UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE

Master's Thesis

Efficient GPU Implementation of Multi-Precision Integer Division

Aske N. Raahauge, Martin B. Marchioro & Marc I. Løvenskjold

Supervisor: Cosmin E. Oancea

Submitted: June 2, 2025

Abstract

Efficient arithmetic on multi-precision integers is a cornerstone of many scientific and cryptographic applications that require computations on integers that exceed the native sizes supported by modern processors. While GPU-efficient addition and multiplication has been well explored, division has been subject to less attention due to its greater algorithmic complexity. This thesis attempts to bridge this gap by implementing a GPU-efficient division, that works on integers up to 250.000 bits in size which fit in a single CUDA block, exploiting the temporal data reuse of fast scratchpad memory. The algorithm is based on the Newton-inspired method for computing the reciprocal of the divisor presented by Watt in [33], which performs exact division entirely within the integer domain. Our main product is an efficient implementation in CUDA, although not outperforming the popular CGBN library, it demonstrates promising scalability results. Moreover, to our knowledge, we are the first to implement a parallel division capable of operating on inputs larger than 2^{15} bits. Finally, we implement a Futhark version to explore the practical aspects of using a high-level functional language, and conclude that current compiler limitations introduce considerable overheads and scalability issues.

Contents

1 Introduction

Modern processors are typically limited to 32-bit or 64-bit integer operations, imposing inherent limits on the size of numbers that can be directly manipulated. However, many scientific and cryptographic applications require computations on integers that far exceed these native sizes. To address this need, multiple precision arithmetic extends standard operations to arbitrarily sized integers by representing them as arrays of smaller fixed-size words, avoiding the rounding errors and numerical instability introduced by floating-point representations.

As modern algorithms demand ever-larger integer sizes, the importance of developing efficient multiple precision arithmetic increases, especially in parallel settings, as the slowing of Moore's Law shifts the focus from faster processors to greater parallelism. While efficient parallel implementations for addition and multiplication have been successfully showcased in the past, the division counterpart continues to lag behind in terms of efficiency and scalability, likely due to its greater algorithmic complexity. Yet, division is particularly important, since it serves as a building block for other fundamental operations, such as the Euclidean algorithm, which efficiently computes the greatest common divisor (GCD) of two integers. Improving the performance of multi-precision division is therefore key to accelerating a wide range of higher-level algorithms.

This thesis addresses this gap by extending parallel arithmetic operations on multiple precision integers with an efficient implementation of division, complementing existing addition and multiplication implementations [30]. More specifically, we present a **CUDA** implementation capable of performing division on integers of approximately 250.000 bits in size, which fit in fast scratchpad memory of a single **CUDA** block. This approach enables us to exploit temporal data reuse through fast scratchpad memory, which offers substantially lower latency compared to global memory. Moreover, to our knowledge, we are the first to report a GPU-based multi-precision division implementation capable of operating on inputs larger than 2^{15} bits.

Our algorithm is based on a method proposed by Watt in [33], which performs exact division by computing the whole shifted inverse of the divisor, a technique that adapts Newton's method to remain entirely within the domain of integers. Moreover, the algorithm is generic with the underlying multiplication technique, allowing it to be paired with different multiplication algorithms, such as classical, Karatsuba, or FFT-based methods, depending on the operand size.

We evaluate our **CUDA** prototype against the state-of-the-art **CGBN** arithmetic library, developed by NVIDIA, which supports arithmetic operations for integers up to 2^{15} bits. Although our implementation does not outperform **CGBN**, it shows promising results, indicating that it may prove competitive for sizes close to 2^{18} bits. However, our main advantage lies in supporting integers with up to eight times more bits than the **CGBN** limit, which corresponds to representing values that are up to 2^{229376} times larger.

We also implement and evaluate a high-level version of the algorithm in Futhark, which unfortunately does not produce performant code due to compiler limitations.

We discuss advantages and challenges of this high-level functional approach, and conclude that there are significant overheads and scalability issues rooted to the compiler shortcomings when handling nested parallelism with irregular structures. This may well serve as inspiration for developing new optimizations passes in Futhark that address said limitations, thereby improving performance for a broader class of parallel applications.

Finally, we extend our CUDA implementation to compute greatest common divisor (GCD) via the Euclidean algorithm, and provide a performance comparison with CGBN.

Main Contributions

- An adaptation of the whole shifted inverse algorithm [33] to support an unsigned multi-precision integer representation, along with corrections of previously unspecified behavior.
- An efficient CUDA division implementation supporting integers up to 2^{18} bits, which to our knowledge is the only parallel division algorithm capable of handling integers up to this size.
- A high-level implementation in futhark, exploring overhead and scalability issues related to compiler limitations.
- A benchmark driven performance evaluation of our CUDA division against CGBD, finding it $3\times$ slower at 2^{15}-bit integers. However, unlike CGBD, our implementation scales to 2^{18} bits.

The thesis is structured as follows. Section 2 provides a brief overview of related academic work. Section 3 gives a short introduction to the technical foundations, including general GPU architecture, the CUDA programming model, the Futhark programming language, along with our representation of a multi-precision integer. Section 4 touches upon Newton's method and how it relates to integer division, which serves as a foundation for later sections. Section 5 showcases previous work consisting of parallel implementations of supporting arithmetic operations needed for the division algorithm. Section 6 analyses the whole shifted inverse algorithm and it's properties, while Section 7 presents revisions to the algorithm that ensure correct handling of negative values and that highlights existing bugs. Section 8 provides a detailed overview of our high-performance CUDA prototype, showcasing general strategies, implementation specifics and limitations. Section 9 explores how to produce a semantically equivalent program in the high-level programming language Futhark. Section 10 benchmarks our implementations against the CGBN library and analyzes the results. Finally, Section 11 concludes the thesis and examines the prospect of future work.

Software Structure

The code referenced in this thesis can be found in the repository linked below. And all tests can be replicated using the Makefiles in the corresponding directories as follows e.g. `div/cuda/make`, `div/futhark/make` or `cuda/cgbn-tests/make`.

https://github.com/aske0778/midint-arithmetic-division.git

Division implementation The division implementation is located in the `div` directory, and has a sub-directory for the CUDA and FUTHARK implementations respectively. The `main.cu` file contains the benchmarking setup, while `ker-division.cu.h` contains the main CUDA kernels and device functions. The directories `binops` and `helpers` contain auxiliary functions used as part of the division itself and during benchmarking.

All major functions related to the FUTHARK implementation can be found in `div.fut`, which also contains the benchmarks. Similarly the `div-helpers.fut`, `add.fut`, `mul.fut`, `sub.fut` and `helpers.fut` all contain supporting functions similar to CUDA.

CGBN The benchmarking results for CGBN can be found in the `cuda/cgbn-tests` directory, as they extend upon an existing framework. `cgbn-perf.cu` contains the benchmarking setup, while `cgbn-kers.cu` contains the kernels.

2 Related Work

Previous work has demonstrated efficient implementations of basic arithmetic operations on parallel hardware. Our main inspiration for this thesis comes from "GPU Implementations for Midsize Integer Addition and Multiplication" by Oancea and Watt [30]. This paper and the corresponding implementation forms the basis of this thesis, since the code developed as part of this thesis builds on top of their parallel implementation of addition and multiplication. Oancea and Watt provide both implementations and performance results for integer addition as well as for classical and FFT-based (Fast Fourier Transform) multiplication of integers. In particular their FFT multiplication is shown to produce log-linear work, outperforming classical multiplication for big integers of sizes 2^{16} bits and greater. Traditionally FFT-based multiplication uses the Discrete Fast Fourier Transform (DFFT) in the domain of complex numbers, but this introduces the possibility of rounding errors and loss of precision due to leaving the domain of integers. To avoid this, the authors find primes numbers, such that the DFFT can be mapped to prime fields that are guaranteed to have all roots of unity, mapping the computation to the integral domain. A suitable prime is chosen to maximize utilization of native integer operations, loosing only one bit per multi-precision digit. This method is based on the Cooley-Tukey algorithm, and is designed for execution within a single CUDA block, allowing reuse of fast scratchpad memory. Additionally the authors examine the efficiency of FUTHARK in supporting this type of operations, however the FUTHARK implementation has overhead and scalability issues, caused by the absence of efficient sequentialization of excess parallelism in the FUTHARK compiler [30].

Previous efforts have been made to implement the algorithm by Watt [33] in same manner as addition and multiplication by [30]. One such work is presented in "Efficient Big Integer Arithmetic Using GPGPU" by Bringgaard [10], which details the process of establishing a sequential and a parallel division implementation in Futhark. This paper provides the foundation for our own implementation in Futhark, as we use some of the existing library by Bringgaard, in particular their addition, subtraction and multiplication. Working CUDA implementations of addition and classical multiplication are provided, and shown to be competitive with the CGBN library [26], for operations on multi-precision integers in the range of 2^{14} to 2^{18} bits.

A previously successful attempt at implementing arithmetic for multiple precision integers is the Cooperative Groups Big Numbers (CGBN) library, authored by NVLabs. This library supports multi-precision integers of sizes up to 32K bits, and includes high-performance implementation of most common arithmetic operations. The library advertises exceptional performance, due to operations utilizing architecture specific warp-level primitives, which allows for efficient sharing of values between individual threads in a warp. However the library stops short of supporting integers of up to the 2^{18} bits, as limited by shared memory, which prompts our thesis. We use this library to establish a baseline comparison for the limits of parallel performance on CUDA GPUs.

We include some other loosely related works that have tackled similar problems

in the field of arithmetic on multi-precision integers:

- Previous work includes looking at the algorithms behind some of the basic operations included in multi-precision libraries like GMP, and if they could be expanded to a parallel setting. Ultimately, it was found that due to the architectural differences of the CPU and the GPU, most of the existing sequential algorithms could not easily be ported to a parallel setting, instead prompting research into new inherently parallel algorithms [13].

- Efficient parallel algorithms for division has been previously explored when certain conditions are met. One such study is by Emmart and Weems, which used a modified version of Jebelean's exact division algorithm for the case when the divisor is of a single precision value. Their study utilizes this property to propose an algorithm with a complexity of $O(n/p + \log p)$ for a given number of processors p [14]. Their paper shows a significant speedup is possible when restrictions on the input are considered.

- Other multi-precision arithmetic libraries have also been developed. One such example is CAMPARY, which includes both sequential CPU and parallel CUDA implementations. It represents real numbers as unevaluated sums of multiple floating-point numbers to extend precision, using error-free transformations to manage rounding errors during arithmetic operations [18]. CAMPARY supports basic arithmetic operations such as addition, subtraction, multiplication, division, and square root, and is designed for precisions up to a few hundred bits

- A similar, but older GPU-based library, CARPREC provides arithmetic operators, mathematical functions, and data-parallel primitives. It supports both multi-digit and multi-term number formats. The library includes a GQD (Generalized Quad-Double) module that offers precision up to approximately 62 decimal digits [22]. While the authors report results for computations with up to 1000 decimal digits, such extended functionality does not appear to be available in the public implementation.

3 Background

The following section is intended to provide a brief introduction to the GPU architecture and programming models associated with this thesis, which laid the foundation for the work done. We describe the necessary preliminaries, beginning with general concepts of modern GPU architecture, followed by introductions to the CUDA programming model and the Futhark programming language, which are the two languages used throughout this thesis. Finally, we define the underlying representation of the multiple precision integer considered in this thesis, along with its properties.

3.1 GPU Architecture

In this section, we present an overview of the GPU architecture relevant to understanding the work in this thesis. The terminology will be based upon NVIDIA technology as their CUDA programming model has become the industry standard within the field of parallel computing [6]; however, the underlying principles should be generic and therefore relevant for other architectures as well.

Modern CPUs include upwards of dozens of complex cores, and advertise advanced features like out-of-order execution and branch prediction, designed to minimize the execution time of sequential tasks. GPUs in contrast incorporate hundreds or thousands of simpler processing cores, which are then explicitly run in parallel in order to maximize the total amount of work completed per unit time (throughput). These GPUs feature a hierarchical hardware structure designed to manage and execute parallel workloads efficiently. More specifically, the GPU contains a number of Streaming Multiprocessors (SMs) ranging from a few to over a hundred for high-end models. Each SM acts as an independent processor capable of executing multiple thread blocks concurrently as part of a kernel. The SM is responsible for the scheduling and execution of threads, which happens in groups of 32 parallel threads at a time called a *warp*. Each thread in a warp executes the same instruction in lockstep, which optimizes the total throughput when run on large parallel problems. The core design philosophy of a GPU is thus to execute a massive number of threads concurrently. This makes GPUs exceptionally well-suited for problems that exhibit significant data parallelism – where the same operation can be applied independently to many different data elements simultaneously.

	SM (Streaming Multiprocessors)	CUDA-core
hierarchy	high-level, contains multiple CUDA-cores	low-level, processing core
purpose	scheduling & thread management	execution of thread instructions
contains	cores, registers, cache, scheduler	simple ALU

Table 1: Brief comparison between SM and CUDA-core.

Table 1 gives a simplified overview of the connection between the SM and the processing cores it schedules, called CUDA-cores. The SM dispatches a warp of

threads to the CUDA-cores at a time. The total number of cores per SM is specific to the hardware, while warps always consists of 32 threads.
Figure 1 showcases a simplified view of the architectural differences between a CPU with four cores and a GPU containing eight SMs. The figure is color coded to represent the equivalent parts of the hardware between GPU and CPU. Some initial observations are that while GPUs have a much greater number of processing cores, they are each without a control block, and as such are dependent on the limited number of SMs for scheduling. Additionally the size and amount of lower level caches are much smaller for the GPU making them less versatile, and inefficient memory management much more costly.

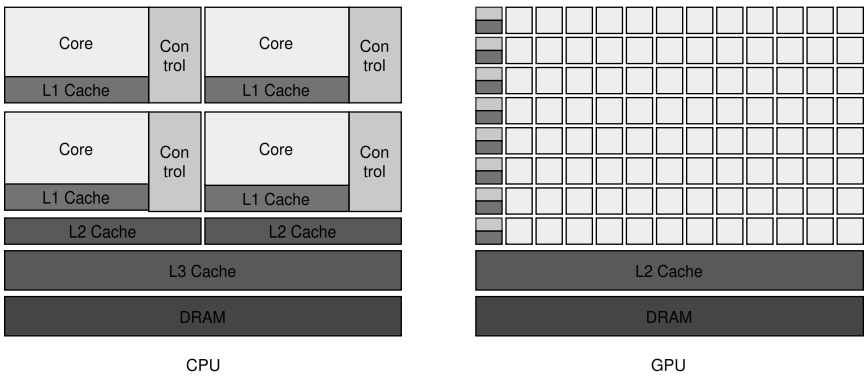


Figure 1: Figure showing the layout of CPU vs GPU.

Memory Hierarchy The memory layout of modern GPUs is split into a hierarchical structure consisting of three main layers, each one of a different size and with different behavior. An overview of the hierarchy can be seen in Figure 2, and we will briefly describe each of them below.

- Each thread has access to a number of it's own private 32-bit registers. This memory is associated with extremely low latency, and is private in the sense that no synchronization issues can occur as only one thread can read/write to it. If a thread runs out of register memory, the compiler can instead *spill* these into shared or even global memory. This practice ensures that whatever is stored in memory is never lost, however it comes at the cost of increased latency due to reads and writes to this higher level cache [4]. Thus registers are primarily used for storing variables and smaller arrays that are associated with the given thread.

- Shared memory, also called scratchpad memory, is the next immediate layer accessible by a thread. It is aptly named for being shared between all threads in a block, meaning that other threads from other blocks in the grid are not able to access it. Shared memory is associated with more latency than registers, however it is the lowest level memory in which it is possible for threads to share information aside from warp shuffles, thus any application with any amount of inter-warp communication needs to access this memory. Managing this memory efficiently is crucial for the performance of applications as the latency introduced by accesses can quickly produce

significant overhead. Shared memory can both be allocated statically and dynamically during runtime and has the same lifetime as its associated thread-block.

- Global memory is the largest and consequently slowest of the three memory layers, it is accessible to all threads and is not associated with a any specific kernel. As this memory can outlive a kernel and is instead associated with the life-time of the host application, the responsibility of allocation and de-allocation is delegated to the host. Global memory is therefore used for the initial copying of data from the host to the device.

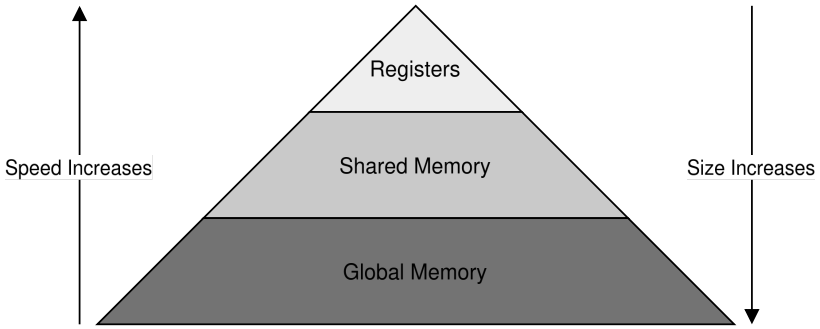


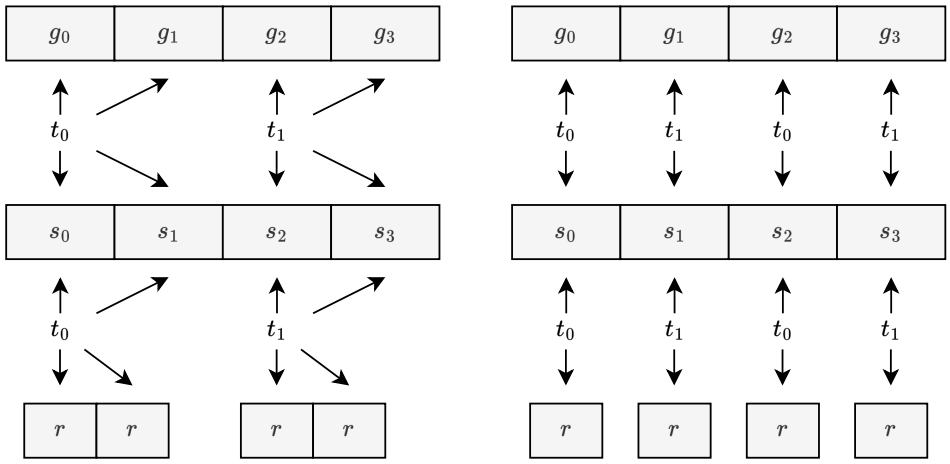
Figure 2: The CUDA memory hierarchy, showing access speed vs size

Occupancy As the main idea behind GPUs is to maximize throughput, we want to maximize the number of warps concurrently running at all times i.e. we want to maximize the *occupancy* of our program. The main idea is that when one warp stalls (e.g. is waiting for data from global or shared memory), the SM can quickly switch to another idle warp, keeping the cores busy. If occupancy is low, it means that fewer warps are available to be scheduled, reducing the SM's ability to hide latency and leading to underutilization of the underlying hardware. A number of factors affect how many threads are active at any given time, most notable is the combination of the block size, and the amount of memory it requires. Generally, as the amount of memory used by a thread block increases, the number of blocks executed concurrently on the SM decreases.

As an example, if the blocks of a kernel utilize 1024 threads each, these will be allocated in terms of 32 warps. However, as an SM has a maximum number of resident warps of 48, only one block will be scheduled at a time, meaning that each SM is underutilized by 16 warps during execution [4].

Coalesced Memory Access Some algorithms have their performance bounded by the overhead of accessing higher level memory, and we call these algorithms *memory bound* in contrast to *compute bound*. However, regardless of the bound, optimizing the accesses to global memory is imperative to ensure good performance, which on GPUs is ensured through coalesced memory access. Coalesced memory is optimized when consecutive threads within a block access contiguous data or memory addresses in global memory. It is encouraged to perform memory accesses like this which optimizes the spacial locality of a program.

Since all data transfers to and from global memory happens by the memory bus, coalesced access will reduce the total amount of reads/writes that has to be performed when accessing the data. As an example, the Nvidia A100 has a memory bus with a width of 5120 bits, this means that if all 32 threads in a warp read 64-bit integers each in a coalesced access in global memory, it could be done in a single memory transfer. The worst case occurs when each thread accesses different memory addresses such that the stride is larger than the width of the bus, resulting in one access for each thread, significantly slowing down the computation [10].



(a) An example of uncoalesced access. (b) An example of coalesced access.

Figure 3: Examples of both uncoalesced and coalesced memory access to global memory, when the sequentialization factor is 2.

Figure 3 shows an example of both uncoalesced and coalesced memory accesses, with arrows representing reads/writes to a memory cell. The separation between registers is meant to emphasize that registers are thread exclusive. Figure 3a shows each thread accessing multiple consecutive elements in global memory, resulting in poor spacial locality when processing high volumes of data. In comparison Figure 3b shows consecutive threads access consecutive elements of global memory, resulting in coalesced access. This optimizes locality, as each warp of threads can fit into the width of the memory bus. Both examples use the sequentialization factor of $Q = 2$, meaning that each thread fetches two elements from memory. In practice, this factor changes based on the input, which will affect the performance when memory access is uncoalesced [28].

Having some degree of serialization can improve performance by reducing the need for inter-thread communication, and allow for better temporal locality. An example is that of a simple reduction. A parallel implementation with no serialization would involve each thread computing a single reduction before performing an expensive write back to shared memory. In contrast, with some degree of serialization multiple elements are reduced sequentially by each thread. This approach results in more work for each thread, however these computations may still be faster, as they result in significantly less accesses to shared memory. In practice, the degree of serialization is often chosen as a result of testing different

values, and picking the best performing one on the given problem [27].

3.2 The CUDA Programming Model

CUDA is a heterogeneous programming model, meaning that it integrates more than one kind of processing unit in order to handle different tasks. It acts as a framework on top of C++ allowing for easy integration between the CPU, called the *host*, and the GPU, called the *device*. To distinguish which code runs on which processor, CUDA introduces a set of function specifiers:

- The `__host__` defines a function that can only be called and executed by the host i.e. on the CPU.
- Similarly, functions annotated with `__device__`, can only be called and executed by the device, and are not allowed to be called from the host.
- Lastly, functions annotated with `__global__`, also called kernels, are functions called by the host that signal to start execution of the function on the device.

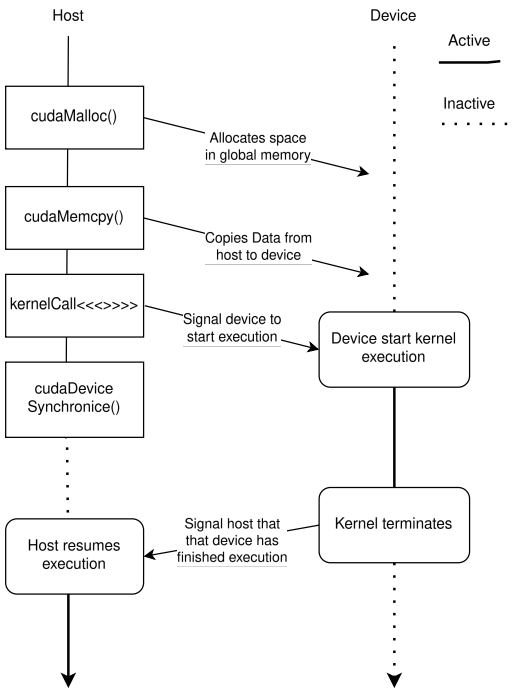


Figure 4: Execution sequence between host and device

When writing in CUDA the programmer has to manage the control-flow between the host and the device using synchronization barriers. In particular, when a kernel returns control flow to the host, all threads are not necessarily finished computing. In this case if the host requires the result, a barrier is used tells the calling host-process to wait for the device to have returned. This process of interweaving control flow between the host and device is showcased in Figure 4.

Individual threads on the device itself can get our of sync as well, this can happen when thread-diverging behavior is necessary to achieve a desired functionality,

or simply due to an uneven distribution of the workload. In some cases this can result in a data dependency due to a race condition between threads, if multiple threads attempt to read/write to a shared memory location [29]. In order to prevent this, the programmer can programatically synchronize all threads in a block, providing a barrier between reads and writes. However, an overuse of this can negatively impact performance, as it potentially causes a large number of threads to idle for an extended period of time.

In addition to the synchronization barriers CUDA offers atomic functions which can be used to interact with shared or global memory while ensuring atomicity. These functions are among others defined for all standard arithmetic operations and bitwise operations. They work by reading a value from a shared address, modifying it according to the given operation, before writing back to the same address. These functions can easily be used to implement block-level reductions, however due to their atomic nature of blocking other threads from reading or writing to the memory address, this is generally inefficient. Additionally, they do not work as general barriers, meaning that a synchronization is often required afterwards to ensure memory consistency across threads [4].

Thread Hierarchy The CUDA programming model contains a given number of virtual threads, which are bundled into warps of 32. When a kernel is launched these warps are scheduled as part of block, which are matrices of three dimensions in which the threads are exposed to the programmer. Additionally, a block is itself a part of a three dimensional grid, in which all blocks of the given kernel are stored. In practice both the block and the grid can have any shape between one, two or three dimensions, which is typically chosen by the programmer to fit the shape of the data, as the dimensionality is merely an abstraction of the underlying implementation. Each thread can then be accessed by the programmer by specifying the corresponding identifier of the block in the dimensions $\{x, y, z\}$ as seen in Figure 5.

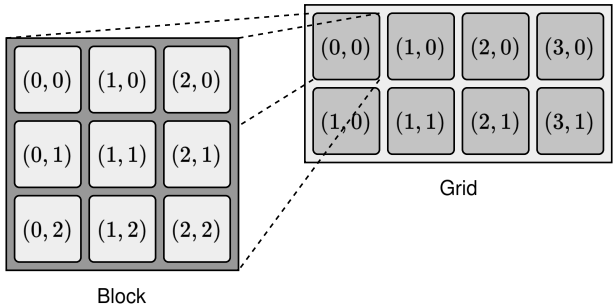


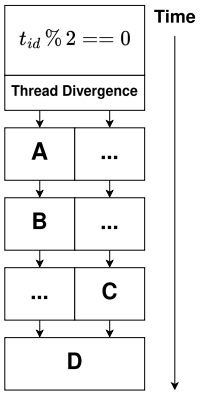
Figure 5: Two-dimensional grid (right) of two-dimensional blocks (left), each containing individual threads.

The blocks within a grid are required to execute independently, a concept know as *block independence*, allows them to be scheduled in any order and enables scalable execution. This means that the result of a kernel is invariant to the specific scheduling of thread-blocks during runtime, and allows an implementation to scale with the number of cores available on the given hardware [4].

Thread divergence GPUs make use of the Single Instruction Multiple Threads (SIMT) execution model, which is a multi threaded extension of the Single Instruction Multiple Data (SIMD) model. The result is that threads execute the same instructions in warps of 32, which results in very high throughput when all threads are engaged in relevant computation. A downside of this approach is that while individual threads programatically look to be separable, they are not. Instead, if diverging paths of execution occur within a warp, the SM will serialize them, essentially taking both paths in series. This means that for optimal performance, one should strive to eliminate diverging execution paths for the different threads within a warp [4].

```
if (threadIdx.x % 2 == 0) {
    A;
    B;
} else {
    C;
}
D;
```

(a) Code causing thread divergence



(b) Serialization caused by diverging threads

Figure 6: Example of thread divergence.

The above Figure 6 showcases a simple example of thread divergence, where the threads within a block will be divided evenly between the two branches based on the individual thread identifiers. However as the split does not cleanly divide a warp, the divergent paths are serialized.

3.3 The Futhark Programming Language

Futhark is a high-level, pure functional programming language designed to ease data parallel programming on the GPU. It compiles to intermediate code in both sequential C, OpenCL, and CUDA, meaning that it is hardware-agnostic in the sense that it can compile to both sequential CPU and parallel GPU code [16]. Traditional GPU programming models, such as CUDA or OpenCL, require manual management of threads, memory hierarchies, and synchronization, which can be error-prone and complex. Futhark abstracts these details away by providing a functional programming model that automatically compiles to efficient GPU code. The language is statically typed and free of side effects, which simplifies reasoning about program behavior, instead relying on aggressive compiler optimizations.

One of the key selling points of Futhark is the efficient parallel implementation of *Second-Order Array Combinators* (SOACs), which are higher-order functions that operate on arrays and encapsulate common parallel patterns. By leveraging SOACs, Futhark can be written in a style that closely resembles mathematical specifications, allowing concise and readable programs, while achieving efficient

parallel performance. Some of these main SOACs are described below, as combinations of these are what allow for complex expressions to be modeled in parallel and analyzed. When analyzing the asymptotic time complexity of the parallel implementation of a given algorithm, one considers the work and depth of the algorithm. Work considers the classical notion of asymptotic time complexity, covering the total amount of work done as part of the algorithm, across all operations that may be performed in parallel. The depth, also called span, considers the longest sequential dependency of the algorithm, and describes the degree of parallelism the algorithm exhibits. As an example, if an algorithm has a span of $O(k)$ and is executed on a theoretical machine with infinite threads, it will terminate in $O(k)$ time. In the same sense, if an algorithm has a constant span, it will execute in $O(1)$ time on our theoretical machine, regardless of the total amount of work. When describing the asymptotic complexities below, we assume that the operators carried by the higher order functions are executed in constant ($O(1)$) time, as the complexities naturally depend on this.

The map combinator is a direct implementation of a mathematical mapping, and takes in a function f and *maps* the input array to an output array where the function has been applied to all elements. The total work of the mapping is $O(n)$. However since there is no sequential dependency between the individual function applications, each operation can be performed in parallel, resulting in a span of $O(1)$.

$$\texttt{map} :: (\alpha \to \beta) \to [\alpha] \to [\beta]$$
$$\texttt{map}\ f\ [a_0,\ a_1,\ ...,\ a_{n-1}] \equiv [f\ a_0,\ f\ a_1,\ ...,\ f\ a_{n-1}]$$

The reduce is semantically similar to that of fold as seen in other functional programming languages, with a few key differences [2, 8]. Reduce enforces associativity of the operator and requires an explicit neutral element (or identity element) to ensure correctness in parallel execution. The total work of reduce is $O(n)$, as each element is processed exactly once. Additionally, due to the associative property of $\oplus$, the computation can be structured as a balanced binary tree, where partial reductions are computed in parallel. This results in a span of $O(\log n)$, meaning the computation depth grows logarithmically with input size.

$$\texttt{reduce} :: (\alpha \to \alpha \to \alpha) \to \alpha \to [\alpha] \to \alpha$$
$$\texttt{reduce}\ \oplus\ e\ [a_0,\ a_1,\ ...,\ a_{n-1}] \equiv e \oplus a_0 \oplus a_1 \oplus \cdots \oplus a_{n-1}$$

The scan (or *prefix scan*), shares similarities with sequential fold operations but differs in that it retains all intermediate reduction results rather than producing a single final value. Like reduce, scan requires an associative operator and a neutral element to guarantee correct parallel execution. It is asymptotically equivalent to a reduce operation, as it requires only an extra phase called the upsweep, which is equal in complexity. The scan combinator comes in two variants, an *inclusive* and an *exclusive* scan, with the difference being that elements are reduced up until i at index i for the inclusive scan, and only $i - 1$ for the exclusive scan. The

native function in futhark implements an inclusive scan.

$$\texttt{scan} :: (\alpha \to \alpha \to \alpha) \to \alpha \to [\alpha] \to [\alpha]$$
$$\texttt{scan} \ \oplus \ e\,[a_0,\, a_1,\, ...,\, a_{n-1}] \equiv [a_0,\, a_0 \oplus a_1,\, ...,\, a_0 \oplus \cdots \oplus a_{n-1}]$$

The key value of using Futhark lies in modeling problems through the combination of these higher order functions in order to generate efficient parallel code, which maintains readability. In this regard, Futhark does make some sacrifices as well. Most notable of which is it requires that nested arrays must be regular, as the compiler needs to perform an incremental flattening on these in order to efficiently schedule the operations across parallel hardware. In extension of this, recursion is completely unsupported as well, despite the functional nature, as the compiler currently does not infer recursion bounds, which may result in poor performance when run in parallel [17].

3.4 Representation of Multiple Precision Integers

In the context of performing arithmetic on multiple precision integers we may need an arbitrary amount of space in order to represent them. In general, as the size of integers increase beyond the scope of what can be computed on traditional hardware with fixed-point arithmetic, it becomes unfeasible to store numbers using words of a fixed number of bits, corresponding to the size of processor registers. Instead, we implement these numbers using a statically sized array u with elements composed of individual statically sized integers each u_i. Similar to the format of a polynomial, each u_i corresponds to the product of the value at index i and the base lifted to the power of the index itself. Formally we use the following representation as previously used in [30] which is defined as follows

Definition 1. *(multiple precision integer).*

We say that an integer $u \in \mathbb{N}$ can be expressed in base $B \in \mathbb{N}^+$ with m digits such that

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

Additionally, we denote the precision of an integer as the number of base-B digits, i.e.

$$prec_u = \lfloor \log_B |u| \rfloor + 1$$

When choosing the size of the base B (or the *radix*) of the big integer we intentionally choose the size of a machine word, as this allows for the use of efficient implementations of arithmetic operations on the individual digits. More specifically, we choose each index to be represented by an unsigned integer `uint_t` of base B, as this allows for the most direct mapping between our big integers and regular ones. In fact the representation of these big integers can be intuitively viewed as a big binary of size $B \cdot m$ bits, stored in little endian; Thus, if the word-size is 32 bits and m is 16, then the result is a 512-bit integer, where the bits 0-32 represent the least significant bits. The big integer thus represents a $(B \cdot m)$-bit unsigned integer.

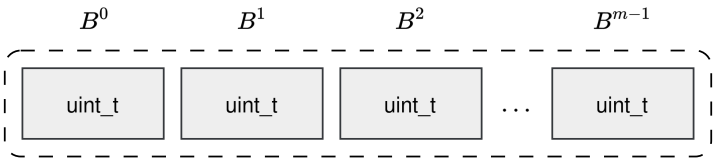


Figure 7: Representation of a mutiple precision integer of base B.

Generally in integer arithmetic, we distinguish between two types of integers; arbitrary precision, and exact precision integers. Arbitrary precision integers are able to dynamically scale based on the required size during runtime. These types of integers never lose precision as long as there is enough memory to store the result, and are thus commonly used for multiple precision arithmetic libraries such as GMP [1]. The problem with arbitrary precision is that since the sizes of the integers can change during runtime, they have to be stored in dynamically allocated memory. Due to the memory layout of modern GPUs (as presented in Section 3.1), the overhead related to accessing shared dynamic memory is much higher than when accessing thread-level static memory. Additionally, dynamically changing workloads during runtime maps poorly to parallel hardware, as it becomes much harder to distribute the collective work in a load-balanced way, practically rendering arbitrary precision arithmetic unproductive on the GPU.

Instead for the purposes of our implementation we choose to represent the integers using exact precision, meaning that all of the numbers are bounded. The sizes can then be arbitrarily set either statically or during runtime, meaning that the exact size of the integers is always known. This allows for better load-balancing between threads, and for the integers to be stored in the lowest level memory for faster lookup.

Name	Definition
`uint_t`	The base
`ubig_t`	Double the base
`uquad_t`	Quadruple the base
`carry_t`	Size of carries
bits	Number of bits in base
HIGHEST	Maximum value of base

Table 2: Big integer interface definitions from [10]

By default the multiple precision integer as per Definition 1 is meant to represent a big unsigned number. However, at times we may need to account for the sign as part of some algorithm, and as such need to represent signed integers. The most common signed number representation is *two's complement* which requires bitwise negation, however for the sake of simplicity, we choose to go with a more simple approach. In the rare case we utilize a modified version of *signed magnitude* representation, we simply return an extra bit of information corresponding to the sign of the integer. This does have the unfortunate side effect of representing 0 and -0 as two different numbers, but since this can easily be circumvented and the sign is rarely used, we consider this a minor byproduct. Furthermore, signed

multiplication and division can be accomplished by performing the operations on the unsigned big integer representation, and then applying a bit according to the signs of the original operands afterwards.

In general we refer to the digits of a multiple precision integer as size `uint_t` corresponding to a machine word, which can be statically changed to be between 16, 32 and 64 bits. We extend this notion into a big integer interface containing definitions like `ubig_t` and `uquad_t`, corresponding to double and quadruple the word size, among others. We refer to Table 2 for an overview of the whole interface, and their definitions.

4 Division via Newton's Method

The most common method for integer division is the long division algorithm, also known as grade-school division, which involves computing one digit of the quotient at a time. Another common approach called repeated subtraction works by subtracting the divisor from the dividend until the remainder is less than the divisor. Although these methods are conceptually simple, they scale inefficiently for large numbers. Finally, division can also be performed by multiplying the dividend by the reciprocal of the divisor i.e. $\frac{u}{v} = u \cdot \frac{1}{v}$, which opens up the possibility for faster algorithms by computing the reciprocal instead of a division [20].

However, the GNU Multiple Precision Arithmetic Library (GMP) uses a hybrid division approach, where different algorithms are used depending on the size of the input [1]. When the divisor fits within a single machine-word – or limb in GMP terminology – the reciprocal approach is used. Otherwise, it resorts to a specialized long division method, namely Knuth's algorithm [20]. This algorithm is inherently sequential, finding one digit at each iteration, resulting in linear depth, making it unfit for parallel execution. In contrast, efficient parallel division usually relies on Newton's method to compute the reciprocal of the divisor, by solving for $f(x) = 1/x - v = 0$, where v is the divisor, and $x = \frac{1}{v}$ its reciprocal. This is achieved by the Newton iteration:

$$x_{(i+1)} = x_{(i)} - \frac{f(x_{(i)})}{f'(x_{(i)})} = x_{(i)} + x_{(i)} \left(1 - v \cdot x_{(i)}\right) \tag{1}$$

Newton's method uses a series of iterative steps to refine the precision of the reciprocal using multiplication. Even though these iterations are also inherently sequential, each iteration roughly doubles the precision, implying logarithmic depth, which is much more suitable for parallel architectures such as GPGPUs. However, Newton's method in its general form requires working in a related domain where the reciprocal exists. Working in another domain is not always desirable and can lead to a library structure where arithmetic domains are interdependent. Additionally, internal floating point representation can lead to potential loss of precision and overhead when converting between the domains.

To address said issue, this thesis works upon the concept of a whole shifted inverse, proposed by Watt in [33]. This approach enables the computation of an adapted reciprocal that remains entirely within a single domain, provided a suitable shift operation exists, such as for integers or polynomials. The intuition is to use the shift operation to scale the input to such a degree that captures all the necessary information from the fractional counterpart. Computations are then carried out at this higher scale, and a corresponding inverse shift is applied at the end to restore the result to its original magnitude, avoiding the potential loss of information. This strategy is explained further in detail in Section 6.

4.1 Quotient and Remainder

Whenever the dividend does not exactly divide the divisor, the result ends up as a fraction. However, as we operate exclusively in the integer domain, we adopt

the notion of a quotient and remainder defined as follows:

$$\begin{aligned} u \text{ quo } v &= \lfloor u/v \rfloor \\ u \text{ rem } v &= u - v \cdot \lfloor u/v \rfloor \end{aligned} \tag{2}$$

As previously mentioned, the quotient can be computed using a shift, and the whole shifted inverse. The two operations are defined as follows:

Definition 2. *(Whole shift and shifted inverse in $\mathbb{Z}$)*

Let $B > 1$ be an integer base. For integers n, u, and $v \neq 0$, with $n \geq 0$, the base-B whole n-shift of u and the base-B n-shifted inverse of v are defined as

$$\mathit{shift}_{n,B}(u) = \lfloor uB^n \rfloor \qquad\qquad \mathit{shinv}_{n,B}(v) = \left\lfloor \frac{B^n}{v} \right\rfloor$$

When B is clear from context, we write $\mathit{shift}_n(u)$ and $\mathit{shinv}_n(v)$.

When $n \geq 0$ $\text{shift}_{n,B}(u)$ corresponds to integer multiplication $u \times B^n$. When $n < 0$ it is instead a specialized quotient operation, with u as dividend and B^n as divisor. Using our multi-precision integer representation, this is equivalent to an arithmetic shift, e.g. $\text{shift}_1([1,2,3]) = [0,1,2]$ and $\text{shift}_{-1}([1,2,3]) = [2,3,0]$. In contrast, the whole shifted inverse $\text{shinv}_n(v)$ corresponds to a specialized reciprocal. In other words, it is simply a reciprocal that has been shifted into our domain e.g. $\text{shinv}_3(8) = \text{shift}_3(0.125) = 125$. These operations enable us to express the quotient of two integers as presented by the Theorem 1:

Theorem 1. *(Quotient by whole shifted inverse in $\mathbb{Z}$)*

Given two positive integers u and v, with $u \leq B^h$,

$$u \mathbin{\mathit{quo}} v = \mathit{shift}_{-h}(u \cdot \mathit{shinv}_h\, v) + \delta, \quad \delta \in \{0, 1\}.$$

Thus by multiplying u by the whole shifted inverse of v, and then applying a reverse shift, we obtain a result that has at most an error; δ at most 1 unit off from the correct result, as proven by Watt [33]. Additionally, the outcomes of $\delta = \{0, 1\}$ occur with equal frequency.

Example: Let $u = 22$, $v = 11$, $h = 3$ and $B = 10$. Then the whole shifted inverse of v in base B is given by:

$$\text{shinv}_{10}\,(11) = \left\lfloor \frac{100}{11} \right\rfloor = 9$$

Next, we multiply u with the shifted inverse, and then shift the result by $-h$:

$$\left\lfloor \frac{22 \cdot 9}{100} \right\rfloor = 1$$

However, since the correct result is $\lfloor \frac{22}{11} \rfloor = 2$, the approximation is off by one.

Luckily, adjusting for δ is quite trivial. We first compute an approximate quotient $q = \text{shift}_{-h}(u \cdot \text{shinv}(v, h, B))$ along with the corresponding remainder $r = u - v \cdot q$. Then if $r \geq v$, it indicates that we underestimated the initial quotient and need to add $\delta = 1$. The full procedure for computing the division of two big integers using the shifted inverse can be seen in Algorithm 1.

Algorithm 1: Div(u, v, m, B) in $\mathbb{Z}$

Input: $u,\ v \in \mathbb{Z}_+^m,\ m,\ B \in \mathbb{Z}_+$

Output: (q, r) where $q = u \operatorname{quo} v$, $r = u \operatorname{rem} v$

Uses: PREC, to compute precision
SHIFT, for shifting the integer
SHINV, the shifted inverse (Algorithm 6.5)

```
1: function Div(u, v, m, B)
2:     h ←prec(u)
3:     q ← shift_{-h}(u · shinv(v, h, B))
4:     r ← u − v · q
5:     if r ≥ v then
6:         q ← q + 1
7:         r ← r − v
8:     return (q, r)
```

4.2 Parameterization over Multiplication

The efficiency of computing the reciprocal via Newton's iteration (shown in Equation 1), is fundamentally tied to the cost of the underlying multiplication. Although each iteration involves both addition, subtraction, and multiplication, it is the multiplication that asymptotically dominates the performance, making it the most critical operation to optimize.

The best known theoretic upper bound for multiplication is $O(n \log n)$, which is believed to be tight [15]. In contrast, classical multiplication scales poorly with a much greater asymptotic complexity of $O(n^2)$. However, it is typically preferred for smaller input sizes due to very limited overhead. For intermediate-sized values, asymptotically faster methods such as Karatsuba ($O(n^{\log_2 3})$) [19] or Toom-Cook ($O(n^{\log_3 5})$) [11] are often used. For large inputs, more advanced techniques such as Schönhage-Strassen or FFT-based methods become more efficient [31].

Since Newton's method approximately doubles the number of correct bits at each iteration, computing the reciprocal requires $O(log(n))$ iterations. Thus, if multiplication has complexity $O(M(n))$, then Newton's method yields an overall complexity of $O(M(n) \cdot \log n)$. Although this can be greatly improved by exploiting the fact that, since the precision doubles at each iteration, most multiplications only need to be performed on a small part of the input size, which can reduce the overall asymptotic work to $O(M(n))$. This concept of shorter iterates is elaborated further in Section 6.3.

5 Supporting Arithmetic

In order to discuss a parallel algorithm for division we require equally performant implementations of the supporting arithmetic operations. As our division relies on calculating the reciprocal of the divisor (as shown in Equation 1), we are required to define both addition, subtraction, and multiplication. In this section, we will look at how to adapt these supporting arithmetic operations to work on the multiple precision integers as per Definition 1. In particular, we will discuss how these can be implemented efficiently and adapted to a parallel setting.

5.1 Addition

Possibly the most basic arithmetic operation, the grade-school algorithm for performing addition requires each digit of the two numbers to be added together individually, propagating the carries throughout. This algorithm scales poorly in parallel systems as the asymptotical time complexity of the longest sequential dependency (or span) is $O(n)$ i.e. linear with respect to the number of bits n, with no degree of parallelism possible.

In practice, modern ALUs use a more efficient version of this operation which works for binary digits by utilizing bitwise operations instead. More specifically, for two unsigned integers it's the case that bitwise XOR corresponds to an addition without the carries, and bitwise XOR left-shifted by 1 corresponds to the carries between them. Oancea and Watt describe an algorithm for adapting this operation to the big integers as per Definition 1, which can be seen in Figure 8. This algorithm adds together all indices correspondingly, while propagating the carries separately. Computing the carries is a case of a true dependency, as each element is dependent on the computation of the element immediately preceding it, resulting in an algorithm inherently linear with respect to the number of bits.

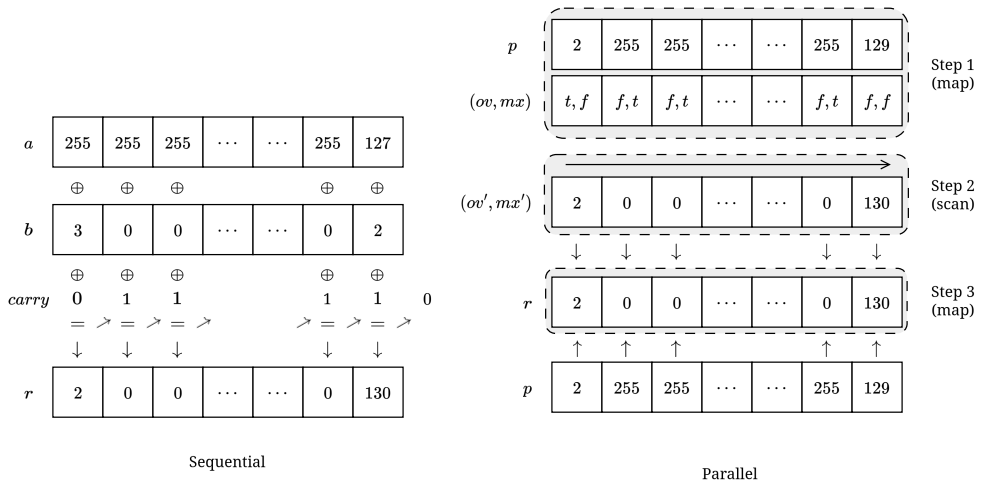


Figure 8: Sequential and parallel procedures for addition as defined in [30].

Contrary to perhaps initial intuition this problem is shown to have $\log n$ span by [9, 30], meaning that the depth is sublinear. The secret lies in recognizing that only the carry propagation itself requires a sequential dependency between elements, and that this can be modeled in terms of a scan [30]. More specifically, it

has been shown by [21] that each overflow occurs *if and only if* the current addition overflows, or if the previous addition overflowed and the current addition results in the maximally representable element. For two input tuples, corresponding to the boolean value of overflow and maximal value for two elements, we can define the operator capturing this logic $\otimes$ like so:

$$x \otimes y = (ov_x, mx_x) \otimes (ov_x, mx_x) = (ov_x \wedge mx_y \vee ov_y,\ mx_x \wedge mx_y)$$

This operator has been shown by [21] to be associative and have the neutral element (`False`, `True`). With this in mind, the addition of two integers can be modeled in terms of SOACs (map and scan), with only three simple steps, as shown by [30]. The algorithm works as follows:

1. Compute the element-wise addition at each index i to form a partial result p. Additionally for each element, record the following: (i) whether the addition results in an overflow i.e. there is a carry, and (ii) if the result of the addition is the maximum element of the base B.
2. Combine the overflow-highest pairs over all indices.
3. For each element add together the partial result and the carry.

The resulting algorithm can be seen in Figure 8, where the three steps are outlined. In Step 1, the element-wise sums without the carries are calculated alongside the wrap-around semantics needed to compute the carries. In terms of a more functional approach we say that this step maps over the inputs with an operator $\oplus$ such that:

$$x \oplus y = (x + y,\ (x + y < x,\ x + y == B - 1))$$

Step 2 is modeled by a scan as detailed above, and finally in Step 3 the partial result is simply added with the carries via the $\odot$ operator. The resulting algorithm can thus be defined in terms of the operators $\oplus$, $\otimes$, $\odot$ and the neutral element `e` like so

$$\begin{aligned} x \oplus y &:= \Big((x + y,\ (x + y < x,\ x + y == B - 1)\Big) \\ x \otimes y &:= (ov_x, mx_x) \otimes (ov_x, mx_x) = (ov_x \wedge mx_y \vee ov_y,\ mx_x \wedge mx_y) \\ x \odot y &:= x + (y \,\&\, 1) \\ \texttt{e} &:= (\texttt{False},\ \texttt{True}) \end{aligned} \tag{3}$$

The operators can then be used to implement the algorithm using simply two maps and a scan for two big integers $u, v \in \mathbb{N}^m$:

```
1 (r, a) = map2 ⊕ u v
2 c = scan_exc ⊗ e a
3 w = map2 ⊙ r c
```

All operations as part of the parallel SOACs are constant time operations i.e. $O(1)$. Hence it is trivial to deduce that the algorithm runs in $O(n)$ work and $O(\log n)$ span with respect to the number of elements in the big integers.

5.2 Subtraction

Due to our choice of big integer representation as per Definition 1 signed integers are not possible to represent. However, it is still relevant to define subtraction on these, as it is a required operation in order to calculate the reciprocal of the divisor. More specifically, it is required as a part of Newton's method, as seen in Equation 1. Thus, if it at any point is possible for the signage to change throughout the algorithm due to subtraction, we will instead consider this separately, as it is trivial to compute whether the subtraction has underflowed or not.

Subtraction is exceedingly similar to the case of addition, resulting in a true dependency between carries as well. In fact, we are able to define an algorithm for subtraction on big integers based on the one for addition, by only changing the operators used during Step 1 and 3. This results in the following algorithm:

$$\begin{aligned} x \oplus y &:= \Big(x - y,\ (x - y > x,\ (x - y == 0) \ll 1)\Big) \\ x \otimes y &:= (ov_x, mx_x) \otimes (ov_x, mx_x) = (ov_x \wedge mx_y \vee ov_y,\ mx_x \wedge mx_y) \\ x \odot y &:= x - (y + 1) \\ \texttt{e} &:= (\texttt{False}, \texttt{True}) \end{aligned} \tag{4}$$

5.3 Classical Multiplication

The last remaining arithmetic operation needed for Newton's method is multiplication, which is asymptotically more expensive than addition and subtraction. The classical algorithm (long multiplication) requires multiplying all digits of the multiplicand with all digits of the multiplier individually, followed by adding all of the intermediate results together to get the final product. The result is an algorithm that requires asymptotically quadratic work $O(n^2)$, which is suboptimal (an improvement will be discussed in Section 5.4), but is still performantly competitive in a wide variety of inputs. Classically multiplying two integers $u, v \in \mathbb{N}$ with m digits and base B has previously been formulated by [10] as follows:

$$u \cdot v = \sum_{k=0}^{m-1} \left(\sum_{\substack{i+j=k \\ 0 \leq i,j,k < m}} u_i \cdot v_j \right) \cdot B^k \tag{5}$$

From Equation 5 we denote the following observations:

- The intermediate products w_k can have double the size of the bases of u_i and v_i i.e. $2B$.
- All products across the m convolution are unique, and the number of terms needed to calculate w_k scales linearly.

Since there are m convolutions of sizes 0 to $m - 1$, there are a total of $m^2/2$ products to be computed. In order to parallelize this, we present two load-balancing schemes for which to delegate the parallel work between threads. One option is to let the k'th thread, denoted t_k compute the k'th convolution w_k, then

if the number of convolutions exceeds the number of threads, simply loop around. This scheme performs poorly as m increases because the amount of work increases linearly, resulting in an uneven distribution of workload between threads.

An alternative "embarrassingly parallel" load-balancing is proposed by [30]. This scheme comes from the realization that the number of products as part of computing w_k, in addition to those from computing w_{m-k-1}, always results in $m+2$ products, regardless of k. Thus, if the same thread is used to compute convolutions k and $m-k-1$, the work distributed between threads stays constant. As a natural extension of this, one can increase the number of convolutions calculated by each thread, denoted the *sequentialization factor* Q. As long as this factor is a multiple of two, the scheme can extend to support arbitrary sizes of integers, even though the number of threads stays constant.

Because the convolutions can be handled in parallel by independent threads, and only ever require $O(m)$ space, the schema allows these to be computed in the lowest thread-level memory (in CUDA this corresponds to register memory). Additionally, since all products across all convolutions are unique, the amount of computation as part of the proposed scheme is optimal, as all products are computed exactly once. This proposed load-balancing scheme is shown in Figure 9, where the sequentialization factor is $Q = 2$ meaning that every thread computes two convolutions each.

$$
\begin{aligned}
w_0 &= u_0 \cdot v_0 && := t_0 \\
w_1 &= u_0 \cdot v_1 + u_1 \cdot v_0 && := t_1 \\
w_2 &= u_0 \cdot v_2 + u_1 \cdot v_1 + u_2 \cdot v_0 && := t_2 \\
&\vdots && \vdots \\
w_{m-3} &= u_0 \cdot v_{m-3} + \ldots + u_{m-3} \cdot v_0 && := t_2 \\
w_{m-2} &= u_0 \cdot v_{m-2} + \ldots + u_{m-3} \cdot v_1 + u_{m-2} \cdot v_0 && := t_1 \\
w_{m-1} &= u_0 \cdot v_{m-1} + \ldots + u_{m-3} \cdot v_2 + u_{m-2} \cdot v_1 + u_{m-1} \cdot v_0 && := t_0
\end{aligned}
$$

Figure 9: Embarrassingly parallel load-balancing schema as proposed by [30] where each thread computes m products.

Since the intermediate products can double the size of the integer, the accumulation of the partial results as computed by the convolutions, need to be stored in buffers of double the size of the base B. Additionally, the process of summing multiple double-sized values can itself lead to overflows if the cumulative sum exceeds the capacity of the accumulator. Thus, once a partial sum and it's corresponding carries have been computed we are left with the following parts:

1. A *low part*: These are the Q elements of the accumulated sum that correspond to a single digit in base B (i.e., the sum modulo B, or the least significant B digits of the result).

2. A *high part*: These B bits represents the more significant portion of the accumulated sum (i.e., the most significant digits of the result).

3. An *aggregated carry*: These final B digits correspond to any overflow of the high part of the aggregated result as generated during the initial accumulation of products.

Finally the the three different parts can be published to arrays corresponding to the low and high bits of the result (denoted `L` and `H` respectively). The method for this is defined as follows; The low part keeps the original indices and are published to `L`. The high part is published to `H`, at the original indices shifted by a factor of Q. Lastly the carry is published to `H` as well, at the original indices shifted by a factor of $Q + 1$. The result are the two arrays `L` and `H` which can finally be added together using the algorithm described in Section 5.1 in order to complete the product of the two multi-precision integers. This method of which to publish the low part, high part and aggregated carry can be seen in Figure 10.

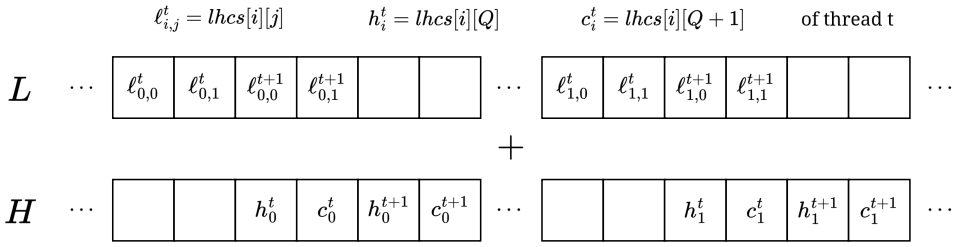


Figure 10: An example of publishing the low part, high part and aggregated carry for a thread when $Q = 2$. Finally the two arrays are added to form the product.

5.4 FFT Multiplication

The classical multiplication algorithm requires asymptotically quadratic work, which is suboptimal on larger integers as sub-quadratic algorithms exist such as the Karatsuba, Toom-Cook, or the Schönhage-Strassen algorithm, which utilizes a Fast Fourier Transform (FFT) [31]. In particular, FFT-based multiplication algorithms runs much faster on larger integers with a time complexity of $O(n \log n)$. These algorithms have been shown to be efficient [7, 30] and are actively used as part of the GMP library [1]. However in practice, performing multiple FFTs increase the runtime by a large constant factor, meaning that they are inefficient on integers below a certain size. For this reason it is common practice to define size thresholds for using the different algorithms, and then switch between them depending on this [1]. Tempted by the prospect of an asymptotic decrease in runtime we explore how this algorithm can be implemented on multiple precision integers.

$$\texttt{fft}(w_k) = \texttt{fft}\left(\sum_{\substack{i+j=k \\ 0 \leq i,j,k < m}} u_i \cdot v_j\right) = \texttt{fft}(u_k) \cdot \texttt{fft}(v_k)$$

$$w_k = \texttt{ifft}\Big(\texttt{fft}(u_k) \cdot \texttt{fft}(v_k)\Big)$$

The motivation behind using the FFT algorithm lies in the fact that multiplying two integers represented as polynomials, when converted to the domain of the discrete Fourier transform (DFT) can be performed using pointwise multiplication. The pointwise multiplications can then be performed in linear time (if the representation lies within a single machine word), making the algorithm upper bounded by the cost of computing the FFT and the inverse FFT (IFFT).

To multiply two multiple precision integers using the FFT algorithm without losing precision due to floating-point rounding errors, we first need to find suitable roots of unity in a finite field (a *Galois field*) rather than relying on complex-valued roots on the unit circle. Following the method by [30] we initially want to find primes of the shape:

$$p = k \cdot 2^n + 1$$

Better known as *Proth primes*, these accept 2^n distinct roots of unity, for a large enough n. Using Fermat's little theorem we want to find values of a not divisible by p i.e. values that satisfy the following:

$$\forall a,\ a^{k \cdot 2^n} \equiv 1 \pmod{p}$$

We denote $g = a^k$, and then following from the equation above we have that $g^{2^n} = 1 \mod p$. Then to find a g that is a 2^n'th root of unity, we iterate through all elements $a \in \mathbb{Z}_p$ and choose one such that $q < 2^n$ and $g^q \neq 1$. Finally we are able to make an M'th root of unity called ω such that M is a power of 2 and $M < 2^n$ i.e.

$$\omega = g^{2^n/M} = g^{n - \log_2 M}$$

Oancea and Watt have already found that

$$p_{32} = 3221225473 \quad \text{and} \quad p_{64} = 4179340454199820289,$$

are suitable prime fields for implementing the FFT algorithm, when individual digits are represented by 32-bit and 64-bit words respectively [30]. Thus for the remainder of this paper we shall consider these values as well.

The multiplication algorithm itself an adaptation of [32] and is conceptually similar to the classical algorithm described in Section 5.3, with some additional restrictions due to the FFT. Firstly, M has to be a power of 2, which can be ensured by sufficient padding. Secondly, the sequentialization factor Q must be equal or greater than 2, and has to evenly divide M. Which thus restricts the number of threads to be $\frac{M}{Q}$ per **CUDA** block. With these restrictions in place, the algorithm has the following steps

1. First the big integers are converted to the DFT domain using the prime fields as described above.
2. Then the algorithm performs an pointwise multiplication.
3. The algorithm then converts back into the regular domain using the inverse FFT algorithm.

4. The word size of the result is changed to one less. And the per-thread results are published to shared memory following a method similar to the one described in Figure 10.

5. Finally the low and high parts can be added together to compute the output.

The final result is an asymptotically faster but more restrictive multiplication algorithm, that computes the product of two variable sized integers of $(\texttt{word size} - 1) \cdot M$ bits.

6 The Whole Shifted Inverse

Equation 1 illustrates the use of Newton's iteration for computing the reciprocal of a number. However, since our goal is to compute the shifted inverse i.e. $\text{shinv}_{n,B}(v) = \left\lfloor \frac{B^n}{v} \right\rfloor$, it requires some adaptation. Instead of solving the equation $f(x) = 1/x - v = 0$, we consider $f(x) = B^n/x - v = 0$. Applying Newton's method to this function results in the following Newton's iteration:

$$x_{(i+1)} = x_{(i)} - \frac{f(x_{(i)})}{f'(x_{(i)})} = x_{(i)} + x_{(i)} \left(1 - \frac{v}{B^n} \cdot x_{(i)}\right) \tag{6}$$

Solving this iteration requires division, which is what we ultimately are trying to implement, therefore this is not a viable approach. Luckily, this division is close to being a shift, which allows us to use a specialized version of Newton's iteration:

$$\begin{aligned} w_{(i+1)} &= w_{(i)} + \text{shift}_{-h}(\text{shift}_h(w_{(i)}) - v w_{(i)}^2), \quad w_{(i)} \in \mathbb{Z} \\ &= w_{(i)} + \left\lfloor w_{(i)}(B^n - v w_{(i)}) B^{-n}) \right\rfloor \end{aligned} \tag{7}$$

As this is not the usual Newton's iteration, but instead discretized to integers, its correctness and convergence properties need to be reestablished. Watt addresses this in [33] through a series of theorems, some of which we present in the following subsections.

6.1 Convergence

In the subsequent sections, we will refer to $S_{\mathbb{Z}}$ as the integer-valued counterpart of Newton's iteration:

$$S_{\mathbb{Z}} : \mathbb{Z} \to \mathbb{Z} = w \mapsto w + \left\lfloor w \left(1 - \frac{v}{u} w\right) \right\rfloor, \quad \text{where } 1 < v < u \tag{8}$$

Note that this corresponds to an integer approximation of Equation 6, where B^n is generalized to an arbitrary value of u. Through detailed analysis Watt shows the fixed points and convergence properties of $S_{\mathbb{Z}}$, which culminates in the following two theorems:

Theorem 2. *($S_{\mathbb{Z}}$ Convergence)*

The sequence of iterates $S_{\mathbb{Z}}^{(i)}(w)$, $i \geq 1$, converges if and only if $w \in [0 \,..\, 2u/v]$. If the series converges, then it is to one of $\{0, 1, \lfloor u/v \rfloor - 1, \lfloor u/v \rfloor\}$. For $w \in [2 \,..\, u/v]$, $S_{\mathbb{Z}}^{(i)}(w)$ converges to $\lfloor u/v \rfloor - 1$ or $\lfloor u/v \rfloor$.

Theorem 3. *(Fast $S_{\mathbb{Z}}$ Convergence)*

If $w_{(0)} \in \left[(1 - \frac{1}{4})\frac{u}{v} \,..\, (1 + \frac{1}{4})\frac{u}{v}\right]$ and $\frac{u}{v} \geq 2$, then

$$S_{\mathbb{Z}}^{\lceil \log_2 \log_2 (u/v) \rceil}(w_{(0)}) \in \{\lfloor u/v \rfloor - 1, \lfloor u/v \rfloor\}.$$

Theorem 2 shows that $S_{\mathbb{Z}}$ always converges when $w \in [0 \,..\, 2u/v]$, and when it does, it is to one of the fixed points $\{0, 1, \lfloor u/v \rfloor - 1, \lfloor u/v \rfloor\}$. Additionally, if

$w \in [2\,..\,u/v]$ then we are guaranteed that it converges to either $\lfloor u/v \rfloor - 1$ or $\lfloor u/v \rfloor$.

Theorem 3 builds on top of Theorem 2, and shows that if the initial value $w_{(0)}$ is chosen within 25% of u/v, then it converges to one of the final two fixed points within $\lceil \log_2 \log_2(u/v) \rceil$ steps. Note that the theorems state convergence to either $\lfloor u/v \rfloor - 1$ or $\lfloor u/v \rfloor$, which might seem problematic, since we want to compute $\text{shinv}_{n,B}(v) = \lfloor \frac{B^n}{v} \rfloor$ as per Definition 2 with no error. Watt addresses this issue by increasing the intermediate results by a factor of B^g by using a shift, where g is the number of guard digits, followed by shifting the result back to its original magnitude in order to return the final value [33].

6.2 Initial Value Choice

Having a bound for w that yields fast convergence allows us to compute an initial choice, satisfying the conditions of Theorem 3:

Theorem 4. *(Initial Value Choice)*

Let $B \leq v < B^{k+1}$ *and* $2v \leq u = B^h$ *for* $B \geq 16$*, and let* $v = VB^{k-f} + R$ *with* $f, V, R \in \mathbb{Z}$*,* $B^f \leq V < B^{f+1}$*,* $0 \leq R < B^{k-f}$ *and* $f \geq \min(k, 2)$*.*

Then the choice $w_{(0)} = \left\lfloor \frac{B^{f+2}}{V} \right\rfloor B^{h-k-2}$ *gives*

$$S_{\mathbb{Z}}^{\lceil \log_2 \log_2(u/v) \rceil}(w_{(0)}) \in \left\{ \left\lfloor \frac{u}{v} \right\rfloor, \left\lfloor \frac{u}{v} \right\rfloor - 1 \right\}.$$

Theorem 4 states that an iteration starting point that ensures fast convergence can be efficiently computed by inverting a short prefix of v. Specifically, when $k \geq 2$, we have $B^2 \leq V < B^3$, which means that V is chosen as the three most significant digits of v.

Example: Let $v = 123123$, $h = 8$. Then $V = 123$, and the initial approximation $w_0 = \lfloor \frac{10000}{123} \rfloor \cdot 10 = 810$, which is very close to the exact value of the whole shifted inverse and well within the 25% accuracy required by Theorem 3:

$$\text{shinv}_8(123123) = \left\lfloor \frac{100000000}{123123} \right\rfloor = 812$$

6.3 Shorter iterates & divisor prefixes

A common strategy when dealing with iterative methods of multi-precision values is to start with low precision and increase it with each iteration, a method that Watt refers to as short iterates. The intuition is that because the precision of the approximated shifted inverse starts low and roughly doubles with each step, the same principle can be applied to the size of w, since only the leading digits significantly influence the computation. This reasoning also extends to the intermediate values of v. When v is large compared to w, only the leading digits will have a significant effect on the iteration, and thus divisor prefixes can be applied. However, this optimization introduces a potential risk, which we discuss in Section 7.3. Applying both strategies allows us to reduce the cost

substantially, as most arithmetic operations operate on values that are a fraction of the full size. The exact complexity is described in Section 6.6. Incorporating these optimizations into Equation 7, the resulting iteration scheme becomes:

$$\begin{aligned} w_{(i+1)} &= S_{\mathbb{Z}}\left(k + 2\ell_{(i)} - s_{(i)},\ \text{shift}_{-s_{(i)}} v,\ \text{shift}_{\ell_{(i)}} w_{(i)}\right) \\ &= w_{(i)}B^{\ell} + \left\lfloor w_{(i)}\left(B^{\ell} - B^{-k+s_{(i)}}\lfloor vB^{-s_{(i)}}\rfloor w_{(i)}\right)\right\rfloor \end{aligned} \tag{9}$$

where

$$\ell_{(i+1)} = 2\ell_{(i)} - 1, \qquad s_{(i)} = \max(0,\ k - 2\ell_{(i)} + 1)$$

6.4 Close Products

Each iteration of the integer Newton's method (Equation 7) involves computing the difference $B^h - vw$. However, vw is often very close to B^h, which allows us to optimize this computation by avoiding redundant work. Watt refers to this concept as *close products*. Essentially, what this means is that when $|B^h - vw| \leq B^e$ for some $e < h$, the result of $B^n - vw$ contains significantly fewer than h base B digits, meaning most of the digits in vw are predetermined. When $B^h > vw$ the difference is positive, and the upper digits of vw are all $B - 1$. When $B^h < vw$ the difference is negative, and the upper digits of vw are all 0. For example, in base 10:

- If $B^h < vw$ all upper digits of vw are 9 (i.e. $B - 1$):

  $10000000 - 9999667 = 333$

- If $B^h > vw$ all upper digits of vw are 0:

  $10000000 - 10000333 = -333$

In both cases, only the lower e digits are needed to compute the result:

$$10000000 - 9999667 = 333 \iff 1000 - 667 = 333$$

$$10000000 - 10000333 = -333 \iff 1000 - 1333 = -333$$

Since most digits of vw are predetermined, it is enough to compute only the lower e digits as:

$$vw \text{ rem } B^e \iff (v \text{ rem } B^e)(w \text{ rem } B^e)$$

This optimization can significantly improve efficiency by avoiding full-width multiplication and subtraction. However, the optimization is most noticeable when "shorter iterates" and "divisor prefixes" are not applied, as they already provide a part of this benefit.

6.5 Original Algorithm

The algorithm presented by Watt, which encapsulates the findings of the previous sections, is outlined below. We divide it into 3 main parts: special case handling, initial approximation and iterative refinement.

Special Case Handling (lines 2-10) This ensures that the easy cases are handled ($B < v \leq B^h/2$), and that the base is sufficiently large ($B >= 16$), such that the prerequisites for the initial value choice in Theorem 4 are met.

Initial Approximation (lines 11-14) This computes the initial value as described in Section 6.2, using $f = min(k, 2)$. If sufficiently many digits are correct, this value is shifted to the appropriate magnitude and then returned. Since the size of the divisor prefix used to approximate the initial value is bounded, this short inversion at line 11 can be performed in constant time. Note that the division at line 11 is semantically the same as computing $\lfloor B^{2l}/V \rfloor = B^{2l}$ quo V. By subtracting B^{2l} by V, the dividend goes from 5 to 4 digits, meaning that the division can be performed on a smaller native word size.

Iterative Refinement (Refine, Step, PowDiff) If the initial approximation is not sufficiently accurate, it is refined using one of three refinement routines described below:

- `Refine1` corresponds to the unoptimized integer version of Newton's iteration (Equation 7). It starts by shifting w to the desired final length, while using the entire divisor v to compute each iteration.
- `Refine2` improves on `Refine1` by taking upon the shorter iterates strategy, proposed in Section 6.3, to only work with the lowest necessary precision of w, and thus decreasing the size needed for the intermediate arithmetic operations.
- Finally, `Refine3` is the most optimized and the one used in practice. It employs both shorter iterates and divisor prefixes and corresponds to the optimized iteration scheme from 9.

All refinement methods call the `Step` function, which performs a single Newton iteration. `Step` invokes `PowDiff`, which computes $B^h - v \cdot w$ using the close product strategy described in Section 6.4 for improved efficiency. Lastly, `Step` shifts the intermediate result according to the chosen refinement strategy.

Algorithm 1: SHINV$(v,\ h,\ B)$ in $\mathbb{Z}$

Input: $v,\ h,\ B \in \mathbb{Z}_{>0},\ B^k \le v < B^{k+1}$
Output: $\text{SHINV}_h v$ ▷ All shifts are with respect to B
Uses: MULT, a multiplication method
POWDIFF, to compute $B^h - v \cdot w$ (Algorithm 2)
REFINE

Function `Shinv`$(v,\ h,\ B)$**:**
  ▷ Group digits if base is small
  if $B < 16$ **then**
    $p \leftarrow \min(6 - B, 2)$
    return $\text{SHIFT}_{h \text{ rem } p - p}\text{SHINV}(v, h \text{ quo } p + 1, B^p)$
  ▷ Special cases guarantee $B < v \le B^h/2$
  if $v < B$ **then return** B^h quo v ▷ Divide by 1 digit
  if $v > B^h$ **then return** 0
  if $2v > B^h$ **then return** 1
  if $v = B^k$ **then return** B^{h-k}
  ▷ Form initial approximation, returning it if sufficient
  $\ell \leftarrow \min(k, 2)$
  $V \leftarrow \sum_{i=0}^{\ell} v_{k-\ell+i} \cdot B^i$
  $w \leftarrow \left(B^{2\ell} - V\right)$ quo $V + 1$ ▷ Divide 4 digits by 3 digits
  if $h - k \le \ell$ **then return** $\text{SHIFT}_{h-k-\ell}(w)$
  ▷ Refine iteratively using one of the methods below
  return $\text{REFINE}(v, h, k, w, \ell)$

Function `Refine1`$(v,\ h,\ k,\ w,\ \ell)$**:**
  $g \leftarrow 1$
  $h \leftarrow h + g$
  $w \leftarrow \text{SHIFT}_{h-k-l}(w)$ ▷ Scale initial value to full length
  while $h - k > \ell$ **do**
    $w \leftarrow \text{STEP}(h,\ v,\ w,\ 0,\ \ell,\ 0)$
    $\ell \leftarrow \min(2\ell - 1, h - k)$ ▷ Number of correct digits
  return $\text{SHIFT}_{-g}(w)$

Function `Refine2`$(v,\ h,\ k,\ w,\ \ell)$**:**
  $g \leftarrow 2$
  while $h - k > \ell$ **do**
    $m \leftarrow \min(h - k + 1 - \ell,\ \ell)$
    $w \leftarrow \text{SHIFT}_{-1}(\text{STEP}(k + \ell + m + g,\ v,\ w,\ m,\ \ell,\ g))$
    $\ell \leftarrow \ell + m - 1$
  return $\text{SHIFT}_{-g}(w)$

Function `Refine3`$(v,\ h,\ k,\ w,\ \ell)$**:**
  $g \leftarrow 2$ ▷ Guard digits
  $w \leftarrow \text{SHIFT}_g w$
  while $h - k > \ell$ **do**
    $m \leftarrow \min(h - k + 1 - \ell,\ \ell)$
    $s \leftarrow \max(0,\ k - 2\ell + 1 - g)$ ▷ How to scale v
    $w \leftarrow \text{SHIFT}_{-1}(\text{STEP}(k + \ell + m - s + g,\ \text{SHIFT}_{-s}(v),\ w,\ m,\ \ell,\ g))$
    $\ell \leftarrow \ell + m - 1$
  return $\text{SHIFT}_{-g}(w)$

Function `Step`$(h,\ v,\ w,\ m,\ \ell,\ g)$**:**
  $\text{SHIFT}_m(w) + \text{SHIFT}_{2m-h}(\text{MULT}(w, \text{POWDIFF}(v, w, h - m, \ell - g, B)))$

Algorithm 2: PowDiff(v, w, h, ℓ, B) in $\mathbb{Z}$

Input: $v,\ w,\ h,\ \ell,\ B \in \mathbb{Z}_{>0}$ such that $\text{prec}|w - \text{SHINV}_h v| \leq \text{prec}(w) - \ell$

Output: $B^h - v \cdot w$

Uses: $\text{MULT}(a,\ b) = a \cdot b$

$\text{MULTMOD}(a,\ b,\ d,\ B) = (a \cdot b) \bmod B^d$

```
1: Function PowDiff(v, w, h, ℓ, B):
2:     L ← prec_B v + prec_B w − ℓ + 1;
3:     if v = 0 ∨ w = 0 ∨ L ≥ h then return B^h − MULT(v, w);
4:     else
5:         P ← MULTMOD(v, w, L, B);
6:         if P = 0 then return 0;
7:         else if P_{L−1} = 0 then return −P;
8:         else return B^L − P;
```

6.6 Complexity Analysis

As multiplication is asymptotically dominating the runtime of the algorithm, we naturally base the complexity analysis on the number of multiplications performed. Each iteration performs two multiplications; one inside Step and one as part of PowDiff. PowDiff returns a big integer of length $\min(n, L)$, which is used by the multiplication inside Step. Size L is returned when the close products strategy is applied, and is given by

$$L = \text{prec}_B v + \text{prec}_B w - \ell + 1,$$

where n is particular to the chosen refinement method. In the following we compare the time complexity of the algorithm using each of the three refinement routines. We denote $O(M(N))$ as the time complexity of a single multiplication and we ignore additive constants.

Refine1 This function starts by shifting w to its final length of $h - k$ digits, while the full v of $k + 1$ digits is used, meaning that $L \approx h - 2^i$. We also have $n = h$. The multiplication inside Step is then of length $h - k$ and $h - 2^i$. The multiplication inside PowDiff is of length $\min(h-k, h-2^i)$ and $\min(k, h-2^i)$. Both multiplications are upper bounded by $O(h)$ and since we perform $O(\log(h - k))$ iterations the time complexity becomes:

$$\begin{aligned} T(h,k) &= O\left(\sum_{i=0}^{\log(h-k)} Step + \sum_{i=0}^{\log(h-k)} PowDiff\right) \\ &= O\left(\sum_{i=0}^{\log(h-k)} M(h) + \sum_{i=0}^{\log(h-k)} M(h)\right) \\ &= O(\log(h-k) \cdot M(h)) \end{aligned}$$

Refine2 This function employs the shorter prefixes strategy, reducing the length of w to 2^i, while v remains at $k+1$ digits. In addition, we have $L \approx k$ and $n \approx k+2^i$. Consequently, the multiplication in Step is of length 2^i and k, while the multiplication in PowDiff is of length $\min(2^i, k)$ and k. The total time complexity thus becomes:

$$\begin{aligned} T(h,k) &= O\left(\sum_{i=0}^{\log(h-k)} Step + \sum_{i=0}^{\log(h-k)} PowDiff\right) \\ &= O\left(\sum_{i=0}^{\log(h-k)} M(k) + \sum_{i=0}^{\log(h-k)} M(\max(2^i, k))\right) \\ &= O\left(\log(h-k)\cdot M(k) + \sum_{i=\log(k)}^{\log(h-k)} M(2^i)\right) \\ &= O(\log(h-k)\cdot M(k) + M(h-k)) \end{aligned}$$

The final derivation assumes that $M(n)$ has at least polynomial growth, that is $M(n) = C \cdot n^\epsilon$ for some $\epsilon > 0$, the sum $\sum_{i=0}^{\log(n)} M(2^i)$ is dominated by its largest term, and thus $O(M(n))$.[1]

Refine3 The final refinement routine uses both shorter iterates and divisor prefixes strategies, which reduces the size of w, v, L and n to 2^i digits. The multiplication inside Step is of length 2^i and 2^i. The multiplication in PowDiff is of length 2^i and $\min(2^i, k)$. This results in the total time complexity:

$$\begin{aligned} T(h,k) &= O\left(\sum_{i=0}^{\log(h-k)} Step + \sum_{i=0}^{\log(h-k)} PowDiff\right) \\ &= O\left(\sum_{i=0}^{\log(h-k)} M(2^i) + \sum_{i=0}^{\log(h-k)} M(2^i)\right) \\ &= O(M(h-k)) \end{aligned}$$

Again, the final derivation assumes that $M(n)$ has at least polynomial growth.

Both `Refine1` and `Refine2` exhibit complexities in the form $M(N)$ multiplied by a logarithmic factor, whereas `Refine3` is linear in relation to $M(N)$. i.e., for the theoretical $M(N) \in O(n \log n)$, then the overall time complexity of whole shifted inverse using `Refine3` is $O(n \log n)$.

[1] $\sum_{i=0}^{\log n} M(2^i) \leq \sum_{i=0}^{\log n} C(2^i)^\epsilon = C \sum_{i=0}^{\log n} 2^{i\epsilon} = O(2^{\epsilon \log n}) = O(n^\epsilon) = O(M(n))$

7 Algorithmic Revisions

Watt's algorithm provides an efficient method for computing the whole shifted inverse, but certain edge cases require additional care to ensure correctness in all scenarios. In this section, we introduce a series of refinements that address these edge cases without altering the core structure of the algorithm. Specifically, we tackle the handling of negative intermediate values, limitations of the initial approximation, and subtle overestimation problems that arises when using divisor prefixes. These refinements culminate in a revised algorithm that remains asymptotically efficient while offering stronger guarantees of correctness. We then implement this revised algorithm in both CUDA and Futhark.

7.1 Handling Negative Values

When $B^h < vw$, `PowDiff` returns a negative value. However, the multi-precision integer representation presented in Section 3.4 does not support negative values. Even though it could be extended to include a sign bit, negative values are rarely used in the whole shifted inverse computation, and thus not necessarily worth the extra memory usage. Instead, we modify `PowDiff` to always return the absolute value along with a separate boolean indicating the sign. The `Step` function needs to be updated accordingly, as when `PowDiff` is negative, the second term of `Step` will be negative as well, requiring subtraction rather than addition. The decision between addition and subtraction is made based on the sign boolean returned by `PowDiff`.

A more subtle yet important distinction between handling positive and negative values lies in the behavior of shift. We want shift to mimic integer base-B division, which rounds down to the nearest integer. When shift is applied to positive values, it naturally rounds the number down towards 0, which is what we expect i.e.

$$\lfloor 11/10 \rfloor = \lfloor 1.1 \rfloor = 1 \iff \text{SHIFT}_{-1}(11) = 1$$

However, for negative values, we also want shift to behave the same as for negative division, which rounds down towards negative infinity. If we naively perform shift on the absolute value, and then apply the sign afterward, we risk rounding in the wrong direction.

$$\lfloor -11/10 \rfloor = \lfloor -1.1 \rfloor = -2 \not\iff -\text{SHIFT}_{-1}(11) = -1$$

Thus, to ensure rounding toward negative infinity for negative values, whenever the base-B division does not evenly divide the dividend, we want increment it by 1 before applying the sign. Determining whether the division is exact is straightforward, and only requires examining the n least significant digits. If all n digits are 0, the division is exact. Otherwise, it is division with remainder and we add 1 to the shifted result. The resulting procedure can be seen in Algorithm 3.

7.2 Limitations of the Initial Approximation

As described in Section 6.5, Watt uses an inverted short prefix of v as the starting point of the algorithm. The algorithm uses the 3 most significant digits of v as

Algorithm 3: Shift Applied to Negative Numbers

1: $y \leftarrow \text{SHIFT}_n(x)$;
2: **if** *any of the n least significant digits of x are nonzero* **then**
3: $\quad y \leftarrow y + 1$;

the short prefix. This initial approximation is assumed to have $min(k, 2)$ correct digits, and thus it is early returned when the divisor v, has the same or one fewer digits than the dividend u. Unfortunately, we are not always guaranteed that the most significant digits of this initial approximation are correct, an issue that does not seem to be addressed in [33].

Suppose we want to evaluate the whole shifted inverse $\text{shinv}_{h,B}(v)$ in decimal base, with $h = 5$, $B = 10$ and $v = 1119$. Based on Definition 2, this gives us $\text{shinv}_{5,10}(1119) = \lfloor \frac{100000}{1119} \rfloor = 89$. The initial approximation, as described in Theorem 4, is given by:

$$w_{(0)} = \left\lfloor \frac{B^{f+2}}{V} \right\rfloor B^{h-k-2} \quad \text{where} \quad V = \left\lfloor \frac{v}{B^{k-2}} \right\rfloor$$

Because $k = 3$, we have $f = min(k, 2) = 2$, and thus the initial approximation becomes:

$$w_{(0)} = \left\lfloor \frac{10000}{\lfloor 1119/B^{3-2} \rfloor} \right\rfloor B^{5-3-2} = \left\lfloor \frac{10000}{111} \right\rfloor = 90$$

Since $h - k \leq 2$, Watts algorithm terminates early, but clearly this is not safe as $\text{shinv}_{5,10}(1119) = 89 \neq 90$. In other words, by not considering the full value of v, we risk overestimating the initial short inversion. To address this issue, we avoid early termination by always performing at least one refinement after the initial approximation. Now that the initial approximation is never returned prematurely, we can afford a less precise initial approximation, as long as it still adheres to Theorem 3 for fast convergence. Therefore, we modify Theorem 4 to allow for a smaller prefix v when computing the initial inversion approximation.

Theorem 5. *(Modified Initial Value Choice)*

Let $B \leq v < B^{k+1}$ *and* $2v \leq u = B^h$ *for* $B \geq 16$*, and let* $v = VB^{k-f+1} + R$ *with* $f, V, R \in \mathbb{Z}$*,* $B^{f-1} \leq V < B^f$*,* $0 \leq R < B^{k-f+1}$ *and* $f \geq \min(k, 2)$*.*

Then the choice $w_{(0)} = \left\lfloor \frac{B^{f+1}}{V} \right\rfloor B^{h-k-2}$ *gives*

$$S_{\mathbb{Z}}^{\lceil \log_2 \log_2(u/v) \rceil}(w_{(0)}) \in \left\{ \left\lfloor \frac{u}{v} \right\rfloor, \left\lfloor \frac{u}{v} \right\rfloor - 1 \right\}.$$

Proof. Since $V \geq 4$ and $R < B^{k-f+1}$, we have $\frac{1}{4} > \frac{R}{VB^{k-f+1}}$ so

$$\left(1 + \frac{1}{4}\right) \frac{u}{v} > \left(1 + \frac{R}{VB^{k-f+1}}\right) \frac{u}{v} = \frac{B^h}{VB^{k-f+1}} \geq \left\lfloor \frac{B^{f+1}}{V} \right\rfloor B^{h-k-2}$$

On the other hand, since $V < B^f$, we have $\frac{B^{f+1}}{4V} > 1$ and

$$\left(1 - \frac{1}{4}\right)\frac{u}{v} = \frac{3}{4}\frac{u}{VB^{k-f+1} + R} \leq \frac{3}{4}\frac{B^{h-k+f-1}}{V}$$
$$< \left(\left\lfloor \frac{B^{f+1}}{V} \right\rfloor + 1 - \frac{B^{f+1}}{4V}\right) B^{h-k-2} < \left\lfloor \frac{B^{f+1}}{V} \right\rfloor B^{h-k-2}$$

The conditions of Theorem 3 are satisfied, so we have our result.

Intuitively, this means that inverting a short prefix of only 2 digits of v is now sufficient to ensure fast convergence. Or in other words, using the two most significant digits of v still lies within 25% of $\frac{B^h}{v}$. An advantage of this modification is that the smaller prefix can fit within a smaller native word size, allowing us to use a bigger base B without increasing the computational complexity of the initial division.

7.3 Overestimation from Divisor Prefixes

As hinted at in Section 6.3, employing the divisor prefixes strategy comes at a cost. It turns out that when divisor prefixes are used, the overestimation problem presented in the previous section goes beyond the scope of the initial approximation. Specifically, whenever we don't consider the entire v as there are edge cases where the least significant digits affect the much higher ones, resulting in an overestimation. We show an example of such case:

$$\left\lfloor \frac{10000000000}{1111119} \right\rfloor = 8999 \neq \left\lfloor \frac{1000000000}{111111} \right\rfloor = 9000$$

Essentially what happens is that when $\frac{10000000000}{1111119} \approx 8999.94$ i.e. fractional part approximates 1, even a small fractional increase is enough to tip the result over to the next integer. Because the divisor and quotient are inversely proportional, decreasing the divisor by a certain percentage results in an increase in the quotient by the same percentage. Since we have $\frac{9000-8999.94}{9000} \cdot 100 \approx 0.00067$, decreasing v by at least 0.00067 percent is enough to round the result up to 9000. In our case the decrease is: $\frac{1111119-1111110}{1111119} \cdot 100 \approx 0.00081\% > 0.00067\%$, which explains why we observe the incorrect result. We illustrate this issue using the optimized iteration scheme from Equation 9, which incorporates both shorter iterates and divisor prefixes:

$$w_{(i+1)} = S_{\mathbb{Z}}\left(k + 2\ell_{(i)} - s_{(i)},\ \text{shift}_{-s_{(i)}} v,\ \text{shift}_{\ell_{(i)}} w_{(i)}\right)$$
$$= w_{(i)}B^{\ell} + \left\lfloor w_{(i)}\left(B^{\ell} - B^{-k+s_{(i)}}\lfloor vB^{-s_{(i)}}\rfloor w_{(i)}\right)\right\rfloor$$

where

$$\ell_{(i+1)} = 2\ell_{(i)} - 1, \qquad s_{(i)} = \max(0,\ k - 2\ell_{(i)} + 1)$$

Specifically, we compute $\text{shinv}_{10,10}(1111119)$, where $h = 10$ and $k = 6$. In this example, we omit guard digits as it simplifies the computation and does not affect the outcome. Note that each iteration shifts the intermediate result by -1 (i.e. divides by 10), to account for the subtraction in $\ell_{(i+1)} = 2\ell_{(i)} - 1$.

- **Initial approximation:**

$$w_0 = \left\lfloor \frac{10000}{111} \right\rfloor = 90$$

- **Refine iteration 1:**

$$\begin{aligned} &w_0 = 90, \quad \ell = 2, \quad s = 3 \\ &w_1 = \left\lfloor \frac{9000 + \left\lfloor 90 \left(100 - \frac{1}{1000} \cdot 1111 \cdot 90\right) \right\rfloor}{10} \right\rfloor = 900 \end{aligned}$$

- **Refine iteration 2:**

$$\begin{aligned} &w_1 = 900, \quad \ell = 3, \quad s = 1 \\ &w_2 = \left\lfloor \frac{90000 + \left\lfloor 900 \left(1000 - \frac{1}{100000} \cdot 111111 \cdot 900\right) \right\rfloor}{10} \right\rfloor = 9000 \end{aligned}$$

- **Termination:** Since $h - k \leq \ell$, terminate and return 9000.

Hence, the algorithm returns 9000, while the actual whole shifted inverse equals $\left\lfloor \frac{10000000000}{1111119} \right\rfloor = 8999$. In turn, this can make the quotient computation as defined in Theorem 1 invalid. Suppose we define shinv' which overapproximates the original shinv by λ:

$$\text{shinv}_{n,B}(v) = \left\lfloor \frac{B^n}{v} \right\rfloor \qquad \Rightarrow \qquad \text{shinv'}_{n,B}(v) = \left\lfloor \frac{B^n}{v} \right\rfloor + \lambda, \quad \lambda \in \mathbb{Z}_{\geq 0}$$

Since $\lfloor x + y \rfloor = \lfloor x \rfloor + \lfloor y \rfloor + \epsilon$ where $\epsilon \in \{0, 1\}$, we have:

$$\begin{aligned} \text{shift}_{-h}(u \cdot \text{shinv'}_h\, v) &= \text{shift}_{-h}\left(u \cdot \left(\left\lfloor \frac{B^n}{v} \right\rfloor + \lambda\right)\right) \\ &= \text{shift}_{-h}(u \cdot \text{shinv}_h v + u\lambda) \\ &= \text{shift}_{-h}(u \cdot \text{shinv}_h v) + \left\lfloor \frac{u\lambda}{B^h} \right\rfloor + \epsilon, \quad \epsilon \in \{0, 1\} \end{aligned}$$

Which allows us to express the quotient based on shinv' as:

$$\begin{aligned} u \,\text{quo}\, v &= \text{shift}_{-h}(u \cdot \text{shinv}_h\, v) + \delta, \quad \delta \in \{0, 1\} \\ &= \text{shift}_{-h}(u \cdot \text{shinv'}_h\, v) + \delta - \left\lfloor \frac{u\lambda}{B^h} \right\rfloor - \epsilon, \quad \epsilon, \delta \in \{0, 1\} \end{aligned}$$

This shows that directly applying the quotient computation as presented by Watt while using a shifted inverse overestimated by λ can result in an overestimated quotient by:

$$\left\lfloor \frac{u\lambda}{B^h} \right\rfloor + \epsilon, \quad \epsilon \in \{0, 1\}$$

This is quite problematic, as we want to continue using the optimized divisor prefixes, because it allows for smaller intermediate products. Obviously we want to utilize this optimization which significantly reduces the asymptotic runtime, but we cannot compromise on the correctness. Fortunately, it turns out that the algorithm overestimates the shifted inverse by at most 1, that is $\lambda \in \{0, 1\}$. Given that $u \leq B^h$, it follows that $\left\lfloor \frac{u\lambda}{B^h} \right\rfloor = 0$, which means that the quotient is overestimated by at most 1. This allows us to restate Theorem 1 as follows:

Theorem 6. *(Quotient by shinv' in $\mathbb{Z}$)*

Given two positive integers u and v, with $u \leq B^h$,

$$u \; quo \; v = shift_{-h}(u \cdot shinv'_h \, v) + \delta, \quad \delta \in \{-1, 0, 1\}$$

where

$$shinv'_{h,B}(v) = \left\lfloor \frac{B^h}{v} \right\rfloor + \lambda, \quad \lambda \in \{0, 1\}$$

Theorem 6 relies on the fact that $\lambda \in \{0, 1\}$, which is not immediately obvious. To understand this we consider $q = \frac{B^h}{v}$ which has $h-k$ base-B digits. Since $q < B^{h-k}$ it implies that any relative change less than $\frac{1}{B^{h-k}}$ will affect q by less than 1. As stated previously, the divisor and quotient are inversely proportional, which means that as long as the divisor is decreased by less than $\frac{1}{B^{h-k}}$, we are guaranteed that the increase in the quotient is less than 1. Recall that $B^k \leq v < B^{k+1}$, which means that v has $k+1$ digits. If we keep only the $h-k+1$ most significant digits of v, the remaining part has $B^{k+1-(h-k+1)} = B^{2k-h}$ digits. From this we get:

$$\frac{B^{2k-h}}{v} < \frac{B^{2k-h}}{B^k} = \frac{1}{B^{h-k}}$$

Thus, as long as we consider the $h-k+1$ most significant digits of v, the remaining $2k-h$ digits will change v by less than $\frac{1}{B^{h-k}}$, resulting in an error in q less than 1. Now consider how many digits of v are actually used in each iteration of `Refine3`. The algorithm shifts v by $-\max(0, k-2l+1-q)$. In the last iteration of `Refine3` the minimum value of l is $\left\lfloor \frac{h-k}{2} \right\rfloor$. Thus, since $g = 2$, the maximum amount that v is shifted in the last iteration is:

$$k - 2\left\lfloor \frac{h-k}{2} \right\rfloor + 1 - 2 \geq 2k - h - 1$$

Because v has $k+1$ digits, the smallest shifted prefix of v has $k+1-(2k-h-1) = h-k+2$ digits. This means that we always consider at least $h-k+2$ digits of v in the last iteration, which is more than enough to guarantee that the impact on q is at most 1. This justifies the assumption that $\lambda \in \{0, 1\}$. Now that we have established that the whole shifted inverse can overestimate by 1, we need to update the division algorithm presented in Section 4.1 to account for $\delta = -1$ as well. The result is Algorithm 4 as seen below:

Algorithm 4: Div(u, v, m, B) in $\mathbb{Z}$

Input: $u,\ v \in \mathbb{Z}_+^m,\ m,\ B \in \mathbb{Z}_+$
Output: (q, r) where $q = u \operatorname{quo} v$, $r = u \operatorname{rem} v$
Uses: Prec, to compute precision
Shift, for shifting the integer
Shinv, the shifted inverse (Algorithm 7)

```
1:  function Div(u, v, m, B)
2:      h ←prec(u)
3:      q ← shift_{-h}(u · shinv(v, h, B))        ▷ Quotient
4:
5:      ▷ Handle δ = −1
6:      if u < v · q then
7:          q ← q − 1
8:      r ← u − v · q                              ▷ Remainder
9:
10:     ▷ Handle δ = 1
11:     if r ≥ v then
12:         q ← q + 1
13:         r ← r − v
14:     return (q, r)
```

7.4 Refined Algorithm

The refined quotient algorithm based on Algorithm 6.5 by Watt, considers the all of the changes outlined in this section. It still consists of the 3 main components: special case handling, initial approximation and iterative refinement.

Special Case Handling (lines 2-10) This remains the same as the original algorithm, as our redefined initial value choice (Theorem 5) still still relies on the assumption that $B < v \leq B^h/2$ and $B >= 16$.

Initial Approximation (lines 11-13) The initial approximation now inverts only the two most significant digits of v, compared to the previous three-digit approach. As described in Theorem 5, inverting this two-digit short prefix is still an accurate enough approximation to ensure fast convergence. Because the case $k = 0$ is already handled by the first special case $v < B$, we can safely assume $k \geq 1$, implying that v always has at least two digits. Thus it suffices to always use $l = 2$ instead of the original $l = min(k, 2)$. We use the semantically equivalent B^3 quo V instead of $(B^3 - V)$ quo $V + 1$. B^3 fits within 4 digits, and since computer architectures favor sizes that are powers of 2, there is no real advantage in decreasing the size to 3 digits. Lastly, we no longer return the initial approximation early, as we showed that it is not safe in Section 7.2.

Iterative Refinement (Refine, Step, PowDiff) As the initial approximation is no longer considered safe for early return, it is always refined unless one of the special cases applies. This initial value is refined using one of the refinement

methods: `Refine1`, `Refine2` or `Refine3`. `Refine3` employs both shorter iterates and divisor prefixes, allowing smaller products, making it more efficient and the one used in practice. As a result, we have omitted the asymptotically inferior `Refine1` and `Refine2`, and renamed `Refine3` to `Refine`. `Refine1` and `Refine2` can be found in Appendix 8.

The original while-loop has been replaced with a for-loop, as the number of required iterations is predetermined by the values of h and k. Specifically, the loop continues until l reaches at least $h - k$. Since each iteration updates l by $l \leftarrow l + m - 1$ and $m = l$ in all except the final iteration, we can model the loop progress by solving the recurrence relation:

$$l_{i+1} = 2l_i - 1$$

We use forward substitution to try and identify a pattern:

$$\begin{aligned}
l_1 &= 2l_0 - 1 \\
l_2 &= 2l_1 - 1 = 2(2l_0 - 1) - 1 = 4l_0 - 3 \\
l_3 &= 2l_2 - 1 = 2(4l_0 - 3) - 1 = 8l_0 - 7 \\
l_4 &= 2l_3 - 1 = 2(8l_0 - 7) - 1 = 16l_0 - 15
\end{aligned}$$

We observe that iteration l_i is given by:

$$l_i = 2^i l_0 - (2^i - 1)$$

Since $l_0 = 2$ it allows us to simplify:

$$l_i = 2^i \cdot 2 - (2^i - 1) = 2^{i+1} - 2^i + 1 = 2^i + 1$$

We can now determine for which i we have $l_i \geq h - k$:

$$\begin{aligned}
2^i + 1 &\geq h - k \\
2^i &\geq h - k - 1 \\
i &\geq \log_2(h - k - 1)
\end{aligned}$$

Since i is an integer we round up to nearest whole number:

$$i = \lceil \log_2(h - k - 1) \rceil$$

This means that the original algorithm performs $i = \lceil \log_2(h - k - 1) \rceil$ Newton iterations when $l_0 = 2$. However, since we no longer return the initial approximation early, we must perform additional iterations to compensate for the inaccuracy of the initial guess. In practice, we found that performing just one extra iteration in rare cases was insufficient to guarantee the accuracy of the 2-digit inversion. Therefore we opted for two extra initial iterations, although a formal analysis would be required to fully justify this choice. Nonetheless, this results in a total iteration count of:

$$\lceil \max(\log_2(h - k - 1),\ 0) \rceil + 2$$

The Step function is responsible for performing a single Newton iteration, adding or subtracting based on the sign boolean returned from Powdiff. In the case of subtraction, we also apply a correction for potential rounding errors in the shift, as described in Section 7.1. Powdiff has been altered to avoid negative values, as our multi-precision integer representation does support them. It now computes the absolute value of $B^h - v \cdot w$ along with a boolean flag indicating the true sign of the result. It still leverages the close products strategy from Section 6.4 for improved efficiency. The result is the optimized Algorithm 7 as seen below:

Algorithm 5: PowDiff(v, w, h, ℓ, B) in $\mathbb{Z}$

Input: $v,\ w,\ h,\ \ell,\ B \in \mathbb{Z}_{>0}$ such that $\text{prec}|w - \text{SHINV}_h v| \leq \text{prec}(w) - \ell$
Output: $(sign,\ |B^h - v \cdot w|)$
Uses: $\text{MULT}(a,\ b) = a \cdot b$
$\text{MULTMOD}(a,\ b,\ d,\ B) = (a \cdot b) \bmod B^d$

1: **Function** `PowDiff(`$v,\ w,\ h,\ \ell,\ B$`)`**:**
2: $L \leftarrow \text{prec}_B v + \text{prec}_B w - \ell + 1$;
3: **if** $v = 0 \vee w = 0 \vee L \geq h$ **then**
4: **if** $B^h > \mathit{MULT}(v, w)$ **then return** $(1,\ B^h - \text{MULT}(v, w))$;
5: **else return** $(0,\ \text{MULT}(v, w) - B^h)$;
6: **else**
7: $P \leftarrow \text{MULTMOD}(v, w, L, B)$;
8: **if** $P = 0$ **then return** $(1,\ 0)$;
9: **else if** $P_{L-1} = 0$ **then return** $(0,\ P)$;
10: **else return** $(1,\ B^L - P)$;

Algorithm 5: SHINV(v, h, B) in $\mathbb{Z}$

Input: $v,\ h,\ B \in \mathbb{Z}_{>0},\ B^k \le v < B^{k+1}$

Output: $\text{SHINV}_h v$ ▷ All shifts are with respect to B

Uses: MULT, a multiplication method
POWDIFF, to compute $B^h - v \cdot w$ (Algorithm 2)
REFINE

Function `Shinv`(v, h, B)**:**
  ▷ Group digits if base is small
  if $B < 16$ **then**
    $p \leftarrow \min(6 - B, 2)$
    return $\text{SHIFT}_{h \text{ rem } p - p}\text{SHINV}(v, h \text{ quo } p + 1, B^p)$
  ▷ Special cases guarantee $B < v \le B^h/2$
  if $v < B$ **then return** B^h quo v ▷ Divide by 1 digit
  if $v > B^h$ **then return** 0
  if $2v > B^h$ **then return** 1
  if $v = B^k$ **then return** B^{h-k}
  ▷ Form initial approximation, returning it if sufficient
  $V \leftarrow v_{k-1} + v_k \cdot B$
  $w \leftarrow B^3$ quo V ▷ Divide 4 digits by 2 digits
  return REFINE3(v, h, k, w, ℓ) ▷ Refine w iteratively
Function `Refine`(v, h, k, w, ℓ)**:**
  $g \leftarrow 2$ ▷ Guard digits
  $w \leftarrow \text{SHIFT}_g w$
  for $i \leftarrow 0$; $i < \lceil \max(\log_2(h - k - 1),\ 0) \rceil + 2$; i++ **do**
    $m \leftarrow \min(h - k + 1 - \ell,\ \ell)$
    $s \leftarrow \max(0,\ k - 2\ell + 1 - g)$ ▷ How to scale v
    $w \leftarrow \text{SHIFT}_{-1}(\text{STEP}(k + \ell + m - s + g,\ \text{SHIFT}_{-s}(v),\ w,\ m,\ \ell,\ g))$
    if $i < 2$ **then** $\text{SHIFT}_{-m}(w)$
    else
      $\text{SHIFT}_{-1}(w)$
      $\ell \leftarrow \ell + m - 1$
  if $(h - k < 2)$ **then** $\text{SHIFT}_{h-k-4}(w)$
  else $\text{SHIFT}_{-2}(w)$
Function `Step`(h, v, w, m, ℓ, g)**:**
  $(sign,\ x) \leftarrow \text{POWDIFF}(v, w, h - m, \ell - g, B)))$
  if *sign* **then** $\text{SHIFT}_m(w) + \text{SHIFT}_{2m-h}(\text{MULT}(w, x))$
  else
    $tmp \leftarrow \text{MULT}(w, x)$
    $res \leftarrow \text{SHIFT}_m(w) - \text{SHIFT}_{2m-h}(tmp)$
    if *any of the* $2m - h$ *least significant digits of tmp are nonzero* **then**
      $res \leftarrow res - 1$
    return res

8 Efficient CUDA Prototype

In order to get a reference for the performance of a highly optimized low-level division implementation, we develop one such prototype based on the revised whole shifted inverse algorithm from Section 7.4 in CUDA. Our implementation applies this technique to multi-precision integers that fits within a CUDA block, leveraging temporal reuse from scratchpad memories, following the same approach used in prior work on addition and multiplication [30]. We parallelize the key components of the shifted inverse iteration, employing efficient use of shared memory and register storage, and apply classical techniques such as efficient sequentialization to maximize throughput. The resulting prototype serves both as a performance benchmark and a demonstration of how exact arithmetic division can be implemented directly on GPUs using only integer operations.

8.1 General Strategies & Limitations

This subsection outlines key strategies and limitations that guided the process of designing an efficient division kernel. We adopt the following notation:

Word size: The number of bits in each digit of the big integer. We assume a word size of either 16-bit, 32-bit or 64-bit, as these are the sizes supported by the prototype.

M: The total number of digits in the big integer. For example, an integer with 2^{17} bits could be represented using $M = 2048$ and a 64-bit word size.

Sequentialization factor (Q): The amount of sequential work each thread performs. For simplicity, we assume that Q evenly divides M.

Kernel Memory Limitations Currently the CUDA prototype supports integer division on integers up to 2^{18} bits in size. This is the upper limit of what is possible to support if all operations are to be performed entirely within a single block, due to limitations on the maximal amount of shared memory[2]. In principle, one could go beyond this bound by utilizing global memory, but this approach would require expensive global memory management, significantly impacting performance.

In addition to shared memory, CUDA also has restrictions on the number of registers available per thread and per block. The current version supports a maximum of 255 registers per thread or 64k registers per block, whichever is lowest. Since we want to maximize occupancy we use all 1024 threads in a block when computing on integers larger than 2^{15} bits, meaning in practice we may have as little as 64 registers per thread. The amount of register memory used by the kernel is extremely difficult to reason about, as it is a point of complex compiler optimizations. When the register demand exceeds the bounds of the specifications,

[2]The current maximum amount of shared memory for a CUDA block is 99KB, and since we need to store two 2^{18}-bit integers of 32KB each in shared memory as part of the multiplication, this bounds the division as well.

the compiler resorts to store these in a higher level cache, a fallback mechanism known as register spilling [4]. Accessing this cache is substantially slower than the register memory directly, and as such this is generally to be avoided if possible, as it can incur significant latency. Unfortunately, our implementation proved to easily reach the maximum number of registers per block when enough threads are used. We suspect that the reason for the large consumption of registers stems from the compiler not being able to reuse registers efficiently. This resulted in a kernel launch error, since the compiler tries to assign too many registers per block during runtime. Luckily we were able to mitigate this by specifying the number of threads per block during compile time by giving the kernel statically known launch bounds using C++ templates. This allowed the compiler to allocate just enough registers per thread to fit within the block register limit, by spilling the remaining ones.

Determining an optimal value of Q we are encouraged to keep it small in order to limit sequential dependencies, and since our classical multiplication implementation is optimized for $Q = 4$ we found the ideal value to be this. When operating on the biggest integers supported with this value of Q we spill 160 bytes to storage, corresponding to 40 registers (as each register stores 32 bit values). In addition we access 392 bytes of spilled storage, corresponding to each stored register being read just under 3 times. We found that if we increased Q sufficiently enough, we could eliminate register spilling completely. This is because as Q is increased, the number of threads decreases correspondingly because the number of threads in a block is set to M/Q, and thus the total register count within the block decreases as well. However regardless of the extra accesses to higher-level caches as a result of spilling, we found that increasing Q beyond 4 reduced the overall performance as well, making the change not worthwhile.

Problem Instances per Block When designing the CUDA kernel we consider how many instances of multi-precision division should be computed per block, we call this metric the number of *problem instances per block* (IPB). As we are motivated by performance reasons to store integers in fast shared (scratchpad) memory, we want to limit the accesses to global memory to the minimum required. Problem instances are thus not shared between blocks, as this would require expensive reads/writes to grid-level (global) memory.

Because we are targeting performance on very large integers, we do not concern ourselves with implementing multiple problem instances per block, as this would only increase performance when integers are small enough such that multiple can be stored simultaneously. Most critically, allowing multiple instances per block would result in nested-irregular parallelism, which would then require flattening, something extremely difficult to achieve in our case due to input-dependent branching. Additionally, multiple problems of the same length might terminate at completely different speeds, e.g. by hitting one of the special fast cases, causing warps to be idle while waiting for slower instances to return.

For these reasons the CUDA kernel is structured to always consider exactly one problem instance per block, which ensures that different blocks can compute independent problems concurrently, with no inter-dependencies. Thus during

stress-testing we simply compute enough problem instances via concurrently running blocks in order to sufficiently saturate the GPU.

Load balancing Ideally, we want to spawn a kernel where all threads within each block remain fully active and saturated throughout the entire division computation. However, due to the use of shorter iterates and divisor prefixes, the sizes required for the arithmetic operations increase for each iteration, making it challenging to maintain full thread saturation. Since we need at least enough threads to perform a full-size multiplication, when performing arithmetics of smaller sizes threads are going to be idle.

If possible, we would want to pair problem instances from later iterations with those from earlier ones to better distribute the computational load. Unfortunately, since CUDA's execution model enforces block independence, once a warp has been assigned a block, it can not be reused by other blocks until the block terminates, making this a nonviable approach. An alternative approach to achieve a more balanced computational load is to increase the sequentialization factor Q. However, this could exhaust registers and local memory leading to sequential bottlenecks, thus decreasing efficiency. Also, since we only spawn one block per SM, initializing a block with fewer threads and thus fewer warps limits the SM's ability to perform key optimizations such as efficient instruction scheduling and memory latency hiding.

Coalesced Global Memory Access Accessing global memory is extremely slow, even on modern GPU architectures, requiring around 290 clock cycles, compared to shared memory which takes roughly 20 cycles [5]. Thus, in order to optimize the memory access pattern of the kernel, we are encouraged to limit the number of accesses to global memory to the initial read and final write. Each read and write to global memory should be managed carefully, to fully exploit spacial locality, and perform as few memory transactions as possible. This means copying Q total elements with all threads, such that each thread copies elements by a stride corresponding to the total number of threads. This maximizes both spacial and temporal locality as whenever a warp of threads accesses global memory, the cached reads can easily be reused as neighboring threads access neighboring elements. Such an implementation can be found in Listing 1 where it is assumed that Q evenly divides M, thus each thread copies Q elements from global to shared, and finally to register memory.

```
1  template<class uint_t, uint32_t M, uint32_t Q>
2  __device__ inline void
3  cpyGlb2Sh2Reg( uint_t* AGlb
4               , volatile uint_t* ASh
5               , uint_t AReg[Q]
6  ) {
7      const int glb_offs = blockIdx.x * M;
8
9      #pragma unroll
10     for (int i = 0; i < Q; i++) {
11         int idx = i * blockDim.x + threadIdx.x;
12         ASh[idx] = AGlb[idx+glb_offs];
```

```
13      }
14      __syncthreads();
15      #pragma unroll
16      for (int i = 0; i < Q; i++) {
17          AReg[i] = ASh[Q * threadIdx.x + i];
18      }
19  }
```

Listing 1: Coalesced copy of big integer from global to shared to register memory.

An essential part that you do not seem to explain, is that "allocating arrays to register memory" is possible (efficient), only when the compiler can perform scalarization, i.e., replacing a thread private array of Q elements with Q registers. This means that the indexing into the thread privte arrays has to be very simple.

Register Allocation for Multi-Precision Integers When operating on multi-precision integers, we ideally want most memory accesses to go through the lowest latency thread-level (register) memory for faster lookup. With that being said, storing the entire integer per thread is infeasible, as each thread only has a maximum of 255 32-bit registers for storing variables. Instead, we distribute the representation of the integer across the threads in a CUDA block, assigning to each thread a small slice of the integer of size equal to the sequentialization factor Q. The integers are partitioned such that each thread t stores the digits in the range:

$$\{t \cdot Q, t \cdot Q + 1, \ldots, t \cdot Q + Q - 1\},$$

i.e. in sequential order which can also be seen depicted in Figure 11. By ensuring that the indexing into these thread private arrays are simple and known at compile time, the compiler can perform scalarization, which maps the digits to actual registers. This is typically achieved through loop unrolling, a technique used extensively throughout our code. This enables each thread to perform extremely efficient computation on its assigned portion of the integer entirely within registers, thereby limiting shared memory access.

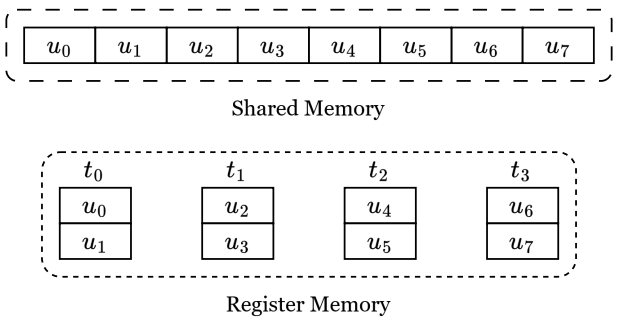


Figure 11: Integer being store in register memory in partitions of $Q = 2$.

8.2 Shifting

Although the shift operation is relatively simple, it is central to our algorithm, and thus we cover its implementation. The shift is given by:

$$\text{shift}_B(u, n) = \begin{cases} u \cdot B^n & n \geq 0 \\ \lfloor \frac{u}{B^{|n|}} \rfloor & \text{otherwise,} \end{cases}$$

and can be intuitively thought of as an arithmetic shift of the big integer u by n digits, either to the left or the right, while zeroing out the rest.

```
template<class uint_t, uint32_t M, uint32_t Q>
__device__ inline void
shift( int n
     , uint_t u[Q]
     , volatile uint_t* sh_mem
     , uint_t RReg[Q]
) {
    #pragma unroll
    for (int i = 0; i < Q; i++) {
        int idx = Q * threadIdx.x + i;
        int offset = idx + n;
        uint_t val = 0;
        if (offset >= 0 && offset < M) {
            val = u[i];
        } else {
            offset = M-idx-1;
        }
        sh_mem[offset] = val;
    }
    __syncthreads();
    #pragma unroll
    for (int i = 0; i < Q; i++) {
        RReg[i] = sh_mem[Q * threadIdx.x + i];
    }
}
```

Listing 2: Shift

The CUDA function accepts four parameters. The big integer u stored in register memory, the shift amount n, a shared memory pointer for cross-thread communication and a register array for the result as we might want to reuse u later in the algorithm. We are generally encouraged to perform as many operations in register memory since lookups there incurs a much smaller time penalty in comparison to shared memory. Depending on the size of n however, most or all of the elements in a given thread will have to be shifted into another thread, resulting in unavoidable accesses to shared memory. More specifically if $n \geq Q$ we need to move the entire integer into shared memory anyway. For simplicity, and since the above condition is expected to occur often, we move the entire offset integer to shared memory at once, before moving it back into register memory at the shifted address of the result. Finally we are required to synchronize all threads in the block after writing to shared memory, in order to eliminate the possibility of a true dependency due to racing threads [29]. The result is the

implementation of the shift operator for multi-precision integers that can be seen in Listing 2, which is able to shift in-place.

8.3 Initial Value Computation

As discussed in Section 7.2, a sufficiently accurate initial approximation for fast convergence can be computed in constant time using $\lfloor \frac{B^3}{V} \rfloor$, i.e. inverting a short prefix of the two most significant digits of v. However, since the divisor B^3 spans four digits, performing this division requires native sizes that are four times the word size. For word sizes up to 32 bits this is not an issue, since C++ has native support for unsigned integers up to 128 bits. However, as our implementation is based on the multiplication from [30], which reports a performance increase of approximately 50% when using 64-bit digits compared to 32-bit, we naturally want to support 64-bit digits as well. For 64 bits, computing $\lfloor \frac{B^3}{V} \rfloor$ would require 256-bit arithmetic, which is beyond the limits of any native type in CUDA. Therefore we split the initial approximation into two cases, one for 64 bits word sizes, and for all smaller ones. The resulting code is found in Listing 3.

```
if (threadIdx.x == 0) {
    ubig_t tmp;
    ubig_t V = (ubig_t)VSh[k - 1] | (ubig_t)VSh[k] << Base::bits;
    if (Base::bits == 64) {
        tmp = divide_u256_by_u128((__uint128_t)1 << 64, 0, V);
    } else {
        tmp = ((uquad_t)1 << 3*Base::bits) / V;
    }
    RReg[0] = (uint_t)(tmp);
    RReg[1] = (uint_t)(tmp >> Base::bits);
    if (tmp == 0) RReg[2] = 1;
}
```

Listing 3: Computation of initial approximation as part of the `shinv` function.

To construct V, we simply combine the two most significant digits of v using a bitwise OR, where the most significant digit is shifted by a word size, as the shift guarantees no overlapping bits between the digits. For word sizes less than 64 bits, we apply native division on a unsigned integer of four times the word size, denoted as a `uquad_t`.

For the 64-bit case, although native 256-bit arithmetic is unavailable, we still have enough space in a uint128 to store the entire divisor, meaning that the division can be achieved via short division. Since short division requires double the divisor size for the remainder, we opt for a bit-wise version. This approach allows us to perform the division using only uint128_t values by handling overflow explicitly using a bool flag. The exact implementation is seen below.

```
__device__ inline uint128_t divide_u256_by_u128(uint128_t high,
                                uint128_t low, uint128_t divisor) {
    uint128_t quotient = 0;
    uint128_t rem = 0;
    bool overflow = false;
```

```
    for (int i = 192; i >= 0; i--) {
        if (rem & (__uint128_t)1 << 127) {
            overflow = true;
        }
        rem <<= 1;

        if (i == 192) {
            rem |= 1;
        }
        quotient <<= 1;
        if (rem >= divisor || overflow) {
            rem -= divisor;
            quotient |= 1;
            overflow = false;
        }
    }
    return quotient;
}
```

Listing 4: Bitwise long division defined for a 192-bit numerator.

Finally, the computed short prefix inverse is stored in the corresponding register array entries of the big integer. In the case of it being 0, it corresponds to a corner case where the result is three digits, but the last bit overflows since tmp is only two digits long. Specifically, this occurs when diving B^3 by B, which results in $B^2 = [0, 0, 1]$. In that case we store a 1 in $RReg[2]$. Also, since $Q \geq 4$, we can store the entire result in the registers of the first thread, allowing us to perform the initial approximation sequentially in the first thread only.

8.4 Classical Multiplication

As previously established, multiplication is the main building block of our division algorithm, making it a critical component to optimize. We base our approach on the classical multiplication method presented in [30], which computes the convolution in an "embarrassingly parallel" fashion, as described in Section 5.3. However, the multiplication from [30] is highly specialized for the fixed M and Q, which are sent as compile time arguments to the kernel. As a result, it always computes exactly M elements utilizing all threads efficiently, while cutting off any potential overflow. In contrast, the whole shifted inverse algorithm performs variable sized multiplications. To avoid redundant computation and to preserve the algorithm's linear theoretical runtime with respect to multiplication, we adapt the original method to support dynamic multiplication based on the operand size. This technique shown in Listings 5.

```
if (maxMul <= blockDim.x) {
    smallMult<Base, Q>(USh, VSh, VReg, RReg, VReg, maxMul);
} else if (maxMul <= 2*blockDim.x) {
    smallMult2x<Base, Q>(USh, VSh, VReg, RReg, VReg, maxMul);
} else {
    bmulRegsQ<Base, 1, Q/2>(USh, VSh, VReg, RReg, VReg, M);
}
```

Listing 5: Dynamic multiplication switching based on operant sizes.

Moreover, Theorem 1 requires computing the product $u \cdot \text{shinv}_h\, v$. Both operands can potentially be of size h, which is upper-bounded by M. Since multiplication can potentially double in size, we need the full result of up to $2M$ digits to ensure that we do not lose information. As the original multiplication discards digits beyond M, we adapt it to ensure that all $2M$ resulting digits are preserved, we call this *Complete Multiplication*.

8.4.1 Small Multiplication

To efficiently handle smaller multiplications, we implement a separate function specialized for this case. Although the efficiency of small multiplications is not critical to the overall performance, provided it maintains the correct asymptotic behavior, we still aim for a highly efficient implementation in practice. For instance, a full-sized multiplication involves M^2 operations, whereas halving the operands yields $(\frac{M}{2})^2 = \frac{M^2}{4}$ operations, and further halving reduces it to $\frac{M^2}{16}$, and so on. Thus all subsequent multiplications, assuming only one is performed per iteration, collectively contribute to approximately $\frac{M^2}{4}$ of the total work load.

While our small multiplication implementation still adopts the structure from [30], it is limited to exclusively operate on big integers of size less than the number of threads initialized in the block. Instead of performing the entire convolution of M elements, we only compute the necessary entries, where each thread is responsible for a single entry in the convolution. This proposed load-balancing schema is illustrated in Figure 12. Although this scheme results in a more unbalanced workload between threads compared to the original approach, the total work is significantly reduced, resulting in less overhead.

$$
\begin{aligned}
w_0 &= u_0 \cdot v_0 && := t_0 \\
w_1 &= u_0 \cdot v_1 + u_1 \cdot v_0 && := t_1 \\
w_2 &= u_0 \cdot v_2 + u_1 \cdot v_1 + u_2 \cdot v_0 && := t_2 \\
&\vdots && \vdots \\
w_{n-3} &= u_0 \cdot v_{n-3} + \ldots + u_{n-3} \cdot v_0 && := t_{n-3} \\
w_{n-2} &= u_0 \cdot v_{n-2} + \ldots + u_{n-3} \cdot v_1 + u_{n-2} \cdot v_0 && := t_{n-2} \\
w_{n-1} &= u_0 \cdot v_{n-1} + \ldots + u_{n-3} \cdot v_2 + u_{n-2} \cdot v_1 + u_{n-1} \cdot v_0 && := t_{n-1}
\end{aligned}
$$

Figure 12: Small convolution where each thread computes `threadIdx.x` + 1 products. Assumes $n <$ `blocksize`.

The per-thread results of the convolution are two variables, *accum* of type bigint_t and *carry* of type uint_t. *accum* accounts for the overflow in multiplication, while the carry manages the potential overflow of the summations. Unfortunately, when publishing these results to shared memory, since we only compute one elm per thread, the high and carry will overlap. This requires us to publish to three locations in shared memory, which in turn results in 2 additions. Luckily, since addition is cheap relative to the convolution, the impact on performance is minimal. Additionally, since we assume that $Q \geq 4$, the available shared memory

is more than sufficient to store three slices of *blocksize*. This process is shown in Figure 13.

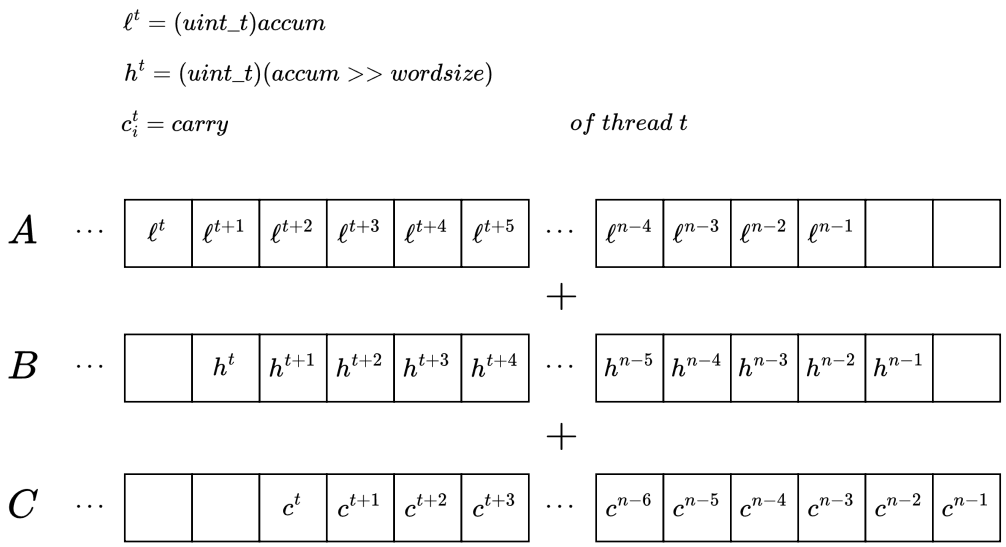


Figure 13: An example of publishing the per thread variables *accum* and *carry* to shared memory.

The CUDA implementation that combines the previous steps is shown in Listing 6. Lines 20-30 perform the convolution, while `reg2ShmConv` on line 33 publishes the result to shared memory. Lines 44-45 perform the additions of the low, high, and carry arrays from Figure 13. The final result is then copied back to register memory.

```
1  template<class Base, uint32_t Q>
2  __device__
3  void smallMult( volatile typename Base::uint_t* Ash
4                , volatile typename Base::uint_t* Bsh
5                , typename Base::uint_t Arg[Q]
6                , typename Base::uint_t Brg[Q]
7                , typename Base::uint_t Rrg[Q]
8                , uint32_t M
9  ) {
10     using uint_t = typename Base::uint_t;
11     using ubig_t = typename Base::ubig_t;
12     using carry_t= typename Base::carry_t;
13
14     // 1. copy from shared to register memory
15     cpyReg2Shm<uint_t,Q>( Arg, Ash, M );
16     cpyReg2Shm<uint_t,Q>( Brg, Bsh, M );
17     __syncthreads();
18
19     // 2. perform small convolution
20     ubig_t accum = 0;
21     carry_t carry = 0;
22     if (threadIdx.x < M) {
23         #pragma unroll
```

```
24          for (int i = 0; i <= threadIdx.x; i++) {
25              ubig_t ck = (ubig_t)Ash[i] * (ubig_t)Bsh[threadIdx.x ↩
                    - i];
26              uint_t accum_prev = (uint_t) (accum >> Base::bits);
27              accum += ck;
28              carry += (  ((uint_t)(accum >> Base::bits)) < ↩
                    accum_prev );
29          }
30      }
31
32      // 3. publish convolution to shared memory
33      reg2ShmConv<Base, ubig_t, carry_t, Q>(accum, carry, Ash, Bsh,↩
            M);
34      __syncthreads();
35
36      // 4. load back to registers.
37      uint_t arg; uint_t brg; uint_t crg;
38      if (threadIdx.x < M) {
39          arg = Ash[M + threadIdx.x];
40          brg = Bsh[M + threadIdx.x];
41          crg = Ash[2*M + threadIdx.x];
42      }
43
44      // 5. perform the addition of the carries.
45      uint_t res = baddRegsNaive<uint_t, uint_t, carry_t, Base::↩
            HIGHEST>( (carry_t*)&Bsh[2*blockDim.x], arg, brg );
46      res = baddRegsNaive<uint_t, uint_t, carry_t, Base::HIGHEST>( ↩
            (carry_t*)Bsh, res, crg );
47
48      // 6. copy the result back to registers.
49      Ash[threadIdx.x] = res;
50      __syncthreads();
51      cpyShm2Reg<uint_t, Q>( Ash, Rrg, M );
52  }
```

Listing 6: CUDA wrapper function that computes a small quadratic integer multiplication.

In addition to `smallMult` we have implemented a similar function `smallMult2x`, which allows for sizes up to two times the *blocksize*. The approach is very similar to the one mentioned above, and hence not shown. The convolution is performed as the original approach from [30], in a low and high part, meaning that each thread processes two elements instead of one in a load balanced fashion.

8.4.2 Complete Multiplication

Extending the original multiplication to produce the full $2 \cdot M$ result is relatively straightforward. We start by modifying the convolution, such that the low part processes the first m elements, while the high part is responsible for the subsequent m. This means that sequential factor Q effectively doubles. This structure shown in Figure 14.

$$
\begin{aligned}
w_0 &= u_0 \cdot v_0 && := t_0 \\
w_1 &= u_0 \cdot v_1 + u_1 \cdot v_0 && := t_1 \\
w_2 &= u_0 \cdot v_2 + u_1 \cdot v_1 + u_2 \cdot v_0 && := t_2 \\
&\vdots && \vdots \\
w_{m-2} &= u_0 \cdot v_{m-2} + \ldots + u_{m-3} \cdot v_1 + u_{m-2} \cdot v_0 && := t_{n-2} \\
w_{m-1} &= u_0 \cdot v_{m-1} + u_1 \cdot v_{m-2} + \ldots + u_{m-2} \cdot v_1 + u_{m-1} \cdot v_0 && := t_{n-1} \\
w_m &= u_0 \cdot v_m + u_1 \cdot v_{m-1} + \ldots + u_{m-1} \cdot v_1 + u_m \cdot v_0 && := t_0 \\
w_{m+1} &= u_0 \cdot v_m + \ldots + u_{m-2} \cdot v_1 + u_{m+1} \cdot v_0 && := t_1 \\
&\vdots && \vdots \\
w_{2m-3} &= u_{m-1} \cdot v_{m-3} + u_{m-2} \cdot v_{m-2} + u_{m-3} \cdot v_{m-1} && := t_{n-3} \\
w_{2m-2} &= u_{m-1} \cdot v_{m-2} + u_{m-2} \cdot v_{m-1} && := t_{n-2} \\
w_{2m-1} &= u_{m-1} \cdot v_{m-1} && := t_{n-1}
\end{aligned}
$$

Figure 14: Embarrassingly parallel load-balancing schema as proposed by [30] where each thread computes $2 \cdot m$ products.

Contrary to the previous convolutions, most of the work lies in the middle entries. Thus we can achieve a load balancing partitioning by assigning threads to entries that are m indices apart, i.e.:

$$\{(w_0, w_m), (w_1, w_{m+1}), \ldots, (w_{m-1}, w_{2m-1})\}.$$

This means that all threads perform a total of $2M$ multiplications (when $Q = 1$). With that being said, to our surprise, we did not observe a performance increase when using this load-balancing schema. Therefore, our current implementation uses the same entry mapping as in [30], where each thread is assigned its opposite counterpart i.e. $\{(w_0, w_{2m-1}), (w_1, w_{2m-2}), \ldots, (w_{m-1}, w_m)\}$, leading to unbalanced workload across threads. Another challenge in regards to this extension is that we no longer can store the entire result twice in shared memory, when adding the carries. Instead we opt for publishing half of the result in shared memory, perform the addition, before handling the remaining half.

The CUDA wrapper function for the complete multiplication implementation is found in Listing 7. Note that we have altered the implementation of addition, such that it returns an overflow flag as well, since the first addition might overflow. After performing both additions we add 1 to the entry m of the integer, if the flag is true. All in all this corresponds to approximately double the work for the convolution compared to the original approach, as each thread has to perform twice as many multiplications.

```
1 template<class Base, uint32_t IPB, uint32_t Q>
2 __device__
3 void bmulRegsQComplete( volatile typename Base::uint_t* Ash
4                 , volatile typename Base::uint_t* Bsh
5                 , typename Base::uint_t Arg[2*Q]
6                 , typename Base::uint_t Brg[2*Q]
7                 , typename Base::uint_t Rrg[4*Q]
8                 , uint32_t M
```

```
) {
    using uint_t = typename Base::uint_t;
    using ubig_t = typename Base::ubig_t;
    using carry_t= typename Base::carry_t;

    // 1. copy from global to shared to register memory
    cpyReg2Shm<uint_t,2*Q>( Arg, Ash, M );
    cpyReg2Shm<uint_t,2*Q>( Brg, Bsh, M );
    __syncthreads();

    // 2. perform the convolution
    uint_t lhcs[2][2*Q+2];
    wrapperConvQComplete<uint_t, ubig_t, 2*Q>( Ash, Bsh, lhcs, M );
    __syncthreads();

    volatile uint_t* Lsh = Ash;
    volatile uint_t* Hsh = Bsh;

    // 3. publish the low parts normally, and the high and carry shifted by one.
    uint_t highCarry[2];
    from4Reg2ShmQFirstHalf<uint_t, Q*2>( lhcs[0], Lsh, Hsh, highCarry, M );
    __syncthreads();

    // 4. load back to register and perform the addition of the carries.
    uint_t Lrg[2*Q];
    uint_t Hrg[2*Q];
    cpyShm2Reg<uint_t,2*Q>( Lsh, Lrg );
    cpyShm2Reg<uint_t,2*Q>( Hsh, Hrg );
    __syncthreads();

    bool overflow = baddRegsOverflow<uint_t, uint_t, carry_t, 2*Q, Base::HIGHEST>( (carry_t*)Lsh, (carry_t*)Hsh, Lrg, Hrg, Rrg, M );
    __syncthreads();

    from4Reg2ShmQSecondHalf<uint_t, Q*2>( lhcs[1], Lsh, Hsh, highCarry, M );
    __syncthreads();

    cpyShm2Reg<uint_t,2*Q>( Lsh, Lrg );
    cpyShm2Reg<uint_t,2*Q>( Hsh, Hrg );
    __syncthreads();

    baddRegs<uint_t, uint_t, carry_t, 2*Q, Base::HIGHEST>( (carry_t*)Lsh, Lrg, Hrg, &Rrg[Q*2] );
    if (overflow) add1<Base, Q*2>(&Rrg[Q*2], Hsh);
}
```

Listing 7: CUDA wrapper function that computes a complete quadratic integer multiplication.

8.4.3 Clipped Products

Sometimes only a portion of the digits of a big integer product are required. Usually these constitute the least or most significant digits, for example when computing initial values or refinement steps in iterative approximation schemes. Occasionally, only a middle segment of the product is required, though this is less common. Computing a segment of the product can be quite tricky, as the digits are not independent, i.e. carries from less significant positions might affect more significant ones.

For classical multiplication, computing the least significant digits is quite straightforward, as it just requires to limit the convolution range. This is effectively what happens inside MultMod, which computes $(a \cdot b)$ rem B^d, retaining only the d least significant digits of the product. However, the multiplication inside step computes a product of size $2N \times N$ and immediately left shifts the result my N digits, essentially only utilizing the last $2N$ digits. Consequently, the work spent computing the first N digits goes to waste. This redundant work can be mitigated by only computing a slice of the product instead, a technique described in [24], referred to as a*clipped product*, which are defined for both classical and Karatsuba multiplication methods.

The same optimization applies to the complete multiplication discussed in the previous section. The algorithm computes $\text{shift}_{-h}(u \cdot \text{shinv}_h\, v)$, i.e. the product is shifted by -h, and thus only the h most significant digits are preserved. Thus, by employing classical multiplication with clipped products, one can reduce the reduce the time complexity by approximately half, since we would only need half as many multiplications in the convolution. Although clipped products are highly relevant and would reduce the overall computational workload of our algorithm, we deem it beyond the scope of this thesis. The optimization may be explored in future work.

8.5 Warp-Level SOACs

The CUDA programming model provides highly optimized warp-level intrinsics that enable efficient data exchange and collective operations within a warp (i.e. a group of 32 threads that execute in lockstep). These intrinsics such as `__shfl_sync`, `__shfl_up_sync`, `__ballot_sync`, among others can be used to implement exceedingly low-latency warp-wide scan and reduce operations. In our implementation, we exploit these warp-level primitives to build specialized versions of the following SOACs with sequential dependencies:

- *Scan* used for the carry propagation during addition and subtraction, where carry flags are propagated throughout the integer in warp-level increments.
- *Reduce* for aggregating boolean conditions as part of determining if an integer is less-than or greater-than another.

Warp-level operations avoid shared memory, operating entirely within the warp's register file. Additionally, since warps perform operations in lockstep there is no need for synchronization barriers, as data-dependencies related to shared

memory resources do not occur in the same way. As a result, these operations are significantly faster than block-level or device-level equivalents. We found that utilizing these warp-level primitives improved the performance of addition in [30] by up to 10 percent. These results are showcased in Appendix 3.

Since these big integers exceed a single warp's width, we combine these warp-level scan/reduce with shared memory staging to construct the full block implementations. Each warp computes a partial result, which is then combined with others through a second level of scan/reduce operation at the block level. This multi-level scan/reduce maintains optimal performance regardless of integer size, we will therefore discuss the implementation of these individually in more detail below.

```
template<class OP>
__device__ inline int
scanIncWarp( int u
           , uint32_t lane
) {
    #pragma unroll
    for (int i = 1; i < WARP; i *= 2) {
        int elm = __shfl_up_sync(0xFFFFFFFF, u, (lane >= i) ? i :↩
             0);
        u = OP::apply(elm, u);
    }
    return u;
}
```

Listing 8: Warp-level scan implementation.

We define operation-invariant implementations using C++ templates, which allow us to plug in any custom *associative* operator (e.g. addition, maximum, logical OR). The low-level warp scan can be seen in Listing 8 which consists of $\log_2(\texttt{WARP})$ iterations, at each of which we shuffle up towards the highest element of the warp with the associative operation `OP::apply` [3]. The schema for which we shuffle upwards as part of the scan can be seen in Figure 15. The scan is consistent with the Kogge-Stone lookahead adder for calculating parallel prefixes [12], and while other scan implementations exist that result in less overall inter-thread communication, they present no immediate performance benefit due to all threads operating in lockstep.

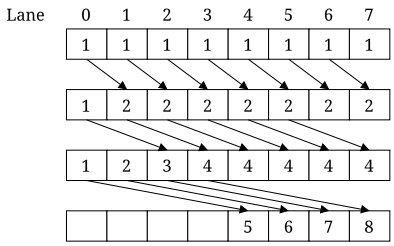


Figure 15: Example of warp-level scan with the addition operator with a warp of size 8.

For implementing the block-wide scan we initially scan each warp individually, this calculates the prefix sum within each warp but does not consider the exclusive prefix (i.e. the sum up to the beginning lane of the warp). The exclusive prefix

can however simply be computed by an additional warp-level scan over all of the last elements within each separate warp. Finally this exclusive prefix can be applied to each element within the corresponding warp which results in the code as seen in Listing 9. The implementation of the block-level scan is presently only used as part of the addition and subtraction functions, however because it uses generic operations it can easily be adapted to any function that can be modeled in terms of a scan.

```
template<class
template<class OP>
__device__ inline int
scanIncBlock( uint32_t u
            , volatile uint32_t* sh_mem
) {
    __syncthreads();
    int idx = threadIdx.x;
    const unsigned int lane   = idx & (WARP-1);
    const unsigned int warpid = idx >> lgWARP;

    int res = scanIncWarp<OP>(u, lane);

    if (lane == (WARP-1)) { sh_mem[warpid] = res; }
    __syncthreads();

    if (warpid == 0) {
        scanIncWarp<OP>(sh_mem[threadIdx.x], lane);
    }
    __syncthreads();

    if (warpid > 0) {
        res = OP::apply(sh_mem[warpid-1], res);
    }
    __syncthreads();

    sh_mem[idx] = res;
    __syncthreads();

    return res;
}
```

Listing 9: Block-level scan implementation.

The code for the block-level reduction is very similar to the scan implementation, except it is missing the final step where the exclusive prefix is applied for all elements of the block. Instead it is simply applied to the last element of the block which is then returned. This function is operation invariant similar to the scan implementation, and as such can easily be extended. It is used to implement the less-than operator for comparing the sizes of big integers. The operator itself is not inherently associative, instead we use the operator presented by Bringgaard in [10] which is defined as follows:

$$(l1,\ e1) \odot (l2,\ e2) := (l2, \vee\ (e2 \wedge l1),\ e1 \wedge e2)$$

The intuition is that if the most significant digit of the first integer is strictly smaller than the equivalent for the second, then the entire first integer is less.

Alternatively, if the two digits are equal then the condition is propagated down towards the less significant digits. The operand is shown to return the boolean value corresponding to whether the first integer is strictly smaller than the second, and it is proven to be associative and have neutral element (`false`, `true`) as part of [10], hence why we omit this. The final code for the block-level reduction which is used to implement this less-than can be found in Listing 14.

8.6 Optimizing Arithmetic on Powers of B

A central part of the whole shifted inverse algorithm revolves around performing arithmetic on integers corresponding to the base B lifted to some power. We denote these as B^n where n is a positive integer, which is numerically represented by a big integer where all digits are zero, except for the n'th digit which is 1. Due to the numerical representation of this type of integer, as the base B is increased, the integer quickly becomes sparsely populated. This results in a lot of unnecessary overhead both in terms of computation but also from simply storing them in memory. Therefore, we exploit the structure of these base-B big integers, to define a set of specialized auxiliary functions, allowing us to define simpler more efficient implementations of addition, subtraction and comparison. Although these operations are not asymptotically dominating the runtime of division, we see to it that they are optimized regardless.

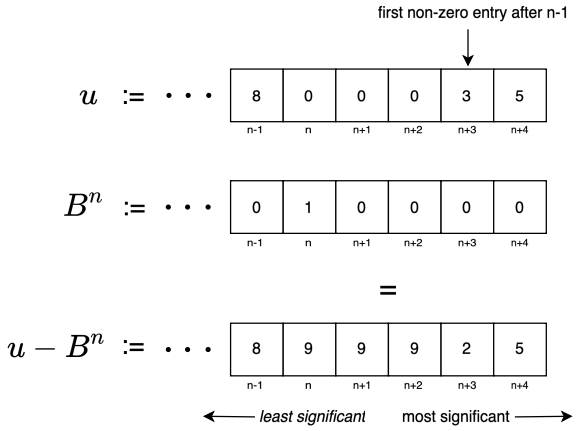


Figure 16: Illustration of efficient specialized subtraction, $u - B^n$ where $u \geq B^n$.

An example of a specialized subtraction is found in Listing 10. The function computes $u - B^n$ where $u \geq B^n$, making the result a positive number. Since B^n is defined as an integer with a 1 at n'th index and zero otherwise, there is no need to save the entire number in memory, and thus only a uint32 is passed as parameter. The function starts by locating the first non-zero digit with an index greater than or equal to n. This is first done sequentially within each thread and then across threads using shared memory, meaning that the inter-thread communication is reduced. Since we need to find the lowest index among each thread, this can be efficiently achieved via efficient block-level primitives. In contrast to ordinary subtraction, which has to keep track of all carries and perform multiple scans, this approach allows us to only do a single reduction using `atomicMin`, which is highly optimized on modern hardware. Once the lowest relevant non-zero index

is identified, we simply subtract one from all digits between n and the index at thread-level, using exclusively register memory. The intuition behind this approach is straightforward, since B^n only contains a single non-zero digit at position n, subtracting it from u only requires borrowing once. This approach is illustrated in Figure 16.

```
template<typename Base, uint32_t Q>
__device__ inline void
sub( typename Base::uint_t u[Q]
   , uint32_t bpow
   , volatile typename Base::uint_t* sh_mem
) {
    using uint_t = typename Base::uint_t;

    uint32_t tmp = UINT32_MAX;
    sh_mem[0] = tmp;
    __syncthreads();

    // find lowest non-zero digit with index >= bpow
    #pragma unroll
    for (int i = 0; i < Q; i++) {
        int rev_i = Q - i - 1;
        int idx = Q * threadIdx.x + rev_i;
        if (u[rev_i] != 0 && idx >= bpow) {
            tmp = idx;
        }
    }
    atomicMin((uint32_t*)sh_mem, tmp);
    __syncthreads();

    // subtract one from all digits between bpow and the computed index
    uint32_t ind = sh_mem[0];
    #pragma unroll
    for (int i = 0; i < Q; i++) {
        uint32_t idx = Q * threadIdx.x + i;
        if (idx >= bpow && idx <= ind) {
            u[i] -= 1;
        }
    }
}
```

Listing 10: Efficient big integer subtraction with divisor specialized to B^n.

8.7 Runtime Analysis

Contrary to Section 6.6 which looked at the asymptotic complexity of the algorithm, this section focuses on the exact amount of work performed by our CUDA implementation. This analysis is then used as the basis for selecting a suitable performance metric during benchmarking. And as the runtime of division is tightly tied to the cost of multiplication, we base the runtime analysis on the number of full multiplications performed. By full multiplication, we mean a multiplication performed on all M digits. We assume that $Q = 4$, since our implementation is optimized for this value, meaning that there are $\frac{M}{4}$ threads in

total per block. Since we do smaller multiplications for sizes up to two times the *blocksize*, a full multiplication will be performed whenever more than $\frac{M}{2}$ digits of the result are required. As showcased in Algorithm 7, each iteration of the whole shifted inverse algorithm performs two multiplications, one inside Step and another one inside PowDiff.

Number of Full Multiplications of PowDiff As part of PowDiff we apply the close products strategy whenever $L < h'$. When calculating

$$L = \text{prec}_B v + \text{prec}_B w - \ell + 1,$$

the terms $\text{prec}_B w$ and l roughly cancel out, leaving $\text{prec}_B v$. Thus L is based on the precision of v. When employing divisor prefixes, we shift v by $k - 2l$, leaving $\text{prec}_B v \approx \min(2l, k)$. On the other hand we have $h' \approx 3l$. This implies that h' grows faster than L, meaning that the algorithm will use close products whenever the iteration count is sufficiently large. Therefore it suffices to analyze the multiplication behavior of MultMod inside PowDiff when determining the number of full size multiplications. Multmod computes $v \cdot w \bmod B^L$, which means that only the first L digits of the product are effectively used. Since L is roughly given by $\text{prec}_B v \approx \min(2l, k)$, the maximum multiplication is less than M/2 when $k \leq h/2$, since $h < M$. As a result no full multiplication is performed.

When $k > h/2$, and a full multiplication is performed, we know that $h/2 < \text{prec}_B k_i \leq 2l_i$. Also, we know that the loop terminates when $l_i \geq h - k$. Given that $k > h/2$ we have $h - k \leq \lfloor h/2 \rfloor$, which means that the algorithm is guaranteed to terminate whenever l_i reaches $h/2$. Since $l_{i+1} = 2l_i - 1$, in the next iteration we have:

$$h/2 < 2l_i \quad \Rightarrow \quad h/2 - 1 < l_{i+1}$$

Note that l_{i+1} is one of reaching $h/2$. This means that we in most cases perform one full multiplications inside PowDiff, and in rare cases we need to perform two.

Number of Full Multiplications of Step Step computes the product between w and v, which are approximately of sizes l and $2l$. Thus, whenever a full multiplication is done we know that $h/2 \leq M/2 < 3l_i$. Again we consider 2 cases, $k \leq h/2$ and $k > h/2$. When $k > h/2$ we can apply the same logic as for Powdiff. After the next two iterations we have:

$$h/2 < 3l_i \;\Rightarrow\; \frac{2}{3}h/2 - 1 < l_{i+1} \;\Rightarrow\; \frac{4}{3}h/2 - 3 < l_{i+2} \;\Rightarrow\; h/2 < l_{i+2} \text{ for } h \geq 18$$

As before, we know that the algorithm is guaranteed to terminate whenever l_i reaches $h/2$, resulting in at most two full multiplications when $h \geq 18$. For $k \leq h/2$ we consider the worst case, which occurs when $3l$ is exactly equal to $h/2$. This means that w is of size $l = h/6$ while v is of size $2l = h/3$. We are are guaranteed that the loop stops at $h - k \leq l$ or $2/3h \leq l$ since $v \geq h/3$. If we were to double l_i (and thus the precision of w) at each iteration, we would only require 2 iterations as we would have $l_{i+2} = 2/3h$. However, since we subtract by

one at each iteration, three iterations would in rare cases be required.

Total Number of Full Multiplications In total, we perform at most four full multiplications as part of computing the shifted inverse. Obtaining the quotient additionally requires two multiplications, one of them being of double length, which corresponds to two full multiplications. This yields a minimum of three and a maximum of seven size-M full multiplications for division.

9 Futhark Implementation

Futhark is a high-level functional programming language designed to allow a programmer to easily implement highly parallel programs. Futhark abstracts away concerns associated with low-level languages like hierarchical memory, thread management and coalesced memory accesses. Instead, the programmer only need to conceptualize the algorithm in terms of SOACs, which the compiler then translates into efficient parallel low-level code to be run on GPUs. This results in cleaner code, and doesn't require the domain knowledge of the underlying GPU architecture, in order to produce semantically equivalent programs.

In this section we present our implementation of the division algorithm by [33] for multiple precision integers in Futhark, and where it is applicable compare it to the lower-level implementation CUDA prototype as described in Section 8. The underlying idea behind developing a semantically equivalent program in Futhark is to explore the compiler's ability to produce competitive code in comparison to a highly optimized low-level implementation.

9.1 Futhark's Strengths and Weaknesses

Futhark is a purely functional data-parallel array language, which offers several advantages for implementing algorithms that express a level of parallel behavior. Because the language is purely functional, it's syntax closely resembles that of algorithmic pseudo code, and since there are no side effects, it makes it straight forward to reason for the correctness of programs. Programming parallel constructs in Futhark can easily be expressed in terms of SOACs, which is then compiled to efficient parallel code. Thus the programmer only has to concern themselves about expressing algorithms in terms of array combinators instead of interacting with manually interacting with individual threads.

Futhark relies on advanced compiler optimization techniques in order to produce efficient parallel programs. In particular it uses a technique called *incremental flattening* to transform nested parallelism in programs operating on regular multi-dimensional arrays into efficient GPU-executable code. This technique incrementally maps levels of map-based parallelism to hardware by applying transformations like map fission and map loop interchange. For each discovered map operation, the compiler generates multiple code versions: one that maps the current parallel context to a GPU kernel, another that recursively flattens inner parallelism into an intra-group kernel using shared memory, and additional versions for deeper transformations. These variants are dynamically selected at runtime via autotuned thresholds to ensure optimal performance across different workloads. Futhark employs use of memory optimizations in the form of memory reuse and short-circuiting analysis. Memory reuse refers to the compiler reusing memory buffers once their lifetimes have ended, similar to how registers are reused. Short-circuiting avoids unnecessary memory copies by directly writing results into the target memory space when it's safe to do so. Together, these optimizations help reduce the overall memory footprint of the program. [16,17,30].

The main weakness of Futhark is that it is an inherently high-level language, meaning that the performance of programs is very much reliant on the compiler's ability to generate optimized code. In particular it is not possible to provide low-level optimizations by hand in the same manor as when interacting with a lower level language e.g. C++. One case where these compiler optimizations cause restrictions is when a program exhibits irregular parallelism, which is the case when a nested structure expresses varying sizes of parallelism across different levels. When confronted with this type of parallelism, the incremental flattening technique used by the compiler breaks down and is unable to compile to a parallel backend. Instead this can lead the compiler to sequentialize parts of the code, which negatively impacts the performance [17]. This restriction on irregular parallelism is particularly problematic with regards to the case of division, as the degree of parallelism changes shape during refinement.

9.2 Implementation

Similar to the structure of the CUDA prototype which uses an existing framework for multiplication and addition, our Futhark implementation utilizes these functions from a library developed by [10]. This library implements the same underlying algorithms, from which we use addition, subtraction and classical multiplication. With these we compute the shifted inverse in Futhark results in code very similar to the pseudo code by [33], and can be seen in Listing 11.

One of the main differences between the function and it's equivalent in CUDA C++ is that most of the auxiliary C++ functions worked on mutable arrays with no return value. In comparison since Futhark is without side effects, all branches need to return a value of the same type. Additionally, the incremental flattening performed by the compiler ensures that the resulting kernel maintains intermediate arrays in shared memory while performing bulk operations in registers. Thus when writing the corresponding functions in Futhark we never need to concern ourselves about efficiently mapping intermediate results in and out of different layers of memory. When computing the initial approximation of the shifted inverse, we require the existence of an integer of 4 times the size of a single digit, meaning that we have had to restrict ourselves to only support 16-bit words. This is because Futhark only supports integer arithmetic up to 64 bits. If Futhark offered support for 128-bit integers, the implementation could easily be extended to include 32-bit integers, as this is the the only part of the algorithm that requires quadruple-sized integer values.

```
1  def shinv [m][ipb] (vs: [ipb*(4*m)]u16) (h: i64) (k: i64) : [ipb↩
       *(4*m)]u16 =
2      if k == 0 then
3          quo_single h (vs) (ipb*(4*m)) :> [ipb*(4*m)]u16
4      else if k >= h && !(eqBpow vs h) then
5          vs
6      else if k == h - 1 && vs[k] > u16.highest / 2 then
7          zeroAndSet 1 0 (ipb*(4*m)) :> [ipb*(4*m)]u16
8      else if eqBpow vs k then
9          zeroAndSet 1 (h - k) (ipb*(4*m)) :> [ipb*(4*m)]u16
10     else
```

```
        let l = i64.min k 2
        let V = (u64.u16 vs[k - 2]) | (u64.u16 vs[k - 1]) << 1*16↩
             | (u64.u16 vs[k]) << 2*16
        let b2l = 1u64 << 4*16
        let tmp = (b2l - V) / (V + 1)

        let ws = tabulate (ipb*(4*m)) (\i ->
            if i == 0 then u16.u64 tmp
            else if i == 1 then u16.u64 (tmp >> 16)
            else 0u16 )
        in if h - k <= l then
            shift (h-k-l) ws
        else
            refine vs ws h k l
```

Listing 11: Implementation of the shifted inverse in Futhark

Using the shifted inverse to perform a division of two big integers works largely the same as in CUDA. In order to circumvent an infinite loop during refinement, we perform an initial check to correct the precision of the divisor if k is initially equal to 2. We also perform a full multiplication, meaning that we compute the full product without truncating the overflow. In CUDA this meant implementing, a special version of multiplication without truncation, where we manage register usage and shared memory explicitly. In Futhark much of this is delegated to the compiler, instead we can simply pad the shifted inverse and dividend arrays with leading zeroes. The full product is then left shifted and truncated to the first M entries, obtaining the quotient in the correct array size. The rest of the function simply corrugates for the error $\delta \in \{0, 1\}$, resulting in the following code as seen in Listing 12.

```
def div [m][ipb] (us: [ipb*(4*m)]u16) (vs: [ipb*(4*m)]u16) : ([↩
    ipb*(4*m)]u16, [ipb*(4*m)]u16) =
    let h = prec us
    let k = (prec vs) - 1
    let (kIsOne, us, vs, h, k) =
        if k == 1 then
            let h = h + 1
            let k = k + 1
            let us = shift 1 us
            let vs = shift 1 vs
            in (true, us, vs, h, k)
        else
            (false, us, vs, h, k)
    let quo =
        let m = m * 2
        let quo_padded = ((shinv vs h k) ++ (replicate (ipb*(4*(m↩
            /2))) 0u16)) :> [ipb * (4 * m)]u16
        let us_padded = (us ++ (replicate (ipb*(4*(m/2))) 0u16)) ↩
            :> [ipb * (4 * m)]u16
        let mul_res = convMulV3 quo_padded us_padded
        let mul_shifted = shift (-h) mul_res
        let res = take (ipb*(4*(m/2))) mul_shifted
        in res
    let quo = quo :> [ipb * (4 * m)]u16
    let (rem, _) = convMulV2 vs quo
        |> bsub us
```

```
    let (quo, rem) =
        if not (lt rem vs) then
            let quo = badd1u16 quo
            let (rem, _) = bsub rem vs
            in (quo, rem)
        else
            (quo, rem)
    let rem =
        if kIsOne then
            shift (-1) rem
        else
            rem
    in (quo, rem)
```

Listing 12: The complete division algorithm in Futhark

9.2.1 Refinement

As part of the process of refining the initial approximation when calculating the shifted inverse, Watt [33] presents multiple subroutines. The routine used as part of the CUDA prototype is the third and most efficient variant, which optimizes the total work of the algorithm by using differently sized multiplications during each step. However, due to the Futhark compiler's limitations with regards to irregular parallelism, it is not possible to generate this pattern routine in parallel code [17]. Instead we perform full multiplications at each step of refinement, essentially doing an amount of work corresponding using the first refinement routine. Using these fixed-size multiplications instead of varying sizes results in an overall runtime of $O(n^2 \cdot \log(h-k))$ instead of $O(n^2)$, meaning that we expect the performance of the Futhark implementation to be slower than that of CUDA. The resulting implementation of the refine routine can be seen in Listing 18.

```
def refine [m][ipb] (vs: [ipb*(4*m)]u16) (ws: [ipb*(4*m)]u16) (h:↩
    i64) (k: i64) (l: i64) : [ipb*(4*m)]u16 =
    let ws = shift 2 ws
    let (ws, _, _) = loop (ws, l, i) = (ws, l, 0)
        while h - k > (l + 1) do
            let n = i64.min (h - k + 1 - l) l
            let s = i64.max 0 (k - 2 * l + 1 - 2)
            let vs = shift (-s) vs
            let tmp = step vs ws (k + l + n - s + 2) l n
            let ws = shift (-1) tmp
            let l = l + n - 1
            let i = i + 1
            in (ws, l, i)
    in shift (-2) ws
```

Listing 13: Refinement routine without irregular parallelism.

10 Validation & Benchmarking

This section presents the results of our two implementations of the division algorithm by Watt [33] in CUDA C++ and Futhark. We start by arguing for the correctness of our implementations, before delving into the performance of each one in detail. The performance will be upheld against the state of the art Cooperative Groups Big Numbers (CGBN) library [26] authored by NVlabs, which is a highly optimized framework for performing arithmetic on multiple precision integers written in CUDA.

10.1 Correctness

In order to effectively measure the performance of our implementations it is essential that we verify their correctness, as if they produce incorrect results then the underlying work may not be representative of the algorithm. This is especially important in the field of arbitrary precision integer arithmetic as is the case for Algorithm 6.5, as the entire idea behind not leaving the domain of integers is to preserve precision during computation.

As we argue for the correctness of the algorithm and our refinements in Section 6 and Section 7 respectively, we will not delve further into this in this section. Instead, we will focus solely on the correctness of our implementations. To do this we validate our efficient CUDA prototype and Futhark implementation against the GNU Multi-Precision Library (GMP) which implements efficient arithmetic operations sequentially on the CPU. Thus, for the purposes of validation, it is assumed the GMP functions are correct.

The CUDA framework allows for the division kernel to be launched with over an arbitrary number of problem instances, which are then scheduled for parallel execution on the GPU. As part of our correctness testing, we initialized each problem instance with randomly generated integers and a randomly generated precision (number of non-zero digits). Each set of integers is then evaluated against the GMP division implementation in order to compare the results. To ensure broad coverage of potential edge cases, we run this setup for 1000 iterations, which provides a substantial amount of variation across inputs. In addition to this randomized testing approach, we also performed manual testing of known edge cases, including the over-approximation of the whole shifted inverse showcased in Section 7.3 We similarly ensured that all fast cases were properly handled. All tests were consistent with the GMP outputs, providing strong evidence for the correctness of our implementation.

We did not find it possible to automatically validate against GMP in Futhark, as there is no port of the library in the language. Instead, we manually verified the correctness of the implementation against GMP for a fixed set of inputs. These tests confirmed that the Futhark implementation, when compiled to parallel CUDA code, produced results identical to GMP. However, for some randomly generated inputs we found discrepancies between the Futhark output and the equivalent from GMP, confirming the existence of bugs in our implementation. Therefore, we consider the Futhark implementation to be partially validating, though a closer

inspection shows that its control flow correctly follows the algorithm for the generated test inputs. Therefore we are confident that the implementation performs an amount of work, corresponding to that of a fully correct implementation. For this reason we include the Futhark benchmark results below, as the performance is still representative of the its general capabilities on this type of problem.

10.2 Performances Metrics

When performing classical multiplication, the number of computations heavily outnumbers the memory transactions, indicating that the operation is compute bound. Since the cost of our algorithm mainly comes from multiplication, we evaluate the performance using a normalized metric which reflects the number of algorithmic operations per second. This is essentially derived by estimating the number of operations based on the known asymptotic complexity and dividing by the measured runtime. This approach is commonly used for benchmarking algorithms (e.g., previous multiplication implementations [30]), as it normalizes the workload, allowing more meaningful comparisons across algorithms, datasets, and hardware platforms.

A natural choice is to base this metric around the number of `uint_t` operations. But as we work with `uint16_t` in Futhark, and both `uint32_t` and `uint64_t` in CUDA, this would create a discrepancy between the measures. Therefore, we further normalize the result to be consistent with `uint32_t`. As a result we use `Gu32ops/s`, which represents the number of giga-32-bit operations per second. The definition follows:

Definition 3. *(division performance metric)*

We measure the number of operations for division based on classical multiplication as follows, where $c, NumInsts, M \in \mathbb{Z}^+$ *represent a constant reflecting the number of full multiplications, the number of problem instances, and the number of big integer digits respectively:*

$$\texttt{Gu32ops} = c \cdot NumInsts \cdot m^2, \qquad m = \frac{M \cdot \texttt{sizeof(uint)}}{4}$$

In our case we fix $c = 3$ *to reflect the minimum number of full multiplications performed during refinement in the* CUDA *version, as discussed in Section 8.7.*

The m in Definition 3 ends up representing the number of digits in the big integer, normalized to 32-bit numbers, which is squared due to the $O(m^2)$ asymptotic complexity of the classical multiplication used by the division implementation. The constant c is based off the number of full multiplications performed as part of refinement. This was found to be between three and seven full multiplications are performed when no fast case is taken. For this reason we use a conservative approach, and set the constant to $c = 3$ during performance testing to effectively reflect the amount of work being performed.

10.3 Benchmark Setup

The benchmarking was performed using an Nvidia A100 GPU, which has 6912 cores, a peak global memory bandwidth of 1,555 GB/sec and `FP32` peak performance of 19.5 TFlops. The CPU in the system is a AMD EPYC 7352, with 24 processing cores running at 2.3 GHz, with 32KB data and 32KB instruciton cache (per core), 512KB of l2 cache (per core) and 128MB of l3 cahce (shared), and a peak memory bandwidth of 204.8GB/s. The system has 503GB of memory, and the operating system is Red Hat Enterprise Linux 8.10.

The **CUDA** benchmarking setup is built on top of an existing framework initiated by [30], which has been extended to include corresponding implementations of division and GCD (see Section 10.4.2). Each problem instance is initialized with randomly generated integers. The precision of u is fixed at $M - 2$ (accounting for the two guard digits in `Refine`), while the precision of v is randomly selected between 2 and M/2. This configuration ensures that the refinement loop always performs the maximum number of iterations.

The framework requires parameters to be stated explicitly during setup such as the integer word size, the sequentialization factor Q and the total number of runs. For the word size we test for both the 32-bit and 64-bit implementations, however we expect the 64-bit word size to prove superior, as this has been previously shown to be the case for multiplication [30]. For similar reasons we set the sequentialization factor Q to default to 4 (as discussed in Section 8.1), as larger values have shown to prove inferior results. However, when testing the maximum sized integer of 2^{18} bits, the 32-bit **CUDA** version requires that we set $Q = 8$ to run. This occurs due to the maximal number of threads of a **CUDA** block being 1024. For a 2^{18} bit integer and a word size of 32, we have $M = 2^{18}/32 = 8192$, meaning that Q must be at least 8 in order to fully distribute the big integer between all threads. Lastly, as in [30], we average over 25 runs to eliminate statistical variance.

Benchmarking is natively supported in Futhark, which runs a specific entry point with randomized inputs until the performance has been determined within a 95% confidence interval across a minimum of 10 runs. However, Futhark benchmarks the entire entry point including our initial setup, responsible for setting the precision of the divisor v, which we ultimately disregard, as it is asymptotically negligeable compared to the algorithm itself. Lastly previous sources [10, 30] have shown that the Futhark benchmarking results can be directly compared with timed **CUDA** kernels, thus we will directly compare our results as well.

In addition to our own implementations we test the performance against both the open source GMP library, and the **CGBN** library by NVlabs. Both libraries abstract away the underlying representation of the integers, and thus its interface provides no notion of word sizes. All benchmarks are performed on a fixed input size of exactly 2^{32} bits, which is decomposed into batches of varying integer sizes, ranging from 2^9 to 2^{18} bits.

Futhark Autotuning As an additional step in optimizing the program, the Futhark compiler provides a native program autotuner, which is a tool designed to automatically improve runtime performance by empirically searching for optimal

values of tunable parameters. Specifically, the autotuner targets threshold parameters, which are compiler-level variables that influence the degree of parallelism of different parts of the code by use of incremental flattening. It performs a series of timed benchmark runs on open entry points with different parameter configurations and generates a tuning file with the best performing settings for a given hardware and input size configuration [23].

Despite the potential benefits of the autotuner, it was not effective in our case. Our implementation of division involves efficient management of dynamically allocated memory as part of nested parallel computations, which likely falls outside the tuning capabilities of the current Futhark autotuner. As such the resulting parameters found during the autotuning process resulted in an identical runtime to the one recorded without the tuned parameters. It known that the autotuner is guaranteed to be near-optimal only when the degree of parallelism conforms with a monotonic property [23]. We speculate that the reason behind the autotuner being ineffective may be due to the large amount of dynamically allocated shared memory required, which then voids this monotonic property. As a result when performance testing the Futhark implementation we test without any particular tuning setup.

10.4 Performance Results

The results of our performance testing are presented in Table 6, where the different platforms have been divided in terms of the total integer size in bits. The first column is the integer size in bits while the second is the number of instances performed. Cells filled with — denote that kernel launch failed due to a compile or runtime error. Please note that the total number of processed bits across all results remains constant i.e.

$$\texttt{num bits} \cdot \texttt{num insts} = 2^{32}$$

Looking at Table 6 multiple observations stand out. Firstly, only our **CUDA** implementation and the GMP framework support integers in the entire range of $\{2^n \mid n \in \mathbb{Z},\ 9 \leq n \leq 18\}$ bits. During testing we found that the **CGBN** division implementation produced a compile-time error which has been previously documented for integers greater than 2^{15} bits. More interestingly, we found that our Futhark implementation failed to produce any results for integers larger than 2^{13} bits, as it failed during runtime with an error message stating:

```
‘bytes of memory exceeds device limit‘
```

We speculate this to be due to the compiler's capability to efficiently manage dynamically allocated shared memory when compiled to the **CUDA** backend. As we specifically had to increase the maximum amount for the **CUDA** implementation. In this context, we would like to further study the compiled intermediate code, in order to understand the reason behind the error. Unfortunately it was not possible to use Futhark's profiler to generate this kernel, as it fails on the entry point as well. We therefore leave the reason for the error, and understanding of it to future works.

Num Bits	Num Insts	CUDA 32-bit	CUDA 64-bit	Futhark 16-bit	CGBN	GMP
2^{18}	2^{14}	725	2.351	—	—	208
2^{17}	2^{15}	1.042	2.338	—	—	166
2^{16}	2^{16}	1.053	2.083	—	—	99
2^{15}	2^{17}	940	1.383	—	5.031	110
2^{14}	2^{18}	792	725	—	4.658	98
2^{13}	2^{19}	455	538	20	3.967	41
2^{12}	2^{20}	318	162	15	3.024	22
2^{11}	2^{21}	109	56	16	2.443	12
2^{10}	2^{22}	36	16	9	1.712	5
2^{9}	2^{23}	11	13	6	812	2

Table 3: Performance of division measured in `Gu32ops/sec` (higher is better).

By examining the supported integer sizes, we observe that the performance of our **CUDA** implementations steadily increases with the integer size. This is partially due to the fact that, for smaller bit sizes, the runtime of the multiplications does not dominate the total runtime as much. As a result, the relative overhead from other arithmetic operations and comparisons becomes more significant, limiting overall performance.

For sizes above 2^{14} bits we find that the 64-bit variant consistently outperforms the 32-bit one. This behavior is expected, as it was recorded that the underlying multiplication kernel, which our division is based upon, showed a ~ 1.6 increase in speed for 64 bits. However, for integer sizes smaller than 2^{15} bits, the 32-bit version generally outperformed the 64-bit one. We expect that this is manly due to the differences in occupancy, i.e. how well the **CUDA** is able to utilize all threads and warps in each streaming multiprocessor (SM), which is essentially a byproduct of only instantiating one instance per block. For the smallest tested integers of 2^9 bits, the number of 64-bit digits is $M = 2^9/64 = 8$, and since we have the sequentialization factor Q set to 4, we only instantiate 2 threads per block. But because **CUDA** allocates threads in warps, a full warp (32 threads) is reserved even though 30 threads remain idle throughout the entire computation. In contrast, when using 32-bit digits, the number of digits M doubles, which in turn doubles the number of threads in each block, thus increasing the thread occupancy in each warp. For 64-bit words, all 32 threads in a warp are used whenever

$$\texttt{numBits} \geq 4 \cdot 64 \cdot 32 = 2^{13},$$

which is reflected in our results as well.

Moreover, each SM is restricted to a maximum of 32 resident blocks and a total of 1536 active threads. Since $32 \cdot 48 = 1536$, whenever we instantiate blocks with less than 48 threads, we are not fully using all of the potential parallellism capabilities of each SM. The number of threads initilaized in a block is given by:

$$\texttt{blockSize} = \frac{\texttt{numBits}}{\texttt{WordSize} \cdot Q}$$

For 64-bit digits a block size of 48 results in:

$$\texttt{numBits} \geq 48 \cdot 64 \cdot 4 \quad \Rightarrow \quad \texttt{numBits} \geq 12288$$

Which corresponds to at least 2^{14} `numBits` for the SM to be able to allocate all its available threads. For 32-bit integers it is 2^{13} instead. We were surprised to find that for 2^{14} bits, the 32-bit configuration outperformed the 64-bit one, which slightly contradicts our earlier arguments. A possible explanation is that the 32-bit version uses four warps per block instead of two, which enables more efficient warp scheduling and better potential for latency hiding.

Comparing the runtime across implementations we see that the Futhark version is much slower, and in particular the runtime scales much worse, resulting in performance being orders of magnitude worse than the **CUDA** variant, for the larger integers. This was the expected behavior for three main reasons. Partially because the Futhark implementation uses 16-bit integers as the internal digit representation, and since the 64-bit **CUDA** implementation proved superior compared to 32-bit, we expect the 16-bit variant to perform even worse. However, the main reason for the poor scaling is due to the Futhark implementation being asymptotically slower than the **CUDA** version. As showcased in Section 9.2.1, the Futhark implementation performs work asymptotically equivalent to the `Refine1` subroutine instead of the improved `Refine3`, which incurs an asymptotic penalty by a factor of $O(\log(h-k))$ as described in Section 6.6. This is an inherent problem related to the shorter iterates strategy used by the improved refinement routine, which in practice involves nested irregular parallelism, something the current Futhark compiler is unable to deal with. Lastly, as Futhark is a higher-level language, it prioritizes simplicity and generalizability over fine-grained control of threads and blocks, which results in certain limitations in compiler flexibility and performance.

The **CGBN** library outperforms both our **CUDA** and Futhark implementations for the supported integers in the bit-range 2^{11} to 2^{15}. This was somewhat anticipated, as **CGBN** is highly optimized for warp-level operations meaning that it utilizes specific instructions that work in lockstep on one warp of threads at a time, which allows for very low latency communication between threads. In comparison, our **CUDA** implementation performs most operations on a block-level basis (with a few exceptions, see Section 8.5) which is less efficient but does not rely on specialized hardware instructions. Likewise, the Futhark compiler is likely not able to generate **CUDA** code featuring these advanced instructions either, due to them being highly specific.

Additionally, we expect the **CGBN** library to achieve higher occupancy for smaller integer sizes compared to our **CUDA** implementation, by processing multiple problem instances per block. However, as the main objective of this thesis was to optimize performance for the largest integer sizes that could fit in shared memory (i.e. integers up to 2^{18} bits) we disregarded this optimization.

Lastly, since most division algorithms are inherently parameterized over multiplication, we suspect that **CGBN** library exploits this by using dynamic thresholds to switch between different multiplication strategies, such as classical, Karatsuba

and FFT-based methods. As our implementation always resorts to classical multiplication, we expected GCBN to be asymptotic superior, especially for bigger input sizes. For 2^{15}-bit input sizes our **CUDA** implementation is approximately 3 times slower, a gap that would shrink if additional optimizations were applied, such as clipped products and dynamic multiplication switching.

GMP performs significantly worse than both **CUDA** and **CGBN** versions, being about 10 times slower on average than the best performing **CUDA** implementation. This comes at no surprise, as GMP is a sequential CPU-based framework. It is only slightly faster than our Futhark implementation for medium sized integers (2^{12} and 2^{13} bits), which we attribute to Futhark performing asymptotically more work. This is both due to the fact that the Futhark implementation always performs full multiplications, and that GMP likely switches between different multiplication algorithms based off integer size, similar to **CGBN**.

The total runtimes for all implementations when not normalized to `Gu32ops/s` can be found in Appendix 7.

10.4.1 Performance of Division Relative to Multiplication

To examine the overhead introduced by our **CUDA** division implementation, as well as to reason about how **CGBN** might perform if input sizes up to 2^{18} bits were supported, we compare the performance of division relative to multiplication for both our 64-bit **CUDA** version and **CGBN**. The multiplication performance was normalized using the metric defined in Definition 3, without the constant c. We were able to produce previously undocumented results for multiplication on 2^{18} bit integers [30], by ensuring that dynamic shared memory was allocated for storing the big integers in each block. These results are shown in Table 4.

Num Bits	Num Insts	CUDA MULT		CGBN MULT		CUDA DIV		CGBN DIV	
		μs	`Gu32ops/sec`	μs	`Gu32ops/sec`	μs	`Gu32ops/sec`	μs	`Gu32ops/sec`
2^{18}	2^{14}	271.205	4.054	7.354.041	149	1.402.839	2.351	—	—
2^{17}	2^{15}	130.994	4.196	488.289	1.125	705.266	2.338	—	—
2^{16}	2^{16}	69.059	3.980	66.306	4.145	395.807	2.083	—	—
2^{15}	2^{17}	38.048	3.612	33.840	4.061	297.871	1.383	81.950	5.031
2^{14}	2^{18}	24.619	2.791	16.287	4.219	284.117	725	44.251	4.658
2^{13}	2^{19}	19.198	1.789	9.088	3.780	191.265	538	25.982	3.967
2^{12}	2^{20}	9.834	1.746	5.047	3.403	316.834	162	17.043	3.024
2^{11}	2^{21}	6.096	1.409	2.977	2.885	457.797	56	10.547	2.443
2^{10}	2^{22}	7.504	572	1.945	1.104	757.187	16	7524	1.712
2^{9}	2^{23}	3.845	558	1.624	2.644	482.085	13	7931	812

Table 4: Performance of 64-bit **CUDA** and **CGBN** division and multiplication measured in μs and `Gu32ops/sec`

As analyzed in Section 8.7, our **CUDA** implementation performs between 3 and 7 full multiplications depending on the precision of the inputs. In practice, whenever we do the maximum iterations in refinement (which we ensured during benchmarking) we expect an average of 5 full multiplications. For the lower bit-sizes however, the runtime differences between multiplication and division is far greater than 5×, which is linked to **CUDA** not being able to fully utilize its available hardware, and due to the previously established overheads. For larger sizes, the runtimes seem to stabilize around 5 times the runtime of a single multiplication.

The CGBN division is pretty consistently 2,5× slower than multiplication across all input sizes. The specifics of how this is achieved is unclear, as the available documentation is limited. For instance, we are not aware whether CGBN uses a variation of Newton's iteration as for our case, or if it employs an alternative strategy. Assuming that the 2,5× slowdown factor would still persist for integer sizes above 2^{15} bits, a linear extrapolation based on the CGBN multiplication would reveal a slower runtime than our CUDA implementation for integers of 2^{17} bits and above. Although these projections are only speculatives, and should not be interpreted as a definitive.

10.4.2 Performance of Greatest Common Divisor

We extended the division implementation to also support computation of the Greatest Common Divisor (GCD), reflecting a more realistic use case for big integer arithmetic. This was achieved using the Euclidean algorithm, which works by iteratively computing the remainder of two integers, replacing the inputs with the divisor and remainder at each step. The process continues until the remainder is zero, which means that the greatest common divisor has been found. We argue that our GCD implementation is correct, as all tests validate, and it additionally validates against GMP.

According to Knuth [20], the expected number of divisions performed by the Euclidean Algorithm is approximately:

$$\mathbb{E}[steps] \approx \frac{12 \ln 2}{\pi^2} \ln N \approx 0.843 \ln N$$

Expressed in terms of bits, i.e. $N = 2^n$, we get:

$$\mathbb{E}[steps] \approx 0.843 \ln 2^n \approx 0.58n$$

This implies that the number of divisions in the Euclidean algorithm grows linear with the input bit length. Thus, when applying GCD to integers of 2^{18} bits, we would need $2^{18} \cdot 0.58 \approx 152.043$ iterations (and thus divisions) on average.

Due to the extremely high computational demand, we have decreased the total works of the input from 2^{32} to 2^{28} bits, which as before is split into batches of varying integer sizes, ranging from 2^9 to 2^{18} bits.

We denote the number of giga-32-bit unit operations per second based on $\mathbb{E}[steps] \approx 0.58n$ as:

$$\texttt{Gu32ops} = 0.58 \cdot NumBits \cdot \text{Runtime of division},$$

although this is an overestimation in practice, as we perform smaller and smaller divisions as the algorithm progresses.

Analyzing the results in Table 5, we find that our GCD implementation performs orders of magnitude slower than both the GMP and CGBN libraries. When running on smaller inputs, the performance is initially comparable to GMP, however as we increase the size of the inputs we are steadily outperformed. We expect the difference in magnitudes arises from the libraries employing a more optimized

version of the Euclidean method or an entirely different algorithm. In addition, if these algorithms involve multiplication, they likely avoid the classical $O(n^2)$ approach for unsuitable input sizes, achieving better asymptotical complexity relative to the number of bits. Furthermore, during GCD computation, the sizes of the divisions naturally decrease over time, leading to progressively smaller operations. In our implementation, however, only the multiplications scale down accordingly, while some other functions continue to operate on the full size of the initial input, irrespective of any leading zeros. This shortcoming, combined with the fact that we already perform poorly on small numbers, produces a significant amount of overhead.

Num Bits	Num Insts	CUDA 64-bit	CGBN	GMP
2^{18}	2^{10}	—	—	2.417.963
2^{17}	2^{11}	—	—	820.691
2^{16}	2^{12}	2.889	—	287.257
2^{15}	2^{13}	2.519	5.971.960	102.729
2^{14}	2^{14}	1.920	3.073.873	36.890
2^{13}	2^{15}	1.228	1.208.655	12.030
2^{12}	2^{16}	397	390.020	3.631
2^{11}	2^{17}	138	138.063	1.077
2^{10}	2^{18}	43	50.990	325
2^{9}	2^{19}	30	15.626	110

Table 5: Performance of GCD measured in `Gu32ops/sec` (higher is better).

10.5 Performance on Other Hardware

The parallel performance results have all been recorded using NVIDIA A100 GPUs, as this is the hardware that has been available to us. However both of our implementations alongside the **CGBN** library should easily be able to be deployed directly to any **CUDA** GPU with minimal issues. The **CUDA** prototype may possibly be hindered by the amount of shared memory and registers per thread of smaller GPUs, restricting the sizes for which it is possible to perform a division.

We expect the performance of our implementations to be invariant of the underlying hardware because no hardware-specific techniques have been used. As such we predict that the performance should remain consistent when normalized by the peak compute performance of the underlying hardware. Regardless of this, it could prove interesting to test the performance across different hardware to ensure this claim.

11 Conclusion

We have shown how to implement an efficient exact division, based on the concept of a whole shifted inverse from [33]. We provided a series of refinements to said algorithm, addressing previously undocumented edge cases. By exploiting temporal reuse of fast scratchpad memory, our approach enables efficient division of integers up to approximately $\sim$ 250.000 bits, which we, to our knowledge, are the only ones to have achieved in parallel. Additionally, we have conclude that there exists several compiler limitations in Futhark, in particular related to irregular nested parallelism, which limits the ability of performant high-level code, when dealing with complex problems.

The implementations are tested for performance against the GMP and **CGBN** libraries which represent state-of-the-art performance in the field of multiple precision arithmetic for sequential and parallel computing respectively. Although not managing to outperform **CGBN** for input of sizes 2^{15} and below, our main advantage lies in providing a parallel implementation capable of handling integers as large as 2^{18} bits.

11.1 Future Work

As our division inherits the asymptotic behavior of the underlying multiplication, future work could explore exactly when and how different multiplication strategies offer performance advantages. By analyzing the overheads and asymptotic behaviour of different multiplications at different input sizes, they could be carefully applied at different stages of the algorithm, to potentially improve overall efficiency. For example by applying an asymptotically faster FFT-based method for the biggest multiplications, which was shown to outperform classical multiplication for sizes 2^{16} and over in [30].

Additionally, as our results show, the restriction of only computing one problem instance per **CUDA** block of threads severely impacts the performance on small and medium sized integers. As a future addition, the implementations can be made more competitive for these input sizes by allowing multiple divisions to be computed within a single block. However, as this would introduce nested-irregular parallelism, careful flattening strategies would need to be applied, a non-trivial challenge due to input-dependent branching.

Lastly, one could incorporate the clipped products strategy as described in [25] into the algorithm, by only computing the required section of the product, thereby reducing the overall workload and increasing performance.

References

[1] Gnu multiple precision arithmetic library. `https://gmplib.org/manual/index`. [Accessed 12-05-2025].

[2] Haskell wiki. `https://wiki.haskell.org/Fold`. [Accessed 30-04-2025].

[3] Using cuda warp-level primitives. `https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/`. [Accessed 20-05-2025].

[4] 1. Introduction; CUDA C++ Programming Guide — docs.nvidia.com. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model`, 2016. [Accessed 25-03-2025].

[5] Hamdy Abdelkhalik, Yehia Arafa, Nandakishore Santhi, and Abdel-Hameed Badawy. Demystifying the nvidia ampere architecture through microbenchmarking and instruction-level analysis. 2022.

[6] Richard Ansorge. *Programming in Parallel with CUDA: A Practical Guide*. Cambridge University Press, 2022.

[7] Hovhannes Bantikyan. Big integer multiplication with cuda fft(cufft) library. *International Journal of Innovative Research in Computer and Communication Engineering*, 2:6317–6325, 2014.

[8] R. S. Bird. An introduction to the theory of lists. In *Proceedings of the NATO Advanced Study Institute on Logic of Programming and Calculi of Discrete Design*, page 5–42, Berlin, Heidelberg, 1987. Springer-Verlag.

[9] G.E. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, 38(11):1526–1538, 1989.

[10] Thorbjørn B. Bringgaard. Efficient big integer arithmetic using gpgpu, 2024.

[11] Stephen A. Cook and Stål O. Aanderaa. On the minimum computation time of functions. *Transactions of the American Mathematical Society*, 142:291–314, 1969.

[12] Michael Garland Duane Merrill. Single-pass parallel prefix scan with decoupled look-back. 2016.

[13] Niall Emmart. *A Study of High Performance Multiple Precision Arithmetic on Graphics Processing Units*. PhD thesis, University of Massachusetts, 2018.

[14] Niall Emmart and Charles Weems. Parallel multiple precision division by a single precision divisor. pages 1–9, 2011.

[15] David Harvey and Joris van der Hoeven. Integer multiplication in time O(n log n). *Annals of Mathematics*, March 2021.

[16] 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. pages 556–571, 2017.

[17] Troels Henriksen, Frederik Thorøe, Martin Elsman, and Cosmin Oancea. Incremental flattening for nested data parallelism. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, pages 53–67, New York, NY, USA, 2019. ACM.

[18] Mioara "Joldes, Jean-Michel Muller, Valentina Popescu, and Warwick" Tucker. "campary: Cuda multiple precision arithmetic library and applications". pages "232–240", "2016".

[19] Anatoly Karatsuba and Yu. Ofman. Multiplication of many-digital numbers by automatic computers. *Proceedings of the USSR Academy of Sciences*, 145:293–294, 1962. English translation in Physics-Doklady, 7 (1963), pp. 595–596.

[20] Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, Boston, 3rd edition, 1997.

[21] Walter Restelli-Nielsen Kristian Olesen, Amar Topalovic. Multiple-precision integer arithmetic. Master's thesis, University of Copenhagen, 2022.

[22] Mian Lu, Bingsheng He, and Qiong Luo. Supporting extended precision on graphics processors. In *Proceedings of the Sixth International Workshop on Data Management on New Hardware*, DaMoN '10, pages 19–26, New York, NY, USA, 2010. ACM.

[23] Philip Munksgaard, Svend Lund Breddam, Troels Henriksen, Fabian Cristian Gieseke, and Cosmin Oancea. Dataset sensitive autotuning of multi-versioned code based on monotonic properties. In Viktória Zsók and John Hughes, editors, *Trends in Functional Programming*, pages 3–23, Cham, 2021. Springer International Publishing.

[24] Arthur C. Norman and Stephen M. Watt. Computing clipped products, 2024.

[25] Arthur C. Norman and Stephen M. Watt. Computing clipped products. 2024. Unpublished manuscript.

[26] NVlabs. Cooperative groups big numbers (cgbn). `https://github.com/NVlabs/CGBN`. [Accessed 20-05-2025].

[27] Cosmin E. Oancea. Lecture notes for the software track of the pmph course. 1(1), 2018.

[28] Cosmin E. Oancea. Demonstrating locality of reference on multi-cores and gpus, 2024. September 2024 PMPH Lecture Slides.

[29] Cosmin E. Oancea. Loop parallelism i, 2024. September 2024 PMPH Lecture Slides.

[30] Cosmin E. Oancea and Stephen M. Watt. Gpu implementations for midsize integer addition and multiplication, 2024.

[31] Arnold Schönhage and Volker Strassen. Schnelle multiplikation großer zahlen. *Computing*, 7:281–292, 1971.

[32] Charles van Loan. *Computational Frameworks for the Fast Fourier Transform.* SIAM, 1992.

[33] Stephen M. Watt. Efficient generic quotients using exact arithmetic, 2023.

Declaration of using generative AI tools (for students)

☒ **I/we have used generative AI as an aid/tool** *(please tick)*

☐ **I/we have <u>NOT</u> used generative AI as an aid/tool** *(please tick)*

If generative AI is permitted in the exam, but you haven't used it in your exam paper, you just need to tick the box stating that you have not used GAI. You don't have to fill in the rest.

List which GAI tools you have used and include the link to the platform (if possible):

ChatGPT [https://chatgpt.com/]

Describe how generative AI has been used in the exam paper:

1) *Purpose (what did you use the tool for?)*
   We used generative AI exclusively as a helping tool during proof-reading

2) *Work phase (when in the process did you use GAI?)*
   Final stages.

3) *What did you do with the output? (including any editing of or continued work on the output)*
   AI suggestions were carefully reviewed, and the generel ideas were incorporated when restructuring sentences. No AI-generated content was directly copy/pasted into the final work.

Please note: Content generated by GAI that is used as a source in the paper requires correct use of quotation marks and source referencing. Read the guidelines from Copenhagen University Library at KUnet.

12 Appendix

Refine1 and Refine2

Algorithm 5: `Refine1` and `Refine2`

```
1:  Function Refine1(v, h, k, w, ℓ):
2:      g ← 1        ▷ Guard digits
3:      h ← h + g
4:      w ← SHIFT_{h−k−l}(w)
5:      for i ← 0; i < ⌈max(log2(h − k − 1), 0)⌉ + 2; i++ do
6:          w ← STEP(h, v, w, 0, ℓ, 0)
7:          if i > 1 then ℓ ← min(2ℓ − 1, h − k)
8:      return SHIFT_{−g}(w)
9:  Function Refine2(v, h, k, w, ℓ):
10:     g ← 2        ▷ Guard digits
11:     w ← SHIFT_g w
12:     for i ← 0; i < ⌈max(log2(h − k − 1), 0)⌉ + 2; i++ do
13:         m ← min(h − k + 1 − ℓ, ℓ)
14:         w ← SHIFT_{−1}(STEP(k + ℓ + m + g, v, w, m, ℓ, g))
15:         if i < 2 then SHIFT_{−m}(w)
16:         else
17:             SHIFT_{−1}(w)
18:             ℓ ← ℓ + m − 1
19:     if (h − k < 2) then SHIFT_{h−k−4}(w)
20:     else SHIFT_{−2}(w)
```

Warp-level Scan Performance

Num Bits	Num Insts	1-Add Shared Mem	6-Add Shared Mem	1-Add Shuffle-based	6-Add Shuffle-based
2^{18}	2^{14}	1047	574	1155	595
2^{17}	2^{15}	1211	823	1280	854
2^{16}	2^{16}	1272	614	1360	895
2^{15}	2^{17}	1272	833	1360	770
2^{14}	2^{18}	1270	588	1355	685
2^{13}	2^{19}	1273	789	1360	1006
2^{12}	2^{20}	1272	629	1359	880
2^{11}	2^{21}	1269	757	1356	920
2^{10}	2^{22}	1272	719	1359	735
2^{9}	2^{23}	1273	774	1355	759

Table 6: Performance of Addition in GB/sec: Shared Memory vs. Shuffle-based Scan. GB/sec are computed by: $3NumInsts \cdot \frac{NumBits}{8}$ as in [30].

Warp-level Reduce

```
template<class OP, class uint_t>
__device__ inline int
reduceBlock( uint_t u
           , volatile uint_t* sh_mem
) {
    int idx = threadIdx.x;
    const unsigned int lane   = idx & (WARP-1);
    const unsigned int warpid = idx >> lgWARP;

    int res = scanIncWarp<OP>(u, lane);

    if (lane == (WARP-1) || idx == blockDim.x - 1) { sh_mem[warpid] = res; }
    __syncthreads();

    if (warpid == 0) {
        res = scanIncWarp<OP>(sh_mem[threadIdx.x], lane);
        if (threadIdx.x == ((blockDim.x + WARP - 1) / WARP) - 1) {
            sh_mem[0] = res;
        }
    }

    __syncthreads();
    return sh_mem[0];
}
```

Listing 14: Block-level reduce implementation.

Performance of Division in Microseconds

Num Bits	Num Insts	CUDA 32-bit	CUDA 64-bit	Futhark 16-bit	CGBN	GMP
2^{18}	2^{14}	4.548.382	1.402.839	—	—	15.810.697
2^{17}	2^{15}	1.581.627	705.266	—	—	9.896.245
2^{16}	2^{16}	782.638	395.807	—	—	8.239.808
2^{15}	2^{17}	438.181	297.871	—	81.950	3.742.565
2^{14}	2^{18}	259.928	284.117	—	44.251	2.091.939
2^{13}	2^{19}	226.079	191.265	5.270.800	25.982	2.509.687
2^{12}	2^{20}	161.592	316.834	3.395.729	17.043	2.236.662
2^{11}	2^{21}	235.262	457.797	1.560.186	10.547	2.159.846
2^{10}	2^{22}	353.791	757.187	1.297.623	—	2.454.895
2^{9}	2^{23}	549.785	482.085	998.890	—	3.035.170

Table 7: Runtime of division measured in microseconds (lower is better).

Cuda Main Functions

```
/**
 * Calculates (a * b) rem B^d
 */
template<typename Base, uint32_t M, uint32_t Q>
__device__ inline void
multMod( volatile typename Base::uint_t* USh
       , volatile typename Base::uint_t* VSh
       , typename Base::uint_t UReg[Q]
       , typename Base::uint_t VReg[Q]
       , int d
       , typename Base::uint_t RReg[Q]
) {
    if (d <= blockDim.x) {
        smallMult<Base, Q>(USh, VSh, UReg, VReg, RReg, d);
    } else if (d <= 2*blockDim.x){
        smallMult2x<Base, Q>(USh, VSh, UReg, VReg, RReg, d);
    } else {
        bmulRegsQ<Base, 1, Q/2>(USh, VSh, UReg, VReg, RReg, M);
        #pragma unroll
        for (int i=0; i < Q; i++) {
            if (Q * threadIdx.x + i >= d) {
                RReg[i] = 0;
            }
        }
    }
}

/**
 * Calculates B^h-v*w
 */
template<typename Base, uint32_t M, uint32_t Q>
__device__ inline bool
powDiff( volatile typename Base::uint_t* USh
       , volatile typename Base::uint_t* VSh
       , typename Base::uint_t VReg[Q]
       , typename Base::uint_t RReg[Q]
       , int h
       , int l
) {
    using uint_t = typename Base::uint_t;

    int vPrec = prec<uint_t, Q>(VReg, (uint32_t*)USh);
    int rPrec = prec<uint_t, Q>(RReg, (uint32_t*)VSh);
    int L = vPrec + rPrec - l + 1;
    bool sign = 1;

    if (vPrec == 0 || rPrec == 0) {
        zeroAndSet<uint_t, Q>(VReg, 1, h);
    } else if (L >= h) {
        __syncthreads();
        int maxMul = vPrec + rPrec;
        if (maxMul <= blockDim.x) {
            smallMult<Base, Q>(USh, VSh, VReg, RReg, VReg, maxMul↩
                );
        } else if (maxMul <= 2*blockDim.x) {
```

```
55              smallMult2x<Base, Q>(USh, VSh, VReg, RReg, VReg, ↩
                    maxMul);
56          } else {
57              bmulRegsQ<Base, 1, Q/2>(USh, VSh, VReg, RReg, VReg, M↩
                    );
58          }
59          __syncthreads();
60          if (lt<uint_t, Q>(VReg, h, (uint32_t*)USh)) {
61              sub<Base, Q>(h, VReg, VSh);
62          } else {
63              sub<Base, Q>(VReg, h, VSh);
64              sign = 0;
65          }
66      } else {
67          __syncthreads();
68          multMod<Base, M, Q>(USh, VSh, VReg, RReg, L, VReg);
69          __syncthreads();
70
71          if (!ez<uint_t, Q>(VReg, USh)) {
72              if (ez<uint_t, Q>(VReg, L-1, VSh)) {
73                  sign = 0;
74              } else {
75                  sub<Base, Q>(L, VReg, &VSh[4]);
76              }
77          }
78      }
79      return sign;
80  }
81
82  /**
83   * Iterative towards an approximation in at most log(M) steps
84   */
85  template<typename Base, uint32_t M, uint32_t Q>
86  __device__ inline void
87  step( volatile typename Base::uint_t* USh
88      , volatile typename Base::uint_t* VSh
89      , int h
90      , typename Base::uint_t VReg[Q]
91      , typename Base::uint_t RReg[Q]
92      , int n
93      , int l
94      , int g
95  ) {
96      using uint_t  = typename Base::uint_t;
97      using ubig_t  = typename Base::ubig_t;
98      using carry_t = typename Base::carry_t;
99
100     bool sign = powDiff<Base, M, Q>(USh, VSh, VReg, RReg, h - n, ↩
            l - g);
101     __syncthreads();
102
103     int rPrec = prec<uint_t, Q>(RReg, (uint32_t*)VSh);
104     int vPrec = prec<uint_t, Q>(VReg, (uint32_t*)USh);
105     __syncthreads();
106     int maxMul = rPrec+vPrec;
107     if (maxMul <= blockDim.x) {
108         smallMult<Base, Q>(USh, VSh, RReg, VReg, VReg, maxMul);
```

```
109     } else if (maxMul <= 2*blockDim.x){
110         smallMult2x<Base, Q>(USh, VSh, RReg, VReg, VReg, maxMul);
111     } else {
112         bmulRegsQ<Base, 1, Q/2>(USh, VSh, RReg, VReg, VReg, M);
113     }
114     __syncthreads();
115 
116     shift<uint_t, M, Q>(n, RReg, USh, RReg);
117     if (sign) {
118         shift<uint_t, M, Q>(2 * n - h, VReg, VSh, VReg);
119         __syncthreads();
120         baddRegs<uint_t, uint_t, carry_t, Q, Base::HIGHEST>((↩
            carry_t*)VSh, RReg, VReg, RReg);
121     } else {
122         bool isZero = ezShift<uint_t, Q>(VReg, 2 * n - h, VSh);
123         __syncthreads();
124         shift<uint_t, M, Q>(2 * n - h, VReg, USh, VReg);
125         if (!isZero) add1<Base, Q>(VReg, VSh);
126         __syncthreads();
127         bsubRegs<uint_t, uint_t, carry_t, Q>((carry_t*)VSh, RReg,↩
              VReg, RReg);
128     }
129 }
130 
131 /**
132  * Refine the approximation of the quotient
133  */
134 template<typename Base, uint32_t M, uint32_t Q>
135 __device__ inline void
136 refine3( volatile typename Base::uint_t* USh
137       , volatile typename Base::uint_t* VSh
138       , typename Base::uint_t VReg[Q]
139       , typename Base::uint_t TReg[Q]
140       , int h
141       , int k
142       , int l
143       , typename Base::uint_t RReg[Q]
144 ) {
145     using uint_t = typename Base::uint_t;
146 
147     shift<uint_t, M, Q>(2, RReg, (uint_t*)USh, RReg);
148 
149     for (int i = 0; i < (int)ceilf(max(log2f(h-k-1), 0.0f)) + 2; ↩
        i++) {
150         int n = min(h - k + 1 - l, l);
151         int s = max(0, k - 2 * l + 1 - 2);
152         shift<uint_t, M, Q>(-s, VReg, VSh, TReg);
153         __syncthreads();
154         step<Base, M, Q>(USh, VSh, k + l + n - s + 2, TReg, RReg,↩
              n, l, 2);
155         __syncthreads();
156         if (i < 2) {
157             shift<uint_t, M, Q>(-n, RReg, USh, RReg);
158         }
159         else {
160             shift<uint_t, M, Q>(-1, RReg, USh, RReg);
161             l = l + n - 1;
```

```
162         }
163     }
164     shift<uint_t, M, Q>((h - k < 2) ? h - k - 4 : -2, RReg, VSh, ↩
            RReg);
165 }
166 
167 /**
168  * Refine the approximation of the quotient
169  */
170 template<typename Base, uint32_t M, uint32_t Q>
171 __device__ inline void
172 refine2( volatile typename Base::uint_t* USh
173       , volatile typename Base::uint_t* VSh
174       , typename Base::uint_t VReg[Q]
175       , typename Base::uint_t TReg[Q]
176       , int h
177       , int k
178       , int l
179       , typename Base::uint_t RReg[Q]
180 ) {
181     using uint_t = typename Base::uint_t;
182 
183     shift<uint_t, M, Q>(2, RReg, (uint_t*)USh, RReg);
184 
185     for (int i = 0; i < (int)ceilf(max(log2f(h-k-1), 0.0f)) + 2; ↩
            i++) {
186         int n = min(h - k + 1 - l, l);
187         int s = 0;
188         shift<uint_t, M, Q>(-s, VReg, VSh, TReg);
189         __syncthreads();
190         step<Base, M, Q>(USh, VSh, k + l + n - s + 2, TReg, RReg,↩
                n, l, 2);
191         __syncthreads();
192         if (i < 2) {
193             shift<uint_t, M, Q>(-n, RReg, USh, RReg);
194         }
195         else {
196             shift<uint_t, M, Q>(-1, RReg, USh, RReg);
197             l = l + n - 1;
198         }
199     }
200     shift<uint_t, M, Q>((h - k < 2) ? h - k - 4 : -2, RReg, VSh, ↩
            RReg);
201 }
202 
203 /**
204  * Refine the approximation of the quotient
205  */
206 template<typename Base, uint32_t M, uint32_t Q>
207 __device__ inline void
208 refine1( volatile typename Base::uint_t* USh
209       , volatile typename Base::uint_t* VSh
210       , typename Base::uint_t VReg[Q]
211       , typename Base::uint_t TReg[Q]
212       , int h
213       , int k
214       , int l
```

```
215       , typename Base::uint_t RReg[Q]
216 ) {
217     using uint_t = typename Base::uint_t;
218     h = h+1;
219     shift<uint_t, M, Q>(h-k-1, RReg, (uint_t*)USh, RReg);
220 
221     for (int i = 0; i < (int)ceilf(max(log2f(h-k-1), 0.0f)) + 2; ↩
          i++) {
222         __syncthreads();
223         shift<uint_t, M, Q>(0, VReg, VSh, TReg);
224         __syncthreads();
225         step<Base, M, Q>(USh, VSh, h, TReg, RReg, 0, 1, 0);
226         __syncthreads();
227         if (i > 1) {
228             l = min(2*l-1, h-k);
229         }
230     }
231     shift<uint_t, M, Q>(-1, RReg, VSh, RReg);
232 }
233 
234 /**
235  * Calculates the shifted inverse
236  */
237 template<typename Base, uint32_t M, uint32_t Q>
238 __device__ inline void
239 shinv( volatile typename Base::uint_t* USh
240      , volatile typename Base::uint_t* VSh
241      , typename Base::uint_t VReg[Q]
242      , typename Base::uint_t TReg[Q]
243      , int h
244      , int k
245      , typename Base::uint_t RReg[Q]
246 ) {
247     using uint_t = typename Base::uint_t;
248     using ubig_t = typename Base::ubig_t;
249     using uquad_t = typename Base::uquad_t;
250 
251     if (k == 0) {
252         quo<Base, Q>(h, VSh[0], VSh, RReg);
253         return;
254     }
255     if (k >= h && !eq<uint_t, Q>(VReg, h, &USh[2])) {
256         return;
257     }
258     if (k == h-1 && VSh[k] > Base::HIGHEST / 2 ) {
259         set<uint_t, Q>(RReg, 1, 0);
260         return;
261     }
262     if (eq<uint_t, Q>(VReg, k, &USh[3])) {
263         set<uint_t, Q>(RReg, 1, h - k);
264         return;
265     }
266 
267     if (threadIdx.x == 0) {
268         ubig_t tmp;
269         ubig_t V = (ubig_t)VSh[k - 1] | (ubig_t)VSh[k] << Base::↩
              bits;
```

```
270
271         if (Base::bits == 64) {
272             tmp = divide_u256_by_u128((__uint128_t)1 << 64, 0, V)↩
                ;
273         } else {
274             tmp = ((uquad_t)1 << 3*Base::bits) / V;
275         }
276         RReg[0] = (uint_t)(tmp);
277         RReg[1] = (uint_t)(tmp >> Base::bits);
278         if (tmp == 0) RReg[2] = 1;
279     }
280     __syncthreads();
281
282     refine3<Base, M, Q>(USh, VSh, VReg, TReg, h, k, 2, RReg);
283 }
284
285 /**
286  * Implementation of multi-precision integer division using
287  * the shifted inverse and classical multiplication
288  */
289 template<typename Base, uint32_t M, uint32_t Q>
290 __device__ inline void
291 divShinv( volatile typename Base::uint_t* USh
292         , volatile typename Base::uint_t* VSh
293         , typename Base::uint_t UReg[Q]
294         , typename Base::uint_t VReg[Q]
295         , typename Base::uint_t RReg1[2*Q]
296         , typename Base::uint_t RReg2[Q]
297 ) {
298     using uint_t = typename Base::uint_t;
299     using carry_t = typename Base::carry_t;
300
301     int h = prec<uint_t, Q>(UReg, (uint32_t*)USh);
302     int k = prec<uint_t, Q>(VReg, (uint32_t*)&USh[1]) - 1;
303
304     shinv<Base, M, Q>(USh, VSh, VReg, RReg2, h, k, RReg1);
305     __syncthreads();
306
307     bmulRegsQComplete<Base, 1, Q/2>(USh, VSh, UReg, RReg1, RReg1,↩
            M);
308     __syncthreads();
309
310     shiftDouble<uint_t, M, Q>(-h, RReg1, VSh, RReg1);
311     __syncthreads();
312
313     bmulRegsQ<Base, 1, Q/2>(USh, VSh, VReg, RReg1, RReg2, M);
314     __syncthreads();
315
316     if(lt<uint_t, Q>(UReg, RReg2, USh)) {
317         __syncthreads();
318         sub<Base, Q>(RReg1, 0, VSh);
319         bsubRegs<uint_t, uint_t, carry_t, Q>((carry_t*)VSh, RReg2↩
                , VReg, RReg2);
320     }
321     __syncthreads();
322     bsubRegs<uint_t, uint_t, carry_t, Q>((carry_t*)VSh, UReg, ↩
            RReg2, RReg2);
```

```
323     if (!lt<uint_t, Q>(RReg2, VReg, USh)) {
324         __syncthreads();
325         add1<Base, Q>(RReg1, USh);
326         bsubRegs<uint_t, uint_t, carry_t, Q>((carry_t*)VSh, RReg2↩
                , VReg, RReg2);
327     }
328 }
329 
330 /**
331  * Main division kernel
332  */
333 template<typename Base, uint32_t M, uint32_t Q>
334 __global__ void
335 __launch_bounds__(M/Q, BLOCKS_PER_SM*1024*Q/M)
336 divShinvKer( typename Base::uint_t* u
337         , typename Base::uint_t* v
338         , typename Base::uint_t* quo
339         , typename Base::uint_t* rem
340 ) {
341     using uint_t = typename Base::uint_t;
342 
343     extern __shared__ char sh_mem[];
344     volatile uint_t* VSh = (uint_t*)sh_mem;
345     volatile uint_t* USh = (uint_t*)(VSh + M);
346     uint_t VReg[Q];
347     uint_t UReg[Q];
348     uint_t RReg1[2*Q] = {0};
349     uint_t* RReg2 = &RReg1[Q];
350 
351     cpyGlb2Sh2Reg<uint_t, M, Q>(v, VSh, VReg);
352     cpyGlb2Sh2Reg<uint_t, M, Q>(u, USh, UReg);
353     __syncthreads();
354 
355     divShinv<Base, M, Q>(USh, VSh, UReg, VReg, RReg1, RReg2);
356     __syncthreads();
357 
358     cpyReg2Sh2Glb<uint_t, M, Q>(quo, VSh, RReg1);
359     cpyReg2Sh2Glb<uint_t, M, Q>(rem, USh, RReg2);
360 }
```

Listing 15: CUDA main division functions.

Futhark Code Listing

```
1  def div [m][ipb] (us: [ipb*(4*m)]u16) (vs: [ipb*(4*m)]u16) : ([↩
       ipb*(4*m)]u16, [ipb*(4*m)]u16) =
2      let h = prec us
3      let k = (prec vs) - 1
4  
5      let (kIsOne, us, vs, h, k) =
6          if k == 1 then
7              let h = h + 1
8              let k = k + 1
9              let us = shift 1 us
10             let vs = shift 1 vs
11             in (true, us, vs, h, k)
12         else
```

```
13              (false, us, vs, h, k)
14
15      let quo =
16          let m = m * 2
17          let quo_padded = ((shinv vs h k) ++ (replicate (ipb*(4*(m↩
                /2))) 0u16)) :> [ipb * (4 * m)]u16
18          let us_padded = (us ++ (replicate (ipb*(4*(m/2))) 0u16)) ↩
                :> [ipb * (4 * m)]u16
19          let mul_res = convMulV2 quo_padded us_padded
20          let mul_shifted = shift (-h) mul_res
21          let res = take (ipb*(4*(m/2))) mul_shifted
22          in res
23      let quo = quo :> [ipb * (4 * m)]u16
24
25      let (rem, _) = convMulV2 vs quo
26          |> bsub us
27
28      let (quo, rem) =
29          if not (lt rem vs) then
30              let quo = badd1u16 quo
31              let (rem, _) = bsub rem vs
32              in (quo, rem)
33          else
34              (quo, rem)
35
36      let rem =
37          if kIsOne then
38              shift (-1) rem
39          else
40              rem
41
42      in (quo, rem)
```

Listing 16: Implementation division function in Futhark

```
1  def shinv [m][ipb] (vs: [ipb*(4*m)]u16) (h: i64) (k: i64) : [ipb↩
       *(4*m)]u16 =
2      if k == 0 then
3          quo_single h (vs) (ipb*(4*m)) :> [ipb*(4*m)]u16
4      else if k >= h && !(eqBpow vs h) then
5          vs
6      else if k == h - 1 && vs[k] > u16.highest / 2 then
7          zeroAndSet 1 0 (ipb*(4*m)) :> [ipb*(4*m)]u16
8      else if eqBpow vs k then
9          zeroAndSet 1 (h - k) (ipb*(4*m)) :> [ipb*(4*m)]u16
10     else
11         let l = i64.min k 2
12         let V = (u64.u16 vs[k - 2]) | ((u64.u16 vs[k - 1]) << ↩
               1*16) | ((u64.u16 vs[k]) << 2*16)
13         let b2l = 1u64 << 4*16
14         let tmp = (b2l - V) / (V + 1)
15
16         let ws = tabulate (ipb*(4*m)) (\i ->
17             if i == 0 then u16.u64 tmp
18             else if i == 1 then u16.u64 (tmp >> 16)
19             else 0u16 )
20         in if h - k < l then
21             shift (h-k-l) ws
```

```
22          else
23              refine vs ws h k l
```

Listing 17: Implementation of shinv in Futhark

```
1  def refine [m][ipb] (vs: [ipb*(4*m)]u16) (ws: [ipb*(4*m)]u16) (h:↩
       i64) (k: i64) (l: i64) : [ipb*(4*m)]u16 =
2      let ws = shift 2 ws
3      let (ws, _, _) = loop (ws, l, i) = (ws, l, 0)
4          while h - k > (l + 1) do
5              let n = i64.min (h - k + 1 - l) l
6              let s = i64.max 0 (k - 2 * l + 1 - 2)
7              let vs = shift (-s) vs
8              let tmp = step vs ws (k + l + n - s + 2) l n
9              let ws = shift (-1) tmp
10             let l = l + n - 1
11             let i = i + 1
12             in (ws, l, i)
13     in shift (-2) ws
```

Listing 18: Implementation of refine in Futhark

```
1  def step [m][ipb] (vs: [ipb*(4*m)]u16) (ws: [ipb*(4*m)]u16) (h: ↩
       i64) (l: i64) (n: i64) : [ipb*(4*m)]u16 =
2      let (sign, tmp) = powDiff ws vs (h-n) (l-2)
3      let tmp = convMulV2 ws tmp
4          |> shift (2 * n - h)
5      let ws = shift n ws
6      in if sign != 0 then
7          --baddu16 ws tmp
8          baddV3 ws tmp
9      else
10         -- bsubu16 ws tmp
11         let (ret, _) = bsub tmp ws
12         in ret
```

Listing 19: Implementation of shift in Futhark

```
1  def powDiff [m][ipb] (vs: [ipb*(4*m)]u16) (ws: [ipb*(4*m)]u16) (h↩
       : i64) (l: i64) : (u32, [ipb*(4*m)]u16) =
2      let precV = prec vs
3      let precW = prec ws
4      let L = precW + precV - l + 1
5
6      in if (precV == 0 || precW == 0) then
7          let ret = zeroAndSet 1u16 h (ipb*(4*m))
8          let ret = ret :> [ipb*(4*m)]u16
9          in (1, ret)
10     else if (L >= h) then
11         let ret = convMulV2 vs ws
12         in if ltBpow ret h then
13             let bpow = zeroAndSet 1 h (ipb*(4*m))
14             let (ret, _) = bsub bpow ret
15             in (1, ret)
16         else
17             let bpow = zeroAndSet 1 h (ipb*(4*m))
18             let (ret, _) = bsub ret bpow
19             in (0, ret)
```

```
20      else
21          let ret = multmod vs ws L
22          in if !(ez ret) && ret[L-1] == 0 then
23              (0, ret)
24          else
25              let bpow = zeroAndSet 1 L (ipb*(4*m))
26              let (ret, _) = bsub bpow ret
27              in (1, ret)
```

Listing 20: Implementation of powDiff in Futhark

```
1 def multmod [m][ipb] (us: [ipb*(4*m)]u16) (vs: [ipb*(4*m)]u16) (d↩
      : i64) : [ipb*(4*m)]u16 =
2     let res = convMulV2 us vs
3     in tabulate (ipb*(4*m)) (\i -> if i >= d then 0u16 else res[i↩
          ])
```

Listing 21: Implementation of multMod in Futhark