
    -- n is amount to grow the correct part of the approximation in this
       iteration (w.r.t. shorter iterates). For the last iteration we get
       n = h−k+1−l, but for all others we get n = l (doubling the precision).
    n = min l (h - k + 1 - l)
    -- s is amount to scale v with (w.r.t. divisor prefixes)
    s = max 0 (k - 2*l + 1 - g)
    -- take a Newton step (see below)
    w = shift (-1) (STEP (k + l + n - s + g) (shift (-s) v) w n l g)
    -- l is amount of correct leading digits after the step (i.e. one short
       for doubling, except for the last iteration)
    l += n - 1
  return shift (-g) w

Continued in Figure 11 on next page ...
```

Figure 10: Algorithm to compute **shinv**$_h$ v, where v is a base B big integer s.t. $B^k \leq v < B^{k+1}$ and $k \neq 1$, adapted to big integers from Algorithm 1 by Watt in [21], continued on the next page in Figure 11.

... Continuation of Figure 10 from previous page.

● Takes a Newton step for big integers v and w, defined as

$$step(h, v, w, n) := w \cdot B^n + \lfloor w \cdot (B^{h-n} - v \cdot w) \cdot B^{2n-h} \rfloor$$

Here, h is the parameter to shift the inverse, and n the number of extra digits to refine in this step (since the last iteration does not double the correct precision). l is the number of leading digits currently correct, and g is the amount of guard digits added by the refinement method - both of which are passed down to POWDIFF to enable optimizations (see below). MUL is a big integer multiplication method, SUB_{abs} an absolute subtraction method, and ADD an addition method.

```
fun STEP h v w n l g =
  -- 1. Compute $B^{h-n} - v \cdot w$
  (pwd, sign) = POWDIFF v w (h-n) (l-g)
  -- 2. Compute and return $w \cdot B^n + \lfloor w \cdot (B^{h-n} - v \cdot w) \cdot B^{2n-h} \rfloor$
  if sign
  then return $\text{SUB}_{\text{abs}}$ (shift n w) (shift (2*n - h) (MUL w pwd))
  else return ADD   (shift n w) (shift (2*n - h) (MUL w pwd))
```

● Computes the following function efficiently using close products (i.e. we may sometimes truncate the length of the multiplication computing $v \cdot w$):

$$powdiff(v, w, h) := B^h - v \cdot w$$

l is number of correct leading digits in w, MUL is a big integer multiplication method, SUB is a signed big integer subtraction method (with *false* representing unsigned and *true* representing signed), and MULMOD and PREC is as defined below.

```
fun POWDIFF v w h l =
  L = (PREC v) + (PREC w) - l + 1
  -- Best case where $v \cdot w = 0$, so the result is $B^h$
  if (v == 0) || (w == 0) then return ($B^h$, false)
  -- Worst case where the multiplication $v \cdot w$ is in full size
  if L >= h                then return SUB $B^h$ (MUL v w)
  -- Close product case where only the lower L digits of $v \cdot w$ is needed
  P = MULMOD v w L
  if P      == 0 then return (P, false)
  if P[L-1] == 0 then return (P, true)
  return SUB $B^L$ P
```

● Finds the precision p of big integer v, i.e. $v < B^p$.

```
fun PREC v =
  p = (sizeof v) - 1
  while v[p] == 0 do
    p -= 1
  return p + 1
```

● Finds the first L digits of the multiplication between big integers v and w, i.e. $(v \cdot w)$ rem B^L, using some big integer multiplication method MUL.

```
fun MULMOD v w L =
  return take L (MUL v w)
```

Figure 11: Continuation of Figure 10 from last page.

`SHINV` (lines 1-13) The algorithm can roughly be divided into three steps:

I Handle the special cases (lines 4-7).

II Find an initial approximation (lines 9-10).

III Refine the initial approximation iteratively until it is sufficient (lines 12-13).

In step I we have four special cases. The first three cases guarantee that $B < v \leq B^h/2$, which is proven a prerequisite for the initial approximation method developed by Watt in [21], that picks an initial approximation guaranteed to converge fast in the refinement process. The first of these cases, is when v only consists of a single digit, and we use a division by single precision method instead. The next two cases is if $v > B^h$ or $2v > B^h$ and so the inverse is 0 and 1, respectively.

The fourth special case guarantees that $v \neq B^k$. This is important because we then know that $\lfloor B^h/v \rfloor$ is a big integer of $h - k$ digits. E.g. consider the example $h = 2$ in decimal system (so $B^2 = 100$): If v is 10 (so $k = 1$), the result is $100/10 = 10$ (2 digits). However, if $v < 10$ (i.e. $k = 0$), then $10 < 100/v$ (i.e. $h - k = 2$ digits), and likewise, if $10 < v$ (i.e. $k = 1$), then $100/v < 10$ (i.e. of $h - k = 1$ digit).

To rephrase: This case ensures that the refinement is sufficient once it has produced $h - k$ correct digits. It is handled trivially by the rule $x^a/x^b = x^{a-b}$.

Step II computes the initial approximation using the last three digits of v, e.g. for v of size m we get $V = [v[m-3], v[m-2], v[m-1]]$ in a positional notation.[13] It then finds the initial approximation as $((B^4 - V)/V) + 1$. Since V is three digits, we have that $B^4 - V$ is four digits, where the most significant digit is the maximum digit. (E.g. in decimal, V is at most 999, and so $10^4 - 999 = 9001$.) Then, a four digit number divided by a three digit number, where the fourth digit of the dividend is the maximum digit, and the result is then incremented by one, gives us 2 digit number. (E.g. consider again $(9001/999) + 1 = 9 + 1 = 10$). As proven by Watt in [21], this two digit approximation is a good initial approximation, guaranteed to converge fast in the refinement method.

Step III checks whether the initial approximation is sufficient, i.e. if the shifted inverse (of $h - k$ digits) is a one or two digits result. If more correct digits is needed (i.e. $h - k > 2$), it starts the refinement process using the specialized Newton iteration.

`REFIINE` (lines 16-29) Watt presents three refinement methods with a varying degree of optimizations. The method we use is the optimal (`refine3` of Algorithm 1 in [21]), with no alterations to its original formulation. This refinement method expands the specialized Newton iteration with two concepts; *shorter iterates* and *divisor prefixes*. Shorter iterates is

[13] Given our assumption $k \neq 1$, and the first special case in the previous step, where $k = 0$, it is guaranteed that v is of sufficient size to compute such a V (i.e. v is at this point at least three digits).

the idea that not all digits matter in the intermediate results between iterations: Since the precision roughly doubles, only the leading l digits of the intermediate results contributes to the refinement at each iteration, where l denotes the current number of correct leading digits. It effectively allows us to truncate the size of the big integer multiplication in each Newton step (the `STEP` function discussed below), rather than doing each step in the full size of the dividend. The size of the multiplication roughly doubles with the iteration number. In turn, this introduces the need for *guard digits* (line 18), because the truncated-sized steps may differ from the full-sized steps, but only in the two least significant digits.

However, this is where our contributed assumption $k \neq 1$ is important. Shorter iterates have the side effect of each iteration being exactly one digit short from doubling the precision at each iteration. This means that the number of correct leading digits, l, must be ≥ 2. Thus, if $k = 1$ and we are in the case $h - k > 2$, then we start refining on $l = 1$. In turn, one digit short of doubling $l = 1$ is $l = 1$, and hence, the refinement process never increases the precision and runs forever.

Lastly, divisor prefixes is the idea that, when the divisor v is large in relation to the shifted inverse approximated at each iteration, some of the leading digits of v does not contribute to the result of the iteration step. Instead, we use a prefix of v to further shorten the size of the multiplications in each step. This process is imprecise too, but is defined s.t. the 2 guard digits accounts for the imprecision. The factor to shift v in order to obtain a prefix, is determined at line 24.

`STEP` (lines 35-41) This function is where most of the computational work lies, since it contains two multiplications. The function computes $w \cdot B^n + \lfloor w \cdot (B^{h-n} - v \cdot w) \cdot B^{2n-h} \rfloor$ for a given h, n, and big integers v and w in base B, as given by Watt in [21]. It computes straightforward, except it uses the function `POWDIFF` to handle $B^{h-n} - v \cdot w$ more efficiently, as given by Watt. We have made no alterations to the original function formulation, except for explicitly introducing a sign-magnitude representation of the term of $B^{h-n} - v \cdot w$.

`POWDIFF` (lines 44-54) Lastly is the function `POWDIFF`. The term $B^h - v \cdot w$ appears in each iteration step, and while it can be computed straightforward, Watt increases the efficiency by using close products [21]. The idea is that, if the product $v \cdot w$ is close to B^h by a factor of $|B^h - v \cdot w| \leq B^L, L < h$, then only the lower L digits of the product $v \cdot w$ are needed, since the remaining are predetermined. Hence, we can use a truncated multiplication of size L rather than the size of v and w. This case is handled in lines 51-54. Formally, the multiplication for integers v and w corresponds to $(v \cdot w)$ `rem` B^L, and is computed by the function `MULMOD` in lines 64-65. In order to find such a L, `POWDIFF` use the precision of its inputs (line 45), which is computed straightforward by the function `PREC` in lines 57-61.

9.1.2 Computing the big integer division

We now present an algorithm for finding quotient and remainder of big integers using Definition 9 and `SHINV` of Figure 10 – including how to enforce the assumption $k \neq 1$ of `SHINV` using shifts (the necessity is discussed in part 9.1.1). The Pseudocode for this algorithm is in Figure 12 and contains four steps:

1. Lines 3-4 compute the assumptions of the precision of the inputs straightforward using the functions `FINDK` (line 19-23) and `FINDH` (line 26-30).
2. Lines 6-9 handle the case $k = 1$ by shifting both u and v by one. A shift is equivalent to a multiplication by definition, so the quotient is unaffected. Furthermore, the relative difference of h and k is unchanged, so no further iterations are introduced.
3. Lines 11-12 find the quotient and remainder using Equation (28) with $\delta = 0$, some big integer multiplication method `MUL`, and some absolute subtraction method $\texttt{SUB}_{\texttt{abs}}$.
4. Lines 14-16 check the value of $\delta = \{0, 1\}$ based on the nonsensicality of the remainder, and adjust the quotient and remainder accordingly.

Thus, the function `DIV` defined in Figure 12 gives us an exact division operator for big integers. It relies on the function `SHINV` of Figures 10 and 11 (sub-functions thereof), and on arithmetic operators for addition, subtraction, and multiplication of big integers (such as *badd*, *bsub*, and *convmul*), which, it can be parameterized over. Hence, it is possible to e.g. use different operators for different input sizes.

The division is asymptotically dominated by the choice of multiplication operator (given that the addition and subtraction is efficiently implemented). The iterative refinement process takes $\lceil \log(h-k) \rceil$ iterations, where h depends on the precision of the dividend and k on the precision of the divisor. This number comes from the fact that the refinement method produces a shifted inverse of $h - k$ digits (as discussed in 9.1.1), and it roughly doubles the number of correct digits in the approximation at each iteration.

Each iteration step runs at most two multiplications (one in `STEP` and either one or zero in `POWDIFF`). While this gives us at most $2 \cdot \lceil \log(h-k) \rceil$ multiplications, the size of the inputs to the multiplications is doubling at each iteration, due to the shorter iterates method. From the analysis by Watt in [21], the total amount of digits multiplied from this method, gives us $O(M(m))$ work, where $M(m)$ is the work of the chosen multiplication operator and m is the size of the input integers for the division.

```
fun DIV u v =
  -- 1. Find a $h$ and $k$ that satisfies $u \leq B^h$ and $B^k \leq v < B^{k+1}$
  h = FINDH u
  k = FINDK v
  -- 2. Enforce assumption $k \neq 1$ of SHINV by shifting
  if k == 1 then u = shift 1 u
                 v = shift 1 v
                 h += 1
                 k += 1
  -- 3. Find quotient and remainder approximation according to Definition 9
  q = shift (-h) (MUL u (SHINV h k v))
  r = $\text{SUB}_{\text{abs}}$ u (MUL q v)
  -- 4. Adjust the approximation for $\delta$
  if (r >= v) then q = ADD q 1
                   r = $\text{SUB}_{\text{abs}}$ r v
  return (q, r)

-- finds $k$ s.t. $\text{B}^{\text{k}} \leq \text{v} < \text{B}^{\text{k+1}}$
fun FINDK v =
  k = (sizeof v) - 1
  while v[k] == 0 do
    k -= 1
  return k

-- finds $h$ s.t. $u \leq B^h$ (by picking the smallest $h$)
fun FINDH u =
  h = FINDK u
  if u[h] == 1
  then return u
  else return u + 1
```

Figure 12: Pseudocode for a big integer division algorithm, computing u quo v and u rem v using Definition 9 and function SHINV of Figure 10, where MUL is a big integer multiplication method, SUB_{abs} a absolute subtraction method, and ADD an addition method.

9.2 General Reflections and Prototyping

This section discusses some concerns and solutions about the presented algorithm. We wrote a sequential low-level prototype of the algorithm in C to better understand its adaptation of our big integer representation. In its creation, we discovered multiple interesting subject matters – some which are directly included in the pseudocode given in section 9.1, and some which we now present.

Initial Approximation The initial approximation is found in step II of Listing 10. It gathers the three leading digits of v in V and then finds $B^4 - V$, giving a result of four digits. It then divides those four digits by V, giving a result of either one or two digits. Hence, the quadruple base type `qint_t` is necessary to efficiently compute the initial approximation with register arithmetic. This restricts a CUDA implementation to base `uint32_t` and Futhark to `u16`. In C, this process corresponds to Listing 13:

Listing 13: C code to efficiently form the initial approximation of `SHINV` in base `uint_t` of `bits` bits and quad type `quint_t` of 4·`bits` bits.

```
quint_t V = (quint_t) v[k-2];                          // V  = v[k-2]
V +=       ((quint_t) v[k-1]) << bits;                 // V += v[k-1] · B^1
V +=       ((quint_t) v[k])   << (2*bits);             // V += v[k]   · B^2

quint_t r = (((quint_t) 0) - V) / V + ((quint_t) 1); // r = (B^4 - V) / V + 1

w[0] = (uint_t) r;                                     // w = r
w[1] = (uint_t) (r >> bits);                           // (assuming w was 0)
```

Base Powers These are prominent throughout the algorithm, and comes in a multitude of usecases. When multiplied, they take the shape of a `shift` operator, as already incorporated into the pseudocode. In the `POWDIFF` function, they are returned or otherwise used directly, and hence, require construction in a new memory space.

Remaining is the special cases of step I in the `SHINV` function. The base powers are operands to comparisons that express something structural about the divisor v. Hence, they can be integrated in said comparisons, avoiding their construction. E.g. consider the second case $v > B^h$. This optimizes to the check: $\exists i \in \mathbb{N}.\ (h < i < m \land v[i] \neq 0) \lor (h = i < m \land v[i] > 1)$.

(On a similar note, the multiplication by single precision of the third special case $2v > B^h$ should be treated as $v > B^h/2$ and integrated in the comparison as well.)

Preallocating Memory Throughout this thesis, the big integer arithmetic has been considered to preserve the input shape on output. The efficiency of this algorithm requires fine control over the size of operands at each iteration, and hence, we need to adjust the size parameter of inputs and outputs, according to the iteration number. Furthermore, it is ideal to preallocate a memory region large enough to hold the biggest intermediate value of the refinement, rather than periodically doubling the memory regions of the operands. From the analysis in Watt's paper [21], the size of the operands at iteration i is at most 2^i. Since we have $\lceil \log(h-k) \rceil$ iterations, we get the maximum operand size $2^{\lceil \log(h-k) \rceil}$.

Guard Digits In order to get our prototype to validate, we adjusted the amount of guard digits in the refinement method from 2 to m, with m being the size of the inputs. Currently, we are not aware whether this is related to the algorithm or an error in the prototype.

9.3 Parallelization and Futhark Implementation

While section 9.2 focused on general reflections, this section focuses on concerns and solutions regarding parallelization and Futhark implementation. Our implementation is not efficient and only partially validates, but it illustrates some of the complexities involved in a parallel adaptation of the algorithm – which we now present.

Multiple Instances per Block and Load Balancing In our other arithmetic operators, we incorporate multiple big integer instances per CUDA block, which are then handled by segmented operations. It allows to maintain the efficiency of the arithmetics when processing small integers. However, Watt's algorithm for computing `SHINV` (Figure 10) contains branching invariant to the precision of the inputs. The most apparent branching is the special cases of `SHINV`, directly returning from the function. Then we have the close product optimization in `POWDIFF`, which will either do no multiplication, a full size multiplication, or a multiplication with truncated inputs. Lastly, the number of sequential Newton iterations in the refinement process directly depends on the precision of the inputs. Hence, the division algorithm is not suitable to process multiple instances per block.

Furthermore, due to the shorter iterates, the number of digits being multiplied at each refinement step roughly doubles. In turn, mapping this division at block-level must be carefully considered. The simplest solution is to run all operations in full-sized inputs – however, as Watt analyze for the method he calls `refine1` in [21] (a method without shorter iterates), this solution increases the work by a factor logarithmic in the precision of its inputs. Another solution is to spawn a kernel with the exact number of threads needed to efficiently run the arithmetic of greatest input length. However, this results in threads idling for arithmetics of shorter length. Likewise, if the kernel spawns with an inadequate number of threads, the sequentialization factor of the arithmetic of the greatest length input may exhaust registers and local memory, overall decreasing efficiency.[14]

[14]The current Futhark implementation (and C prototype) runs the arithmetics at padded full-size length.

Shift Operator Shifting a big integer in parallel is very efficient and equivalent to a map, as seen on the Futhark code in Listing 14. We can transform the operator to be in-place in CUDA by splitting the map: Fetch the designated digit → synchronize → write the digit.

Listing 14: Shift operator (Definition 8) in Futhark from file `helper.fut`.

```
def shift [m] (n: i64) (u: [m]ui) : [m]ui =
  map (\ i -> let off = i - n -- positive for right, negative for left
              in if off >= 0 && off < m then u[off] else 0) (iota m)
```

Division by Single Precision The first special case of `SHINV` where $v < B$ requires quotient by a single precision integer (digit). The intuition follows from Definition 1 and states that the quotient of a size m integer u by digit d in base B can be computed as:

$$q_i = \frac{\sum_{j=i}^{m-1} u_j \cdot B^{j-i}}{d} \texttt{ rem } B \tag{29}$$

E.g. consider the decimal number 300 divided by 4: We have $q_0 = (300/4) \texttt{ rem } 10 = 5$, $q_1 = (30/4) \texttt{ rem } 10 = 7$ and $q_2 = (3/4) \texttt{ rem } 10 = 0$, giving the number 75. This approach includes the big integer u and does not fit in a word. However, Equation 29 can be rephrased in terms of the standard long division (grade-school) method. The method formalizes to computing the partial quotient $q_{i\in\{0,..,m-1\}}$ using the partial remainder $r_{i\in\{0,..,m\}}$ as follows:

$$q_i = (r_{i+1} \cdot B + u_i) \texttt{ quo } d \tag{30}$$

$$r_i = \begin{cases} (r_{i+1} \cdot B + u_i) \texttt{ rem } d & \text{if } i < m \\ 0 & \text{otherwise} \end{cases} \tag{31}$$

The intermediate results of this approach fits in a double word. This algorithm is a simple solution (opposed to e.g. GMP's) and the one occupying our sequential prototype. However, it is inherently right-to-left sequential and unsuitable for parallel computation.

We recognise three approaches:

1. Parallelize the long division algorithm (possibly over a scan similarly to addition).
2. Computing an inverse (e.g. a whole shifted inverse by a method similar to `SHINV`), but this approach requires a multiplication afterwards.
3. Shifting both u and v by two when $k = 0$ (similarly to the shift by one when $k = 1$).

The potentialities of approach 1 and 2 are unknown to us and beyond the scope of this thesis. Hence, we use approach 3 for our parallel Futhark implementation.

Assumptions, Comparisons, and Signed Subtraction The algorithm has various comparisons of big integers, all of which parallelizes to map-reduce compositions. First, consider the special cases of `SHINV`. As mentioned in section 9.2, they can be optimized to a unary comparison over the big integer. Following the example of that section, the second special case $v > B^h$ parallelizes to Listing 15:

Listing 15: Futhark function to check $u > B^i$ in parallel for big integer u in base B from file `div.fut`.

```
def gtBpow [m] (u: [m]ui) (i: i64) : bool =
  map2 (\ x j -> (x > 1 && j == i) || (x > 0 && j > i) ) u (iota m)
  |> reduce (||) false
```

Next, we have the equalities involved with the `POWDIFF` function. Equality of big integers corresponds to equality between digits, which parallelizes straightforward. Then there are the structural assumptions of big integers, written sequentially as the helpers `FINDK`, `FINDH`, and `PREC` of Figures 11 and 12. They parallelize to a map-reduce composition that finds the greatest nonzero index. E.g. Listing 16 shows a parallel `FINDK`, where the map-reduce is equivalent to the *max* function over the indices of nonzero digits in the input:

Listing 16: Futhark function to find i for big integer u in base B s.t. $B^i \leq u < B^{i+1}$ from file `div.fut`.

```
def findk [m] (u: [m]ui) : i64 =
  map2 (\ x i -> if x != 0 then i else 0 ) u (iota m)
  |> reduce (\ acc i -> if i != 0 then i else acc ) 0
```

Penultimately, the subtraction presented in section 7.4 is unsigned, yet `POWDIFF` requires signed subtraction. Extending the subtraction to signs is trivial, achieved by determining the relation between the operands. For this purpose, we use the general big integer '<' comparison of Listing 17. The map in line 84 encodes whether the pairwise digits of the inputs are either '<' or '='. Line 85 then reduce with the following operator:

$$(l1,\ e1) \odot (l2,\ e2) \coloneqq (l2 \vee (e2 \wedge l1),\ e1 \wedge e2) \tag{32}$$

The rationale is: If the first operand is less than the second operand in the most significant digits, then the whole integer is less than – if it is equal in the most significant digits, then it is decided in the least significant digits. The left-associative neutral element of $\odot$ is (`False`, `True`). Proof of associativity and neutral element are included in Appendix C.

Listing 17: Futhark function to check $u < v$ in parallel for big integers u and v from file `helper.fut`.

```
83 def lt [m] (u: [m]ui) (v: [m]ui) : bool =
84   map2 (\ x y -> (x < y, x == y) ) u v
85   |> reduce (\(l1,e1) (l2,e2) -> (l2 || (e2 && l1), e1 && e2)) (false,true)
86   |> fst -- ‘fst‘ extracts the first tuple field
```

Lastly, we have the '≥' associated with finding δ (line 14 of Listing 12). This can be done efficiently by negating the result of the '<' operator of Listing 17 above.

10 Correctness

Our big integer arithmetic operators are defined to be exact and efficient. The former property collapses under incorrectness and more, the latter loose its meaning. Performance benchmarks of invalid implementations are not trustworthy, as the underlying algorithms may not be fully represented, and therefore, correctness is a prerequisite of measuring the efficiency of our implementations. Additionally, we have build a tower of arithmetics, and so the properties of the whole tower collapses under an invalid operator.

While proofs are given by the authors of the algorithms, this section focuses on testing the correctness of the implementations. It is not plausible to exhaustively test the correctness on all big integers, and we might leave out important input patterns by writing manual test cases. Hence, we test on randomly generated big integers and compare the result of running our implementations against GMP's, assuming that GMP gives the valid answer.

Testing Methodology Testing the results of our CUDA implementations against GMP's is straightforward, because GMP can be linked to C++ and called directly in from our host function. We generate some random big integers on which we call both ours and GMP's arithmetic functions on and compare the results. To generate a random big integer u of size m and nz non-zero digits, we use the code in Listing 18:

Listing 18: Random big integer generator in C++ with u of size m and nz non-zero digits.

```
for(int i = 0; i < m; i++) {
    uint32_t x = 0;
    if(i < nz) {
        uint32_t low  = rand()*2;
        uint32_t high = rand()*2;
        x = (high << 16) + low;
    }
    u[i] = x;
}
```

Regarding Futhark, the compiler has a builtin test functionality called `futhark-test` [3]. It allows us to write test-specifications in Futhark programs that the compiler can generate and run. A specification consist of a set of inputs and their the expected outputs. The inputs can be either randomly generated or fixed, and the outputs can be either fixed or automatically determined (by assuming C as backend gives the correct result).

It is not possible to directly test our Futhark implementations against GMP's through `futhark-test`, but it is possible indirectly by constructing the tests in a peculiar way: We write a small Futhark library, called `gmp-validation-lib.fut`, which exports exactly one of each arithmetic function. Each function takes exactly two inputs arrays and produce exactly one output array, while being agnostic to the shape of the input. I.e. for the underlying arithmetics that requires the input to be a specific shape (read; *convmul*), the library function adds padding to ensure that the shapes matches. Now, we compile this library to a C API using the Futhark library feature [3].

This produces a C header file that exports the functions and data types of the compiled library. In turn, we write a C-program that incorporates GMP as described above, and tests the Futhark arithmetic functions from the C-program through the API.

Hence, we obtain a GMP validated *black box* Futhark library. We can therefore substitute GMP function calls in Futhark test programs with function calls to this library, effectively giving us access to GMP inside the `futhark-test` environment.

The test specifications assign randomly generated 2D arrays of different sizes as inputs and then expects the output to be `true`. The test function that compares our addition functions against GMP is shown in Listing 19 below – an identical test function is defined for multiplication. Here, `test_add` refers to the GMP-validated addition and `oneAddVX` to version `X` of our addition implementations. Line 24 defines a validation function, lines 25-30 run our addition implementations, and line 31 checks that all results validate.

Listing 19: Futhark testing-functions for addition with 64-bit base from `fut-validation.fut`.

```
22    let (N, M) = (n/4, m/4)
23    let validP = (\ws -> map2 eq (map2 test_add us vs) ws |>reduce (&&) true)
24    let ws0 = oneAddV0 us vs
25    let ws1 = oneAddV1 us vs
26    let ws2 = oneAddV2 M (us :> [n][4*M]ui) (vs :> [n][4*M]ui) :>[n][m]ui
27    let usV3 = unflatten (us :> [N*4][4*M]ui) :> [N][4][4*M]ui
28    let vsV3 = unflatten (vs :> [N*4][4*M]ui) :> [N][4][4*M]ui
29    let ws3 = (oneAddV3 M usV3 vsV3 |> flatten) :> [n][m]ui
30    in validP ws0 && validP ws1 && validP ws2 && validP ws3
```

Similarly to the definition of validation tests, we also define property-based tests over the usual arithmetic properties, such as commutativity for addition. They are included for the purpose of completeness (and a potential source for debugging information), but are redundant w.r.t. the GMP validation, assuming GMP gives the correct result.

(A final note on validating division in C; the divisor has a randomly generated nz value to avoid hitting the same special cases repeatedly, providing more meaningful tests.)

Test Results We run the Futhark tests using `futhark-test` on base `u16`, `u32`, and `u64` integers in the size range $4 \leq m \leq 512$ and $num\_instances = 2048$, with $ipb = 4$ where relevant. We run all tests twice with different backends, once on the CPU with C as backend, and once on the GPU with OpenCL as backend. The tests includes properties and validation against GMP, and they run on all four versions of *badd* (addition) and all three versions of *convmul* (multiplication). All tests validates.

Next is the Futhark-GMP validation in C. We test addition, subtraction, multiplication, and division. We run the tests of base `uint16_t`, `uint32_t`, and `uint64_t`, with division only ran on `uint16_t`. They run on two different integer size patterns: 350 tests with $m = \{2, 4, 8, 16, 32, 64, 128\}$, 50 tests for each size, and 2500 tests with 50 randomly generated sizes in the range $1 \leq m \leq 512$, 50 tests for each size. The tests are run twice using the Futhark functions compiled to C and OpenCL, respectively. Tests for addition, subtraction, and multiplication validates – tests for division does not validate.

Lastly, we have the CUDA-GMP validations tests. We test all three versions of *badd* and all five versions of *convmul* with base `uint32_t` and `uint64_t`. The test setup is fused with the performance benchmark setup (i.e. after a benchmark has been run, the result gets valid), so further details about the test-configuration are in section 11.1. All tests validate.

11 Performance

To assess the efficiency of our implementations, we measure their relative performance of running a benchmark suite. We expect that the CUDA implementations yields the best performance given the underlying algorithms and optimizations, and in turn, reveals the overhead and inefficiencies of the Futhark implementations. We also run the benchmarks using the CGBN library described in section 2. It enlights the relative performance of our implementations against state of the art, in addition to the strong and weak points of the underlying algorithms and overarching implementations strategy (e.g. pros and cons of block-level versus warp-level processing of big integers).

The runtimes of benchmarks can be tedious to directly compare. Instead, we define some performance metrics based on the runtime, total number of bits, and arithmetic operation, allowing us to directly compare results. We keep the total number of bits fixed, but vary the size of integers and number of integer instances across runs.

The structure is as follows: Section 11.1 describes the setup of our benchmarks, section 11.2 introduces the performance metrics, and section 11.3 presents benchmark results.

11.1 Benchmark Setup

We have two benchmark types: The first type runs the arithmetic operators straightforward, serving as the basis for evaluating performance. The second type runs multiple consecutive applications of the operators, providing insight on how well the arithmetics scales. The consecutive calls in CUDA are collected in a single kernel to reduce memory overhead by keeping intermediate results in shared memory. In Futhark we have consecutive intra-block calls, benchmarking the compilers ability to fuse the arithmetics compared to CUDA. The consecutive calls of the second type corresponds to computing $10(u + v)$ and $(u \cdot v)^5$. Therefore, it is ten additions and six multiplications.

While we want to vary the size and number of big integers, the benchmark results are only directly comparable if we keep the total amount of work fixed. E.g. consider a fixed work size of 256 bits; the benchmarks can be run with $n = 1$ instance of base 32-bit big integers of size $m = 8$, or, run with $n = 1$, $m = 4$, and base 64-bit, or, run with $n = 2$, $m = 2$, and base 64-bit, etc. – which all uses the same number of bits in total. We choose the fixed total amount of work to be 2^{32}-bits, i.e. 2^{27} `u32`-words or 2^{26} `u64`-words. We measure across the sizes of $\{2^9, 2^{10}, \ldots, 2^{18}\}$-bits big integers, and adjust the number of instances accordingly.

Runtimes are averaged over multiple runs to improve stability and accuracy. In CUDA we average across 300 runs for addition and 100 runs for multiplication, with one dry-run before starting the timer. We transfer memory to the device beforehand and stop the timer as soon as the kernel has completed its runs. In Futhark we use the `futhark-bench` utility, that allows us to write benchmark-specifications, similarly to `futhark-test` [3]. This utility also handles memory and do a dry-run before starting the timer. It runs until the confidence level has reached 95% (with at least 10 runs) and reports both the average runtimes and the confidence interval.

11.2 Performance Metrics

Metrics are defined on an operator basis and derived from the complexities, inner workings, and benchmark setup of the operators. Runtimes are measured in microseconds.

Addition In section 7.1 we analyze the complexity of addition to work $O(m)$ and depth $O(\log m)$. It is computationally efficient, and only require communication w.r.t. the prefix sum. Hence, we expect addition to be *bandwidth* bound rather than *operation* bound.

For n instances of addition of big integers with size m, we calculate the bandwidth (measured in GB/s) using the following formula:

$$bandwidth = 3 \cdot \frac{n \cdot m \cdot (\texttt{bits}/8)}{1000000000} \cdot \frac{1000000}{runtime} \tag{33}$$

The first fraction represents how many gigabytes are in an array of big integers, where the term $\texttt{bits}/8$ is the number of bytes in a word, $n \cdot m$ is the total amount of words, and the denominator converts from bytes to gigabytes. The second fraction is the runtime converted to seconds. The constant 3 is because addition must access global memory thrice – twice for reading the inputs and once for writing the output – regardless of performing one or ten consecutive additions, since intermediate results can remain in shared memory. Thus, we use the same formula for both benchmark setups.

Multiplication In section 8.1 we analyze the complexity of classical multiplication to work $O(m^2)$ and depth $O(m)$. It is a computationally demanding operation, and hence, we expect multiplication to be *operation* bound.

While we could center the metric around the number of `uint_t` operations, it leads to circuitous comparisons of benchmarks results across bases. Instead, we normalize the results by choosing the number of `uint32_t` operations as the operational core for our metric. We then define the metric as the number of gigaoperations per second (*Gu32ops*).

The metric us not as clear-cut as the bandwidth, but we can estimate it. For classical multiplication, the number of operations can be estimated by the squared number of 32-bit words times the number of instances:

$$n \cdot (m \cdot (\texttt{bits}/32))^2 \tag{34}$$

However, classical multiplication is not asymptotically optimal. Instead, we base our metric upon FFT multiplication, on the ground that it should reflect the optimal scenario. We use the one proposed by Oancea and Watt in [19]:

$$\mathit{Gu32ops} = \frac{n \cdot 300 \cdot (m \cdot (\texttt{bits}/32)) \cdot \log(m \cdot (\texttt{bits}/32))}{1000000000} \cdot \frac{1000000}{\mathit{runtime}} \tag{35}$$

When we compute six consecutive multiplications, we execute six times the number of operations, and thus, we multiple the metric by 6 in these benchmarks.

11.3 Benchmark Results

We report and discuss the benchmark results for addition and multiplication in sections 11.3.1 and 11.3.2. Section 11.3.3 summarize the discussions. Regarding the structure of the benchmark results tables: Legends starting with *C-* refers to CUDA implementations, and *F-* refers to Futhark implementations. The legend *CGBN* shows the result of running the benchmark with CGBN, and *Bits* and *Instances* contains the batch and integer size for the benchmarks. Entries denoted as ‘ – ‘ refuse to run (e.g. exceeds block-level). The benchmarks are run on a NVIDIA GTX 1650 SUPER – a GPU with 1280 CUDA cores, 4GB memory, 192.0 GB/s memory bandwidth, and 4.416 TFLOPS compute power [4].

11.3.1 Addition Results

Tables 1 and 2 shows benchmarks for one addition in 64-bit and 32-bit base, respectively, while Tables 3 and 4 shows for ten additions. Here is what we can gather from the tables:

- **Implementation versions** agrees that `V3` has best performance (which is also the most optimized version). The only outliers are that for 10 additions, the segmented scan has a slight cost at around 2^{15}-bits, with biggest difference in bandwidth being 73% and 65% peak bandwidth in Table 3 for 2^{16}-bit integers of base `u64`. We only consider `V3` in the following observations.

Bits	Instances	CGBN	C-V1	C-V2	C-V3	F-V0	F-V1	F-V2	F-V3
2^{18}	2^{14}	62	–	147	161	–	–	–	–
2^{17}	2^{15}	67	–	161	163	–	–	–	–
2^{16}	2^{16}	19	168	166	166	116	155	158	146
2^{15}	2^{17}	19	168	167	166	146	168	168	168
2^{14}	2^{18}	84	168	168	166	150	168	168	168
2^{13}	2^{19}	164	168	168	165	158	168	168	168
2^{12}	2^{20}	165	168	168	166	157	168	142	168
2^{11}	2^{21}	164	169	162	166	134	165	75	168
2^{10}	2^{22}	156	107	92	166	72	91	38	168
2^{9}	2^{23}	118	55	47	167	36	47	20	168

Table 1: Performance of one addition in base `u64` measured in GB/s (higher is better, 192 is peak).

- **Choice of base for one addition** in CUDA is inconsequential until the inputs are of 2^{18}-bits. This is where base `u32` results in sequentialization factor $q = 8$, while `u64` remains at $q = 4$. In Futhark, the difference is more noticeable, where `u64` is around 88% of the peak bandwidth, and `u32` around 82%. As in the CUDA case, `u64` is more suitable for higher bit count.
- **Choice of base for ten addition** in CUDA is impactful, where `u32` runs at roughly 40% peak bandwidth, and `u64` at roughly 64%. For Futhark, the difference is still there, but more subtle at 13% and 15%.
- **Choice of language (Futhark or CUDA)** is inconsequential for one addition of base `u64`, but CUDA has a slight advantage in base `u32` and are able to maintain 88% peak bandwidth, whereas Futhark drops to 82% peak bandwidth. However, the difference in the two languages are much more noticeable for ten additions. The biggest difference is that of Table 3, where CUDA runs at 65% peak bandwidth and Futhark at 15%.
- **Compared to CGBN** we see that our implementations run faster for all benchmark sizes when computing one addition, except for 2^{11} to 2^{13} bits. In this interval, CGBN peaks and all implementations utilize roughly the same amount of bandwidth. However, when moving to ten additions, CGBN is able to maintain its bandwidth utilization, whereas our implementations (especially Futhark) shows a big performance drop. As discussed in section 2, this is due to CGBN working at warp-level rather than block-level, reducing the latency overhead of running consecutive arithmetics. However, our CUDA addition is still faster than CGBN for sizes 2^{14}-bits and up. Another noticeable different is the consistency of bandwidth utilization in our best implementations compared to CGBN.

Bits	Instances	CGBN	C-V1	C-V2	C-V3	F-V0	F-V1	F-V2	F-V3
2^{18}	2^{14}	62	–	104	100	–	–	–	–
2^{17}	2^{15}	67	–	168	168	–	–	–	–
2^{16}	2^{16}	19	–	168	168	–	–	124	115
2^{15}	2^{17}	19	136	168	168	61	83	165	153
2^{14}	2^{18}	84	163	168	168	76	112	169	157
2^{13}	2^{19}	164	169	168	168	79	115	169	158
2^{12}	2^{20}	165	169	168	168	84	122	145	157
2^{11}	2^{21}	164	155	159	168	83	121	77	157
2^{10}	2^{22}	156	105	92	168	72	91	39	158
2^{9}	2^{23}	118	54	47	168	37	47	20	158

Table 2: Performance of one addition in base `u32` measured in GB/s (higher is better, 192 is peak).

Bits	Instances	CGBN	C-V1	C-V2	C-V3	F-V0	F-V1	F-V2	F-V3
2^{18}	2^{14}	25	–	108	92	–	–	–	–
2^{17}	2^{15}	60	–	120	109	–	–	–	–
2^{16}	2^{16}	73	40	142	124	–	–	27	24
2^{15}	2^{17}	45	47	137	124	15	24	34	29
2^{14}	2^{18}	97	50	113	124	19	27	32	29
2^{13}	2^{19}	162	45	82	123	19	31	30	29
2^{12}	2^{20}	164	35	44	123	17	29	20	29
2^{11}	2^{21}	161	24	23	124	15	25	11	29
2^{10}	2^{22}	152	12	12	124	8	13	5	29
2^{9}	2^{23}	113	6	6	124	4	7	3	29

Table 3: Performance of ten additions in base `u64` measured in GB/s (higher is better, 192 is peak).

Bits	Instances	CGBN	C-V1	C-V2	C-V3	F-V0	F-V1	F-V2	F-V3
2^{18}	2^{14}	25	–	75	76	–	–	–	–
2^{17}	2^{15}	60	–	66	55	–	–	–	–
2^{16}	2^{16}	73	–	81	68	–	–	23	21
2^{15}	2^{17}	45	21	88	77	9	14	25	23
2^{14}	2^{18}	97	24	79	77	9	17	26	24
2^{13}	2^{19}	162	25	62	77	10	18	23	24
2^{12}	2^{20}	164	23	44	77	10	17	20	24
2^{11}	2^{21}	161	17	23	77	9	16	11	24
2^{10}	2^{22}	152	12	11	77	8	13	6	24
2^{9}	2^{23}	113	6	6	77	4	7	3	24

Table 4: Performance of ten additions in base `u32` measured in GB/s (higher is better, 192 is peak).

11.3.2 Multiplication Results

Tables 5 and 6 shows the results for one multiplication in 64-bit and 32-bit base, respectively, while Tables 7 and 8 shows for six multiplications. We can gather from the tables:

- **Implementation versions** in CUDA agree that `V5` is best, both for one and six multiplications, as expected. The importance of processing multiple instances per block for sizes 2^9 to 2^{11} bits is very noticeable on the two versions supporting this (`V3` and `V5`). W.r.t. Futhark, the inefficiencies of `V2` and `V3` described in section 8.3 has a huge impact on the performance. E.g. in Table 6 with 2^{12}-bit integers, `V1` gives roughly 4× more Gu32ops than `V3`. However, we still see the pattern that handling multiple instances per block is important for efficiency at sizes 2^9 to 2^{11} bits.
- **Choice of base for one multiplication** strongly favors running in 64-bit base. The average increase in Gu32ops of using `u64` compared to `u32` is in CUDA `V5` 1.4× and in Futhark `V1` 1.9×. This is expected, since halving the base size will double the number of digits, and the classical multiplication is quadratic in the number of digits.
- **Choice of base for six multiplications** also favors `u64`, but with a smaller factor of Gu32ops than for one multiplication, e.g. comparing base `u64` to `u32` for CUDA `V5` is 1.2× and for Futhark `V1` it is 1.7× on average.
- **Choice of language (Futhark or CUDA)** has a big performance impact. Futhark does not have 128-bit integer support, which is a suspect for the gap between the two languages. Even if we compare `V1` in CUDA and Futhark (i.e. with no extra optimizations added to *convmul*), we still see and average increase of Gu32ops for CUDA compared to Futhark of: 1.6× in Table 5, 2.2× in Table 6, 1.7× in Table 7, and 2.7× in Table 8. However, the implementations of both languages seems to roughly follow the same pattern w.r.t. the number of bits and instances, which could indicate that if the gap can be closed (e.g. if Futhark gets 128-bit integers), the Futhark implementation will become competitive as well.
- **Compared to CGBN**, both our best CUDA and Futhark implementation is much faster for one multiplication. However, for six multiplications, CGBN is more efficient than ours up to 2^{16} bit integers. For smaller integers (i.e. of 2^9 to 2^{11} bits), CGBN is *much* faster than our implementations when running six multiplications – while for 2^{14} and 2^{15} bits integers, the performance gap between our implementations and CGBN is less significant.

11.3.3 Summary of Results

Overall, the implementations we have produced are generally competitive in the integer size range of 2^{14} to 2^{16} bits. When only calling one arithmetic at a time, our implementations are consistently faster than CGBN. However, CGBN scales much better over the number of

Bits	Instances	CGBN	C-V1	C-V2	C-V3	C-V4	C-V5	F-V1	F-V2	F-V3
2^{18}	2^{14}	–	–	–	–	–	–	–	–	–
2^{17}	2^{15}	1	836	1131	829	1167	1150	–	–	–
2^{16}	2^{16}	35	1491	1997	1542	2055	2039	974	–	–
2^{15}	2^{17}	116	2578	3367	2719	3512	3471	1674	469	482
2^{14}	2^{18}	217	4239	5326	4476	5640	5515	2671	652	693
2^{13}	2^{19}	340	6425	7530	6722	7968	8082	3880	1035	984
2^{12}	2^{20}	526	8159	8343	9180	6981	10475	4931	1448	1281
2^{11}	2^{21}	793	6059	5575	10749	5572	15745	3836	1551	1899
2^{10}	2^{22}	822	3700	3238	12990	3909	16554	2352	1130	2492
2^{9}	2^{23}	496	1787	1722	13740	2225	16888	1122	704	2798

Table 5: Performance of one multiplication in base `u64` measured in Gu32ops (higher is better).

Bits	Instances	CGBN	C-V1	C-V2	C-V3	C-V4	C-V5	F-V1	F-V2	F-V3
2^{18}	2^{14}	–	–	–	–	–	–	–	–	–
2^{17}	2^{15}	1	–	–	–	699	699	–	–	–
2^{16}	2^{16}	35	893	1195	797	1258	1258	–	–	–
2^{15}	2^{17}	116	1564	2074	1408	2229	2226	647	–	–
2^{14}	2^{18}	217	2619	3342	2378	3774	3773	1090	242	247
2^{13}	2^{19}	340	4203	4889	3767	5961	6061	1683	352	395
2^{12}	2^{20}	526	5975	6117	5412	8350	8815	2300	513	553
2^{11}	2^{21}	793	6270	6041	6728	7305	11363	2781	738	675
2^{10}	2^{22}	822	3704	3355	7238	4342	11448	2069	803	930
2^{9}	2^{23}	496	1731	1585	8112	2138	12979	1117	580	1107

Table 6: Performance of one multiplication in base `u32` measured in Gu32ops (higher is better).

Bits	Instances	CGBN	C-V1	C-V2	C-V3	C-V4	C-V5	F-V1	F-V2	F-V3
2^{18}	2^{14}	–	–	–	–	–	–	–	–	–
2^{17}	2^{15}	11	–	–	–	–	–	–	–	–
2^{16}	2^{16}	888	1337	1775	1259	1718	1747	921	–	–
2^{15}	2^{17}	2832	2414	2081	2569	2229	2602	1595	357	350
2^{14}	2^{18}	4960	3730	1746	4162	2682	2696	1656	528	513
2^{13}	2^{19}	8625	4205	4630	6293	5094	4961	1872	806	778
2^{12}	2^{20}	13924	7557	6537	8609	4374	8981	3307	1217	1029
2^{11}	2^{21}	23424	4889	4482	10084	3721	13717	3028	979	1505
2^{10}	2^{22}	37500	3140	2741	12848	3026	17513	2180	899	1946
2^{9}	2^{23}	70093	1641	1522	14243	2024	17079	1225	728	2156

Table 7: Performance of six multiplications in base `u64` measured in Gu32ops (higher is better).

Bits	Instances	CGBN	C-V1	C-V2	C-V3	C-V4	C-V5	F-V1	F-V2	F-V3
2^{18}	2^{14}	–	–	–	–	–	–	–	–	–
2^{17}	2^{15}	11	–	–	–	–	–	–	–	–
2^{16}	2^{16}	888	878	1192	771	1275	1268	–	–	–
2^{15}	2^{17}	2832	1539	2086	1376	2236	2235	626	–	–
2^{14}	2^{18}	4960	2622	3379	2348	3337	3082	1056	192	190
2^{13}	2^{19}	8625	4024	4023	3342	4948	3841	1217	288	307
2^{12}	2^{20}	13924	4789	5417	4719	7723	6522	1068	422	441
2^{11}	2^{21}	23424	5407	6220	5575	6737	10481	1874	544	555
2^{10}	2^{22}	37500	3517	3393	6524	4467	12414	2123	496	816
2^{9}	2^{23}	70093	1771	1640	7528	2396	15135	1224	477	1062

Table 8: Performance of six multiplications in base `u32` measured in Gu32ops (higher is better).

consecutive calls, and consistently has better performance than ours for sizes in the range 2^9 to 2^{13} w.r.t. ten additions and six multiplications. Comparing Futhark and CUDA, we see that CUDA scales better in the number of arithmetic calls – especially noticeable when going from one to ten additions.

In regard to multiplication: The Futhark implementation suffers from the sub-optimal implementation described in 8.3, but also from not supporting 128-bit integer arithmetics. The CUDA implementation generally has good performance, and in line with the expected outcome of the optimizations (i.e. versions).

In regard to the addition: Both CUDA and Futhark have roughly the same performance for one addition – and it is a good performance, reaching 85% of peak device bandwidth. However, Futhark suffers a significant performance loss when scaling to ten additions, while the CUDA versions sees a less significant performance loss. The optimal versions in both languages have very consistently performance across integer sizes and number of instances.

12 Conclusion

This thesis has shown how to implement exact and efficient big integer arithmetic on a GPU. The focus has been on addition, multiplication, and division, using the high-level programming languages Futhark and C++. The arithmetic operators have been implemented at CUDA block-level, albeit algorithmic efforts have been kept general and can be fitted for other architectures. The arithmetic operators are aimed at medium-sized big integers. We have shown how to represent such integers in a GPGPU friendly way and more, what optimization strategies give consistent performance over the shape of the integers.

Regarding the addition operator, we found that carry propagation in parallel can be expressed using a scan, making it efficiently run on GPGPU. Our multiplication operator is based on the classical multiplication algorithm. The algorithm displays a quadratic amount of sequential work, yet we have shown a scheme that balances the work amongst threads and maximize performance at CUDA block-level. The scheme contains a nontrivial memory layout as a byproduct of the way multiplication convolutions are arranged. We have shown how to efficiently construct the layout, both on paper and in implementation.

The approach for the division operator is based on multiplication by the inverse divisor. We leverage the existence of a shift operator to stay in the domain of big integers, such that it becomes multiplication by a shifted inverse. The approach is grounded in another work detailing an algorithm to find such a shifted inverse. We have adapted the algorithm to big integers, while also expanding on it in the form of handling an otherwise unconsidered cornercase. We have shown how to use this algorithm in order to efficiently find the exact remainder and quotient of two big integers. The division is parameterized over a big integer multiplication, addition, and subtraction method, and its complexity mirrors that of its multiplication method. While we were not able to produce a correct and efficient implementation of the algorithm for GPGPU, the necessary steps to do so have been outlined. Some of the steps have been implemented, including a sign-extended parallel subtraction operator based on our addition operator. Moreover, a correct but inefficient sequential C implementation has been produced, serving as proof of concept for the division algorithm.

Our addition and multiplication operator have been tested against a state of the art CUDA library for big integers. We found that the block-level approach overall exhibit good performance at the target size. Both operators have better performance than the state of the art library when run once, but consecutively applying the operators scales worse and comes with a performance cost. For addition, our C++ implementation is still competitive given this performance cost, while our Futhark implementation becomes slow. For multiplication, our Futhark implementation suffers from an inefficient subroutine, along with not supporting all of the low-level data types that C++ supports, and thus, cannot be as optimized. Our C++ implementation is still competitive at the target integer size, given the performance cost that comes with scaling the number of arithmetic operators applied.

Future work This thesis leaves future work regarding the division implementation. First and foremost, the parallel implementation should be validated. In relation to that, it should be considered whether the increased number of guard digits is a true revision to the algorithm, or due to an error in the sequential prototype. After it validates, the remaining inefficient subroutines should be reworked, mainly ensuring proper growth of the multiplication input lengths over refinement iterations. The last step is to evaluate its performance over big integer sizes, divisor precision, and the underlying multiplication methods, such as FFT and classical.

References

[1] Cuda C++ Programming Guide Version 12.4. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/`. Accessed: May 5th 2024.

[2] Futhark Library Documentation. `https://futhark-lang.org/docs/prelude/`. Accessed: May 19th 2024.

[3] Futhark User's Guide. `https://futhark.readthedocs.io/en/latest/`. Accessed: May 5th 2024.

[4] TechPowerUp GPU Database. `https://www.techpowerup.com/gpu-specs/geforce-gtx-1650-super.c3411`. Accessed: May 6th 2024.

[5] Kristian Olesen Amar Topalovic, Walter Restelli-Nielsen. Multiple-precision Integer Arithmetic. Data Parallel Programming final project, University of Copenhagen, January 2022. `https://futhark-lang.org/student-projects/dpp21-mpint.pdf`.

[6] Hovhannes Bantikyan. Big Integer Multiplication with CUDA FFT(cuFFT) Library. *International Journal of Innovative Research in Computer and Communication Engineering*, 2:6317–6325, 2014. URL: `https://api.semanticscholar.org/CorpusID:14759606`.

[7] G.E. Blelloch. Scans as Primitive Parallel Operations. *IEEE Transactions on Computers*, 38(11):1526–1538, 1989. `doi:10.1109/12.42122`.

[8] Adrian P Dieguez, Margarita Amor, Ramón Doallo, Akira Nukada, and Satoshi Matsuoka. Efficient High-precision Integer Multiplication on the GPU. *The International Journal of High Performance Computing Applications*, 36(3):356–369, 2022. `arXiv:https://doi.org/10.1177/10943420221077964`, `doi:10.1177/10943420221077964`.

[9] Martin Elsman, Troels Henriksen, and Cosmin E. Oancea. *Parallel Programming in Futhark*. 2018. URL: `https://futhark-book.readthedocs.io`.

[10] Niall Emmart and Charles C. Weems. High Precision Integer Addition, Subtraction and Multiplication with a Graphics Processing Unit. *Parallel Process. Lett.*, 20:293–306, 2010. URL: `https://api.semanticscholar.org/CorpusID:46446827`.

[11] Torbjörn Granlund et al. Gnu Multiple Precision Arithmetic Library Version 6.3.0 Manual. `https://gmplib.org/manual/index`. Accessed: May 9th 2024.

[12] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. Futhark: Purely Functional GPU-programming with Nested Parallelism and In-place Array Updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 556–571, New

York, NY, USA, 2017. ACM. URL: `http://doi.acm.org/10.1145/3062341.3062354`, `doi:10.1145/3062341.3062354`.

[13] Mioara Joldes, Jean-Michel Muller, Valentina Popescu, and Warwick Tucker. CAMPARY: Cuda Multiple Precision Arithmetic Library and Applications. In *International Congress on Mathematical Software*, 2016. URL: `https://api.semanticscholar.org/CorpusID:8130451`.

[14] Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, Boston, third edition, 1997.

[15] Simon Marlow et al. Haskell 2010 Language Report. `https://www.haskell.org/onlinereport/haskell2010/`, 2010. Accessed: May 9th 2024.

[16] Duane Merrill and Michael Garland. Single-pass Parallel Prefix Scan with Decoupled Look-back. `https://research.nvidia.com/sites/default/files/pubs/2016-03_Single-pass-Parallel-Prefix/nvr-2016-002.pdf`, March 1 2016.

[17] P. Munksgaard, T. Henriksen, P. Sadayappan, and C. Oancea. Memory Optimizations in an Array Language. In *2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC) (SC)*, pages 424–438, Los Alamitos, CA, USA, nov 2022. IEEE Computer Society. URL: `https://doi.ieeecomputersociety.org/`.

[18] NVlabs. CGBN: CUDA Accelerated Multiple Precision Arithmetic (Big Num) using Cooperative Groups, Library Version XMP 2.0. `https://github.com/NVlabs/CGBN`. Accessed: May 14th 2024.

[19] Cosmin E. Oancea and Stephen M. Watt. GPU Implementations for Midsize Integer Addition and Multiplication, 2024. `arXiv:2405.14642`.

[20] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., USA, 3rd edition, 2000.

[21] Stephen M. Watt. Efficient Generic Quotients Using Exact Arithmetic. 2023. `arXiv:2304.01753`.

Appendix

A

Proof of associativity for operator $\otimes$ of Equation (14).

We want to show:

$$x \otimes (y \otimes z) = (x \otimes y) \otimes z \tag{36}$$

From right-to-left we have:

$$\begin{aligned}
(x \otimes y) \otimes z &= (\ \underbrace{(((x \,\&\, (y \gg 1)) \mid y) \,\&\, 1) \mid (x \,\&\, y \,\&\, 2)}_{\alpha}\ ) \otimes z && (37)\\
&= (((\alpha \,\&\, (z \gg 1)) \mid z) \,\&\, 1) \mid (\alpha \,\&\, z \,\&\, 2) && (38)\\
&= (((\alpha \,\&\, 1) \,\&\, (z \gg 1)) \mid (z \,\&\, 1)) \mid ((\alpha \,\&\, 2) \,\&\, z) && (39)
\end{aligned}$$

From left-to-right we have:

$$\begin{aligned}
x \otimes (y \otimes z) &= x \otimes (\ \underbrace{((y \,\&\, (z \gg 1)) \mid z) \,\&\, 1) \mid (y \,\&\, z \,\&\, 2)}_{\beta}\ ) && (40)\\
&= (((x \,\&\, (\beta \gg 1)) \mid \beta) \,\&\, 1) \mid (x \,\&\, \beta \,\&\, 2) && (41)\\
&= ((x \,\&\, (\beta \gg 1) \,\&\, 1) \mid (\beta \,\&\, 1)) \mid (x \,\&\, (\beta \,\&\, 2)) && (42)
\end{aligned}$$

We must now show that Equation (39) is equal to (42). Consider the second clause. Since (1 & 2) is 0, so is the first clause of both (α & 2) and (β & 2). Furthermore, we have that 2 & 2 is the same as 2. Thus, we have:

$$\begin{aligned}
(\alpha \,\&\, 2) \,\&\, z &= (x \,\&\, y \,\&\, 2) \,\&\, z && (43)\\
&= x \,\&\, (y \,\&\, z \,\&\, 2) && (44)\\
&= x \,\&\, (\beta \,\&\, 2) && (45)
\end{aligned}$$

Consider the first clause. Again, (2 & 1) is 0, so the second clause of (α & 1) is 0:

$$\begin{aligned}
&((\alpha \,\&\, 1) \,\&\, (z \gg 1)) \mid (z \,\&\, 1) && (46)\\
&\quad= ((((x \,\&\, (y \gg 1)) \mid y) \,\&\, 1) \,\&\, (z \gg 1)) \mid (z \,\&\, 1) && (47)\\
&\quad= (((x \,\&\, (y \gg 1) \,\&\, 1) \mid (y \,\&\, 1)) \,\&\, (z \gg 1)) \mid (z \,\&\, 1) && (48)\\
&\quad= ((x \,\&\, (y \gg 1) \,\&\, (z \gg 1) \,\&\, 1) \mid (y \,\&\, (z \gg 1) \,\&\, 1)) \mid (z \,\&\, 1) && (49)
\end{aligned}$$

We have $((y \gg 1) \,\&\, (z \gg 1)) = (y \,\&\, z \gg 1) = (y \,\&\, z \,\&\, 2 \gg 1)$ and get:

$$\begin{aligned}
&= (x \,\&\, ((y \,\&\, z \,\&\, 2) \gg 1) \,\&\, 1) \mid (y \,\&\, (z \gg 1) \,\&\, 1) \mid (z \,\&\, 1)) && (50)\\
&= (x \,\&\, ((y \,\&\, z \,\&\, 2) \gg 1) \,\&\, 1) \mid ((y \,\&\, (z \gg 1)) \mid z) \,\&\, 1) && (51)\\
&= (x \,\&\, (\beta \gg 1) \,\&\, 1) \mid (\beta \,\&\, 1) && (52)
\end{aligned}$$

Thus, Equations (39) and (42) are equal, and hence, (36) holds and $\otimes$ is associative. □

Proof that (15) is left-associative neutral element for (14).

By exhaustive evaluation we have:

$$2 \otimes 0 = (((2 \mathbin{\&} (0 \gg 1)) \mid 0) \mathbin{\&} 1) \mid (2 \mathbin{\&} 0 \mathbin{\&} 2) = 0 \tag{53}$$
$$2 \otimes 1 = (((2 \mathbin{\&} (1 \gg 1)) \mid 1) \mathbin{\&} 1) \mid (2 \mathbin{\&} 1 \mathbin{\&} 2) = 1 \tag{54}$$
$$2 \otimes 2 = (((2 \mathbin{\&} (2 \gg 1)) \mid 2) \mathbin{\&} 1) \mid (2 \mathbin{\&} 2 \mathbin{\&} 2) = 2 \tag{55}$$
$$2 \otimes 3 = (((2 \mathbin{\&} (3 \gg 1)) \mid 3) \mathbin{\&} 1) \mid (2 \mathbin{\&} 3 \mathbin{\&} 2) = 3 \tag{56}$$

□

B

Listing 20: CUDA convolution memory transactions of *convmul* with sequentialization factor of 4 to prepare calling *badd* on the four pairs of parts in registers, from file `ker-mul.cu.h` (slightly edited), with base class `Base`, big integer size `m`, and `ipb` instances per block.

```
cp4Regs2Shm(typename Base::uint_t* lhc0, typename Base::uint_t* lhc1,
            typename Base::uint_t* shmem){
    uint32_t off = threadIdx.x;
    uint32_t str = ipb * m/2;
    shmem[off] = lhc0[0];
    shmem[off+str] = lhc0[1];
    shmem[off+2*str] = lhc0[2];
    shmem[off+3*str] = lhc0[3];

    shmem[str-1-off] = lhc1[0];
    shmem[2*str-1-off] = lhc1[1];
    shmem[3*str-1-off] = lhc1[2];
    shmem[4*str-1-off] = lhc1[3];
}

cpShm24Regs(typename Base::uint_t* shmem, typename Base::uint_t* lhc0,
            typename Base::uint_t* lhc1){
    uint32_t off = threadIdx.x*2;
    uint32_t off_inst = 2*off % m;
    uint32_t str = ipb * m/2;
    lhc0[0] = shmem[off];
    lhc0[1] = shmem[off+str];
    lhc0[2] = shmem[off+2*str];
    lhc0[3] = shmem[off+3*str];

    lhc1[2] = shmem[off+1];
    lhc1[3] = shmem[off+str+1];
    lhc1[0] = (off_inst) ? shmem[off+2*str-1] : 0;
    lhc1[1] = (off_inst) ? shmem[off+3*str-1] : 0;
}
```

C

Proof of associativity for operator $\odot$ of Equation (32).

We want to show:

$$(l1,\ e1) \odot ((l2,\ e2) \odot (l3,\ e3)) = ((l1,\ e1) \odot (l2,\ e2)) \odot (l3,\ e3) \tag{57}$$

By definition we get from left-to-right:

$$\begin{aligned}
&(l1,\ e1) \odot ((l2,\ e2) \odot (l3,\ e3)) && (58)\\
&\quad = (l1,\ e1) \odot (l3 \vee (e3 \wedge l2),\ e2 \wedge e3) && (59)\\
&\quad = ((l3 \vee (e3 \wedge l2)) \vee ((e2 \wedge e3) \wedge l1),\ e1 \wedge (e2 \wedge e3)) && (60)\\
&\quad = ((l3 \vee (e3 \wedge (l2 \vee (e2 \wedge l1))),\ (e1 \wedge e2) \wedge e3) && (61)\\
&\quad = (l2 \vee (e2 \wedge l1),\ e1 \wedge e2) \odot (l3,\ e3) && (62)\\
&\quad = ((l1,\ e1) \odot (l2,\ e2)) \odot (l3,\ e3) && (63)
\end{aligned}$$

Equation (63) is equal to the right-hand-side of (57), and thus, operator $\odot$ is associative. □

Proof that (F, T) *is left-associative neutral element for $\odot$ of (32).*

By exhaustive evaluation we have:

$$\begin{aligned}
(\mathtt{F},\ \mathtt{T}) \odot (\mathtt{F},\ \mathtt{F}) &= (\mathtt{F} \vee (\mathtt{F} \wedge \mathtt{F}),\ \mathtt{F} \wedge \mathtt{F}) = (\mathtt{F},\ \mathtt{F}) && (64)\\
(\mathtt{F},\ \mathtt{T}) \odot (\mathtt{F},\ \mathtt{T}) &= (\mathtt{F} \vee (\mathtt{T} \wedge \mathtt{F}),\ \mathtt{T} \wedge \mathtt{T}) = (\mathtt{F},\ \mathtt{T}) && (65)\\
(\mathtt{F},\ \mathtt{T}) \odot (\mathtt{T},\ \mathtt{F}) &= (\mathtt{T} \vee (\mathtt{F} \wedge \mathtt{F}),\ \mathtt{T} \wedge \mathtt{F}) = (\mathtt{T},\ \mathtt{F}) && (66)\\
(\mathtt{F},\ \mathtt{T}) \odot (\mathtt{T},\ \mathtt{T}) &= (\mathtt{T} \vee (\mathtt{T} \wedge \mathtt{F}),\ \mathtt{T} \wedge \mathtt{T}) = (\mathtt{T},\ \mathtt{T}) && (67)
\end{aligned}$$

□