On GPU Implementation for Multi-Precision Integer Division

Martin B. Marchioro[1], Aske N. Raahauge[1], Marc I. Løvenskjold[1], Cosmin E. Oancea[1], and Stephen M. Watt[2]

[1] DIKU, University of Copenhagen, Copenhagen 2100, Denmark
martin.marchioro@gmail.com, aske.n.r@di.ku.dk, Marc.ivan95@gmail.com, cosmin.oancea@di.ku.dk[0000−0001−5421−6876]

[2] Cheriton School of Computer Science, University of Waterloo, Canada
smwatt@uwaterloo.ca[0000−0001−8303−4983]

Abstract. This paper presents the issues arising in implementing a fast integer division algorithm on general purpose GPUs. The algorithm uses a Newton iteration based on the shifted inverse operation, keeping all arithmetic in the integer domain and relying on data-parallel operators. The principal contribution is an efficient GPU/CUDA implementation for integer precisions from 2^{15} to 2^{18} – sizes not supported by CGBN division. We propose algorithmic refinements, define a cost model in terms of multiplications, build on prefix sums and previous work on multi-precision multiplication, and present an evaluation showing near-optimal performance relative to the model for the target precision.

Keywords: Big integer arithmetic · CUDA · Data-parallel programming · GPGPU · High-level parallel languages · High-performance computing

1 Introduction

Multi-precision integer arithmetic is a basic component of computer algebra, cryptography, exact scientific computation, and symbolic-numeric software. Its performance matters not only for isolated large computations, but also for applications that require many independent integer operations at the same precision. This makes GPUs attractive: they offer high arithmetic throughput and massive parallelism, provided that the computation can be organized to keep data movement and inter-thread communication under control.

Existing GPU support for multi-precision integers is strongest at relatively small precisions, where one arithmetic instance can be mapped to a small cooperative group of threads. The NVIDIA CGBN library (Cooperative Groups Big Numbers) is an important example of this approach [29]. Such libraries provide very high performance in their intended range, but they do not cover all practically interesting sizes. In particular, there is a middle range of integer precisions—large enough that warp-level methods become strained, but still small enough that a complete arithmetic instance can fit within the fast memories of a single GPU thread block. This paper is concerned with this midsize regime.

In previous work we studied GPU implementations of multi-precision addition and multiplication in this regime [32]. That work showed that the classical algorithms, when scheduled carefully, can be made effective on GPUs by assigning one multi-precision operation to a CUDA block, keeping operands and intermediate values in registers or shared memory, and minimizing global-memory traffic. It also showed that such algorithms can be expressed in a high-level data-parallel language such as Futhark, although some low-level transformations needed for peak performance remain beyond the current compiler.

Division is a more demanding operation. It is not a simple local operation on digits, and the usual high-performance approach is to reduce division to multiplication by first computing an approximation to the reciprocal of the divisor. In conventional Newton iteration this typically requires working in a domain where such reciprocals exist, which can introduce multiple precision floating-point approximations and interactions between different arithmetic domains. For exact integer arithmetic this is undesirable: the implementation must preserve exactness while still exposing enough parallelism for the GPU.

The algorithm of Watt [38] addresses this problem by replacing the reciprocal with a whole shifted inverse. Instead of computing $1/v$, it computes an integer approximation to B^h/v, where B is the digit base and h is an appropriate precision. Newton iteration can then be formulated using integer multiplication and shift operations. This is attractive for GPUs because the main operations are data-parallel: multi-precision multiplication, addition, subtraction, comparison, and shifts. Moreover, the algorithm is parameterized by the multiplication method, so it can in principle benefit from either classical multiplication or faster multiplication algorithms.

This paper investigates how this shifted-inverse division algorithm can be implemented efficiently on GPUs for midsize integers. Our present implementation is written directly in CUDA rather than generated from Futhark. This is intentional: the goal is to identify the low-level scheduling, storage, and specialization issues that a high-level compiler would ultimately have to handle. In particular, division stresses the compiler more than addition or multiplication because the Newton refinement uses operations whose effective precisions change during the iteration. The main contributions of the paper are as follows:

- We present a CUDA implementation of multi-precision integer division based on the whole-shifted-inverse algorithm of Watt [38], targeting integer precisions from 2^{15} to 2^{18} bits.
- We describe implementation refinements needed to make the algorithm robust in an unsigned-integer setting, including explicit sign handling in the close-product computation and quotient correction when the computed shifted inverse may overestimate by one.
- We give a cost model for the implementation in terms of the number of full multi-precision multiplications required. For the classical multiplication used here, the model predicts that the full division operation should require at least five and at most seven full multiplication costs.

- We show how the supporting operations—shifts, comparisons, subtractions, close products, and variable-size multiplications—are mapped to CUDA blocks using registers and shared memory.
- We evaluate the implementation on an NVIDIA A100 GPU and compare it with CGBN's corresponding operation. The results show that, at the largest tested precisions, the measured division time is close to the five-multiplication lower bound predicted by the cost model, while also covering precision ranges not supported by CGBN division in our experiments.

The rest of the paper is organized as follows. Section 2 reviews the shifted-inverse division algorithm, including the refinements needed for the implementation and the resulting multiplication-based cost model. Section 3 describes the CUDA implementation, focusing on the use of registers, shared memory, scans, shifts, subtraction, and variable-size multiplication. Section 4 presents the experimental evaluation and compares the implementation with CGBN where the corresponding operations are supported. Section 5 discusses related work on GPU multi-precision arithmetic and high-level data-parallel programming. Section 6 summarizes the results and outlines future directions, including clipped products and compiler support for this class of exact arithmetic kernels.

2 Division Algorithm

2.1 Intuition

Parallel implementations for division of multi-precision integers typically rely on Newton's method to compute the reciprocal of the divisor, by applying it to function $f(x) = 1/x - v = 0$, where v is the divisor and $x = \frac{1}{v}$ its reciprocal:

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

However, Newton's method in its general form requires working in a related domain where the reciprocal exists. This can lead to a complex library structure in which the arithmetic domains are interdependent. As well, internal floating point representation can lead to potential loss of precision and overhead when converting between domains. The algorithm proposed by Watt [38] addresses these shortcomings and allows computation to be carried out in the integral domain, essentially by applying the Newton method to $f(x) = B^h/x - v = 0$, where B is the base of multi-precision integer v and h its precision, i.e., $B^h \geq v$. This computes the "whole shifted inverse" $x = \lfloor B^h/v \rfloor$ with the Newton iteration:

$$w_{(i+1)} = w_{(i)} + \left\lfloor w_{(i)} \cdot (B^h - v \cdot w_{(i)}) \cdot B^{-h} \right\rfloor, \quad w_{(i)} \in \mathbb{Z} \tag{2}$$

Equation (2) provides the intuition that the resulting Newton iteration can be written in terms of multiplications and efficient shift operations:

$$w_{(i+1)} = w_{(i)} + \text{shift}_{-h}(\text{shift}_h(w_{(i)}) - v \cdot w_{(i)}^2), \quad w_{(i)} \in \mathbb{Z} \tag{3}$$

i.e., (1) shift operations are used to scale the input to a degree that captures all the necessary information from the fractional counterpart, (2) then computations are performed at this higher scale, and (3) a corresponding inverse shift is applied at the end to restore the original magnitude, avoiding potential loss of information. The shift and shift inverse operations are formally defined as:

Definition 1 *(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

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

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

When $n \geq 0$, $\mathrm{shift}_{n,B}(u)$ corresponds to integer multiplication, i.e., $u \cdot B^n$. When $n < 0$ it is instead a specialized quotient operation, with u as dividend and B^n as divisor. Using a multi-precision integer representation, this is equivalent to an arithmetic shift, e.g. $\mathrm{shift}_1([1,2,3]) = [0,1,2]$ and $\mathrm{shift}_{-1}([1,2,3]) = [2,3,0]$.

In contrast, the whole shifted inverse $\mathrm{shinv}_n(v)$ corresponds to a specialized reciprocal, i.e., a reciprocal that has been shifted into our domain, e.g., $\mathrm{shinv}_3(8) = \mathrm{shift}_3(0.125) = 125$.

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

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

$$u \operatorname{quo} v \;\equiv\; \mathrm{shift}_{-h}\,(\; u \cdot \mathrm{shinv}_h(v)\;)\;+\;\delta, \quad \delta \in \{0,1\}.$$

Theorem 1 shows how these operations are used to derive the quotient: applying a reverse shift on the result of multiplying u with the whole shifted inverse of v produces a result that is at most one unit away from the correct quotient.

Example 1 Let $u = 314159265358979$, $v = 27183$ and find q such that $u = q \times v + r$ for $0 \leq r < v$. For $B = 10$, $h = 15$ since $B^{14} < u < B^{15}$. The iteration is given by equation (3). Start with initial guess $w_0 = 30000000000$. Then $30000000000 \to 35535300000 \to 36745061624 \to 36787648778 \to 36787698193 = \mathrm{shinv}_h(v)$ and $q = \lfloor B^{-h}(u \times \mathrm{shinv}_h v \rfloor = 11557196238$.

Example 2 Let $u = 726319138718412$, $v = 27183$. Again, let $B = 10$, $h = 15$ and initial guess $w_0 = 30000000000$. Then $30000000000 \to 35535300000 \to 36745061624 \to 36787648778 \to 36787698193 = \mathrm{shinv}_h(v)$ and $q_0 = \lfloor B^{-h}(u \times \mathrm{shinv}_h v) \rfloor = 26719609266$. Since $u - q_0 v = 40734 \geq v$, the correction $\delta = 1$ is needed, giving $q = q_0 + 1 = 26719609267$.

2.2 Original Algorithm and New Revisions

The pseudocode for computing the whole-shifted inverse is recounted in Algorithm 1, which uses multi-precision addition, subtraction and shift operations, and is defined in terms of a generic multi-precision multiplication method, denoted by MULT. As well, it uses the PowDiff function, defined in Algorithm 2.

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: $\text{SHINV}_{h,B}(v) = \left\lfloor \frac{B^h}{v} \right\rfloor$ ▷ All shifts are w.r.t. B
Uses: MULT, a multi-precision multiplication method
POWDIFF, to compute $B^h - v \cdot w$ (Algorithm 2)
SHIFT, a shift operation for multi-precision ints

Function `Shinv`$(v,\ h,\ B)$**:**
  ▷ Group digits if base is small
  if $B < 16$ **then**
    $p \leftarrow \min(6 - B, 2)$
    return $\text{SHIFT}_{h \bmod p - p}$ ($\text{SHINV}(v, h \operatorname{quo} p + 1, B^p)$)
  ▷ Special cases guarantee $B < v \le B^h/2$
  if $v < B$ **then return** $B^h \operatorname{quo} v$ ▷ 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
  $V \leftarrow v_{k-1} + v_k \cdot B$
  $w \leftarrow B^3 \operatorname{quo} V$ ▷ Divide 4 digits by 2 digits
  return $\text{REFINE}(v, h, k, w, 2)$ ▷ Refine w iteratively
Function `Refine`$(v,\ h,\ k,\ w,\ \ell)$**:**
  $g \leftarrow 2$ ▷ Guard digits
  $w \leftarrow \text{SHIFT}_g(w)$
  ▷ loops at least 2 iters; otherwise similar to while(l < h - k)
  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
    $w \leftarrow \text{STEP}(k + \ell + m - s + g,\ \text{SHIFT}_{-s}(v),\ w,\ m,\ \ell,\ g)$
    if $i < 2$ **then** $w \leftarrow \text{SHIFT}_{-m}(w)$
    else
      $w \leftarrow \text{SHIFT}_{-1}(w)$
      $\ell \leftarrow \ell + m - 1$
  $q \leftarrow (h - k < 2)\ ?\ h - k - 4 :\ -2$
  return $\text{SHIFT}_q(w)$
Function `Step`$(h,\ v,\ w,\ m,\ \ell,\ g)$**:**
  $(sign,\ x) \leftarrow \text{POWDIFF}(v, w, h - m, \ell - g, B)$
  if *sign* **then return** $\text{SHIFT}_m(w) + \text{SHIFT}_{2m-h}(\text{MULT}(w, x))$
  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

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: $(sign,\ |B^h - v \cdot w|)$
Uses: $\text{MULT}(a,\ b) = a \cdot b$
$\text{MULTMOD}(a,\ b,\ d,\ B) = (a \cdot b) \bmod B^d$
Function `PowDiff`$(v,\ w,\ h,\ \ell,\ B)$**:**

```
L ← prec_B v + prec_B w − ℓ + 1
if v = 0 ∨ w = 0 ∨ L ≥ h then
    if B^h > MULT(v, w) then return (1, B^h − MULT(v, w))
    else return (0, MULT(v, w) − B^h)
else
    P ← MULTMOD(v, w, L, B)
    if P = 0 then return (1, 0)
    else if P_{L−1} = 0 then return (0, P)
    else return (1, B^L − P)
```

The algorithm's correctness, fast convergence and work asymptotics are proven in [38]. In short, the algorithm has asymptotic work equal to one multi-precision multiplication; this holds even when Strassen's $O(n \cdot \log n)$ algorithm [37] for multiplication is used. The algorithm consists of three main stages: special case handling, initial approximation and iterative refinement, which we summarize below:

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

Initial Approximation (lines 11-13). The original algorithm uses a three-digit approximation of the original v value, namely $V = \sum_{i=0}^{\ell} v_{k-\ell+i} \cdot B^i$, where $\ell = min(k, 2)$. The shifted inverse is approximated to $w = \left(B^{2\ell} - V\right) \text{quo}\, V + 1$, which is more convenient and faster to compute than the equivalent $B^{2l} \text{quo}\, V$, albeit both take constant time. If sufficiently many digits are considered correct, i.e., $h - k \leq \ell$, then the original algorithm shifts w to the appropriate magnitude and performs an early return, i.e., $\text{SHIFT}_{h-k-\ell}(w)$, otherwise w is refined.

We perform the revision shown in lines 11-13 of Algorithm 1 because there are rare corner cases in which the early return refers to an overestimated (unsafe) value (see [34] for an example). Instead, we always pass the initial approximation through `Refine`, which guarantees correctness. This also allows a less-precise initial approximation of v, namely $V = v_{k-1} + v_k \cdot B$ that uses only two (instead of three) digits, i.e., $\ell = 2$, which is proven in [34] to preserve fast convergence and is convenient since it promotes machine arithmetic.

Iterative Refinement (functions `Refine`, `Step`, `PowDiff`). The initial approximation is refined at least once with the fastest-convergence routine—named `Refine3` in [38] and `Refine` here—that employs both shorter-iterates and divisor-prefixes techniques to achieve optimal work. `Refine` iteratively calls the `Step` function, which performs a single Newton iteration. `Step` invokes `PowDiff` (shown

Algorithm 3: Div(u, v, m, B)

Input: $u,\ v \in \mathbb{Z}_+^m,\ m,\ B \in \mathbb{Z}_+$
Output: (q, r) with $u = q \cdot v + r,\ r < v$
Uses: Mult(a, b) $= a \cdot b$
Prec, to compute precision
Shift, for shifting the integer

```
1  h ← prec(u)                                  ▷ precision of u
2  q ← shift_{-h}( Mult( u, shinv(v, h, B) ) )  ▷ Initial Quotient
3  m ← Mult(v, q)
4  if u < m then
5  |  q ← q − 1                                 ▷ Handles δ = −1
6  |  m ← m − v
7  r ← u − m                                    ▷ Initial Remainder
8  if r ≥ v then
9  |  q ← q + 1                                 ▷ Handles δ = 1
10 |  r ← r − v
11 return (q, r)
```

in Algorithm 2), which computes $B^h - v \cdot w$ using the close-product strategy for improved efficiency. Since PowDiff can return negative integers and since our implementation assumes unsigned integers, we refine the original implementation to explicitly keep track of the integers' sign in `PowDiff` and `Step`.

With divisor prefixes [38], overestimation can also arise during refinement: when `Refine` uses only a prefix of v, rare edge cases allow low digits to affect higher ones. It is proven [34] that Algorithm 1 can then overestimate the shifted inverse by at most one, so $\texttt{shift}_{-h}(u \cdot \texttt{shinv}_h(v))$ can be one below or one above $u \operatorname{quo} v$. We therefore revise Theorem 1 and amend Algorithm 3:

Theorem 2 *(Revised Quotient by shinv in $\mathbb{Z}$)*

Given two positive integers u and v, with $u \leq B^h$, the following hold:

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

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

2.3 Algorithm Cost in Number of Full Multiplications

This section assumes multi-precision integers consisting of M digits in base B and approximates the cost of the division algorithm in terms of the minimal and maximal number of full multiplications that are performed with the classical/quadratic algorithm. We consider that a full multiplication is performed whenever the result requires to compute more than $\frac{M}{2}$ digits. In summary, the computation of the shifted inverse requires at least two and at most four full multiplications. Once the shifted inverse is known, the straightforward computation of the quotient and remainder shown in Algorithm 3 requires:

– a full multiplication $v \cdot q$ at line 3, just before the quotient adjustment, and
– a multiplication $u \cdot \text{shinv}(v, h, B)$ at line 2 in the computation that approximates the quotient. This multiplication has to be computed in double precision $2 \cdot M$ because the result is shifted back by h digits. Assuming classical/quadratic multiplication, its cost is thus equal to *two* full multiplications.

It follows that the presented division algorithm requires *at least five and at most seven full multiplications*. The remainder of this section justifies the lower and upper bound of the cost of the whole shift inverse algorithm. Essentially, the loop inside the `Refine` function of Algorithm 1 exhibits at least one and at most two full multiplications inside PowDiff (called from `Step` and shown in Algorithm 2) and similarly for the computation of `Step` excluding PowDiff.

A full multiplications inside PowDiff requires that $k > \frac{h}{2}$, where k is the precision of (the original) v. The precision of the v parameter of PowDiff is $\text{prec}_B(v) = \min(2 \cdot \ell_i, k)$, which is also a good approximation of L. It follows that a full multiplication is performed whenever $\lfloor \frac{h}{2} \rfloor < \text{prec}_B(v) \approx 2 \cdot \ell_i$. However, we also know that the loop in `Refine` executes as long as $\ell_i < h - k$. Since $k > \frac{h}{2}$ it follows that $h - k \leq \lceil \frac{h}{2} \rceil$ and the loop terminates whenever ℓ_i reaches $\lceil \frac{h}{2} \rceil$.

The condition for performing a full multiplication was $\lfloor \frac{h}{2} \rfloor < 2 \cdot \ell_i$, which is equivalent to $\lfloor \frac{h}{2} \rfloor - 1 < 2 \cdot \ell_i - 1 \approx \ell_{i+1}$, since in most relevant cases the update formula for ℓ is $\ell_{i+1} = 2 \cdot \ell_i - 1$. It follows that it is possible, albeit unlikely, to be in the case $\lfloor \frac{h}{2} \rfloor - 1 < \ell_{i+1} < \lceil \frac{h}{2} \rceil$ that requires the loop in `Refine` to execute another iteration performing a full multiplication.

Overall, the focus of an efficient implementation of this algorithm is to achieve a runtime close to that of five full multiplications, which critically requires that the computation outside said multiplication does not introduce bottlenecks.

3 GPU Considerations

3.1 High-Level Rationale of the Implementation

We report a CUDA implementation of the division algorithm of section 2 for multi-precision integers whose computation fits inside one CUDA block. Following prior work on addition and multiplication [32], operands are copied once from global memory to registers, results are copied back, and the remaining computation uses fast memory. Intermediate arrays stay in registers when possible and are materialized transiently in shared memory only when communication or performance requires it, for example for overlapping access in classical multiplication or as a staging buffer for coalesced global-memory transfers. In addition we apply classical techniques such as efficient sequentialization of excess parallelism to minimize inter-thread communication and thus maximize throughput.

More detailed many-core models can account explicitly for memory transactions, synchronization, occupancy, and parallelism overheads [12,22]. Our cost model is deliberately coarser: it uses the measured cost of our multi-precision multiplication kernel as the architecture-aware unit of cost, since that kernel uses the same representation, memory hierarchy, and block-level execution strategy as the division kernel.

This section uses the following notation:

uint: the word size representing a digit of the multi-precision integer; we use word sizes of 16, 32 or 64 bits, since these are hardware supported.

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.

Memory Limitations. Currently, our **CUDA** implementation supports integer division on integers as large as 2^{18} bits. These sizes are limited by the maximum amount of shared memory available per CUDA block. For example, the CUDA implementation of classical multiplication [32] requires manifesting both input arrays in shared memory, which sums up to 64KB. Since the current practical maximal amount of shared memory per-CUDA block of our NVIDIA A100 GPU is about 100KB, this does not permit a multi-precision size of 2^{19} bits.

Another limiting factor is the amount of register memory: currently CUDA supports a maximum of 255 registers per thread or 64K registers per CUDA block, whichever is lowest. When the register demand exceeds these bounds, the `nvcc` compiler resorts to register spilling [28], which allocates the excess registers in a higher level cache that is, however, significantly less efficient to access.

Our implementation uses a maximal thread-sequentialization factor $Q = 4$, since the implementation of multiplication [32] is optimized for this value—i.e., each thread computes four elements of the multiplication result. Using $Q = 4$ for the biggest multi-precision size (2^{18} bits) results in spilling 40 registers (160 bytes) to slower storage, each of them being accessed just under three times (392 bytes of spilled storage are accessed). In principle, suitably increasing Q eliminates the spilling of registers—because it decreases the number of threads in a CUDA block and allows each thread to use more registers—but this did not improve the overall performance of the division implementation.

3.2 Overview of the CUDA Implementation

Figure 1 gives the gist of the CUDA implementation by presenting several parallel components of the division algorithm. Listing 1.1 demonstrates the manner in which the input multi-precision integers are efficiently loaded from global to register memory using shared memory as a staging buffer: The first step uses consecutive threads to copy consecutive memory locations to shared memory (`shmem`), thus achieving coalesced reads from global memory. Each thread performs Q such copies in the loop at lines 11-15, where the CUDA block size is dimensioned such that `M = Q * blockDim.x`. Since shared-memory accesses are not affected by un-coalesced accesses, the loop at lines 17-18 copies Q consecutive elements from shared memory to each of the thread's private (register) memory.

Listing 1.2 presents the operation that shifts a multi-precision integer U by n: the result of the shift is manifested in shared memory by the first loop, and once all threads have terminated work (i.e., reached the `__syncthreads()`

```
template< class uint,
          uint32_t M, uint32_t Q >
__device__ inline void
cpyGlb2Reg( uint* AGlb
          , volatile uint* shmem
          , uint AReg[Q]
) {
  const uint32_t
    glb_off = blockIdx.x * M;

  for (int i = 0; i < Q; i++) {
    int idx = i * blockDim.x +
              threadIdx.x;
    shmem[idx]= AGlb[idx+glb_off];
  }
  __syncthreads();
  for (int i = 0; i < Q; i++)
   AReg[i]=shmem[Q*threadIdx.x+i];
}
```

Listing 1.1. Coalesced copy of integers from global (`AGlb`) to shared (`sh_mem`) to register memory (`AReg`).

```
template< class uint,
          uint32_t M, uint32_t Q >
__device__ inline void
shift( int n, uint U[Q]
     , volatile uint* shmem
     , uint R[Q]               ) {
  for (int i = 0; i < Q; i++) {
    int idx = Q * threadIdx.x + i;
    int offset = idx + n;
    uint val = 0;
    if (offset >= 0 && offset < M)
        val = U[i];
    else offset = M-idx-1;
    shmem[offset] = val;
  }
  __syncthreads();
  for (int i = 0; i < Q; i++)
    R[i] = shmem[Q*threadIdx.x+i];
}
```

Listing 1.2. Shift integer `U` by `n`: input & result `R` are held in registers; shared memory (`sh_mem`) is used as staging buffer.

```
template<class uint, uint32_t Q>
__device__ inline void
subPowB( uint U[Q], uint32_t bpow
       , volatile uint* shmem  ) {
  uint32_t n = UINT32_MAX;
  if(threadIdx.x==0) shmem[0] = n;
  __syncthreads();
  // find lowest non-zero digit ↩
     whose index n >= bpow
  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)
      n = idx;
  }
  atomicMin((uint32_t*)shmem, n);
  __syncthreads();
  // subtract one from all digits ↩
     between bpow and n
  uint32_t nn = shmem[0];
  for (int i = 0; i < Q; i++) {
    uint32_t idx = Q*threadIdx.x+i;
    if (idx >= bpow && idx <= nn)
      U[i] = U[i] - 1;
} }
```

Listing 1.3. Subtraction `U` - B^{bpow}

```
class LTop { public:
  static __device__ inline
  int apply(int a, int b) {
   int r = a & b & 2;
   r +=(b & 1) || ((b & 2)&&(a & 1));
   return r;
  }
};
template<class uint, uint32_t Q>
__device__ inline bool
lt( uint U[Q], uint V[Q]
  , volatile uint* shmem ) {
  int R[Q] = {0};
  for (int i = 0; i < Q; i++) {
    if (U[i] < V[i])
      R[i] |= 1;
    else if (U[i] == V[i])
      R[i] |= 2;
    if (i > 0)
      R[i] = LTop::apply
                (R[i-1], R[i]);
  }
  scanBlk<LTop>(R[Q-1],shmem);
  return shmem[blockDim.x-1] & 1;
}
```

Listing 1.4. Less-than operator: `U<V`

Fig. 1. CUDA code for (1) copying in coalesced way from global to register memory, (2) shifting multi-precision integers, (3) subtracting a power of B and for (4) lower-than comparison of multi-precision integers.

barrier), the second loop loads Q consecutive elements of the result to the register (private) memory of each thread. Please note that the shared-memory buffer is only transiently utilized, and the same buffer is reused for following operations.

Several operations of the whole-shifted inverse algorithm use arguments that are powers of the corresponding base B, e.g., subtracting or comparing with B^{bpow}. Such specialized cases accept a more efficient implementation than when the arguments are arbitrary integers. Listing 1.3 demonstrates the case of subtraction $U - B^{bpow}$: The first loop finds the lowest index n of a non-zero digit of U that is greater than or equal to *bpow* across the Q elements of each thread, and the `atomicMin` operation uses hardware-supported atomics to efficiently extend this computation of n across all threads. The second loop finalizes the implementation by subtracting one from each digit whose index is between *bpow* and n. This specialization significantly reduces the inter-thread communication required by the general implementation of subtraction, presented in the next section 3.3.

Listing 1.4 implements the general case of the less-than operator $U < V$ on arbitrary multi-precision integers U and V. Essentially, the loop computes the less than and equal to relations among the Q digits of each thread. This is encoded in an `int` rather than a tuple of booleans: the first (least-significant) bit being set denotes less than (`U[i] < V[i]`), and the second bit being set denotes equality (`U[i]==V[i]`). The computation is extended across digits by means of the `LTop` operator (line 20) whose semantics on tuple-of-boolean arguments is:

```
def LTop (lt1, eq1) (lt2, eq2) = (lt2 || (eq2 && lt1), eq1 && eq2)
```

where `lt1/2` denote whether the first/second digits are in a less-than relation and `eq1/2` similarly treats equality. The operator implements comparison across two digits: The less-than relation holds either when the second digits are in a less-than relation (`lt2`) or when the second digits are equal and the first digits conform with the less-than relation. Equality holds if both digits are equal.

Finally, since `LTop` is associative, the computation is extended across all digits by performing a reduce with the `LTop` operator across the partial result of all threads. This is implemented as a prefix-sum (`scanBlk` at line 23) followed by selecting the last element of the shared-memory result at line 24. Next section presents the CUDA implementation of prefix sum and the manner in which it is applied to implement subtraction.

3.3 Subtraction of Multi-Precision Integers

We start by recounting the semantics of the map and exclusive scan (prefix sum) parallel skeletons. Scans are a classical primitive for parallel prefix computations [3]: map applies its function argument to each corresponding element of its input arrays and exclusive scan computes all prefixes under an associative operator $\odot$ having neutral element $e_{\odot}$:

$$\text{map } f\ [a_0, \ldots, a_{m-1}] \ = \ [f\ a_0, \ldots, f\ a_{m-1}]$$

$$\text{map2 } f\ [a_0, \ldots, a_{m-1}]\ [b_0, \ldots, b_{m-1}] \ = \ [f\ a_0\ b_0, \ldots, f\ a_{m-1}\ b_{m-1}]$$

```
class CarryOP { public: ...
  static int identity() { return 2;}
  static int apply(int c1, int c2) {
   return ( c1 & c2 & 2 ) |
     ((( c1 & ( c2 >> 1)) | c2) & 1);
} };
template<class D,class S,uint32_t Q>
__device__  void
subRegs ( D as[Q], S bs[Q], D rs[Q]
        , volatile int* shmem ) {
  int cs[Q];
  int carry = CarryOP::identity();
  for(int i=0; i<Q; i++) {
    rs[i] = as[i] - bs[i];
    int c = rs[i] > a[i];
    c = c | ((rs[i] == 0) << 1);
    carry= CarryOP::apply(carry, c);
    cs[i] = c;
  }
  shmem[threadIdx.x] = carry;
  __syncthreads();
  scanBlk<CarryOP>(shmem,threadIdx.x);
  carry = CarryOP::identity();
  if(threadIdx.x > 0)
    carry = shmem[threadIdx.x - 1];
  for(int i=0; i<Q; i++) {
    rs[i] -= (carry & 1);
    carry=CarryOP::apply(carry,cs[i]);
} }
```

Listing 1.5. Subtracting multi-precision integers: `as - bs` where `as > bs`.

```
template<class OP> __device__
int scanWarp( int u, int lane) {
  for(int i=1; i < 32; i *= 2) {
    int go = (lane >=i) ? i : 0;
    int elm = __shfl_up_sync
            (0xFFFFFFFF, u, go);
    u = OP::apply(elm, u);
  }
  return u;
}
template<class OP> __device__
int scanBlk(uint32_t u
        , volatile int* shmem){
  int idx = threadIdx.x;
  const int lane = idx & 31,
            warpid = idx >> 5;
  int r = scanWarp<OP>(u, lane);
  if(lane==31) shmem[warpid] =r;
  __syncthreads();
  if (warpid == 0)
   scanWarp<OP>(shmem[idx],lane);
  __syncthreads();
  if (warpid > 0)
  r=OP::apply(shmem[warpid-1],r);
  __syncthreads();
  shmem[idx] = r;
  __syncthreads();
  return r;
}
```

Listing 1.6. Inclusive scan at warp and block levels.

$$\text{scan}^{exc}\ \odot\ e_{\odot}\ [a_0, \ldots, a_{m-1}]\ \equiv\ [e_{\odot},\ a_0,\ a_0 \odot a_1,\ \ldots,\ a_0 \odot \ldots \odot a_{m-2}]$$

Subtraction follows a similar procedure as the one used for addition in [32]. Assuming two multi-precision unsigned integers $x > y$, each having m digits: $x = x_0 \cdot B^0 + \ldots + x_{m-1} \cdot B^{m-1}$ and $y = y_0 \cdot B^0 + \ldots + y_{m-1} \cdot B^{m-1}$, their subtraction $x - y$ is computed by means of a map-scan-map composition:

```
def ⊖₁ a b = (a - b > a, a - b == 0)
def ⊙ (uf1, z1) (uf2, z2) = ( (uf1 && z2) || uf2, z1 && z2 )
def ⊖₃ a b (c, _) = a - b - c
def sub x y = map2 ⊖₁ x y ▷ scan^exc ⊙ (false,true) ▷ map3 ⊖₃ x y
```

where the ▷ operator pipes the result of the preceding computation as the last argument of the following computation. Essentially, `map2` $\ominus_1$ x y computes whether the (independent) per-digit subtraction results in underflow (`a-b > a`) or in `0`. The result is passed to $\texttt{scan}^{exc}\ \odot$ (`false`, `true`), which computes and propagates the carry for each digit, and finally, the last map performs the per-digit subtraction to which it applies the carry. The mapped operators $\ominus_1$ and $\ominus_3$ differ from the ones used for addition, but the combine operator $\odot$ is the same.

Listings 1.6 and 1.5 show the CUDA implementation: listing 1.6 shows the implementation of the inclusive-scan operator at warp and block level, respectively, for a generic associative operator that accepts arguments of type int. For efficiency, the warp level uses register shuffling to avoid accessing shared memory.

$$
\begin{array}{llll|llll|l}
c_0 & = \sum_{i+j=0} a_i \cdot b_j & \mapsto t_0 & & c_0 & = \sum_{i+j=0} a_i \cdot b_j & \mapsto t_0 & & 1 \text{ term} \\
c_1 & = \sum_{i+j=1} a_i \cdot b_j & \mapsto t_1 & & c_1 & = \sum_{i+j=1} a_i \cdot b_j & \mapsto t_0 & & 2 \text{ terms} \\
c_2 & = \sum_{i+j=2} a_i \cdot b_j & \mapsto t_2 & & c_2 & = \sum_{i+j=2} a_i \cdot b_j & \mapsto t_1 & & 3 \text{ terms} \\
c_3 & = \sum_{i+j=3} a_i \cdot b_j & \mapsto t_3 & & c_3 & = \sum_{i+j=3} a_i \cdot b_j & \mapsto t_1 & & 4 \text{ terms} \\
\vdots & & \vdots & & \vdots & & \vdots & & \\
c_{m-4} & = \sum_{i+j=m-4} a_i \cdot b_j & \mapsto t_3 & & c_{m-4} & = \sum_{i+j=m-4} a_i \cdot b_j & \mapsto t_1 & & m-4 \text{ terms} \\
c_{m-3} & = \sum_{i+j=m-3} a_i \cdot b_j & \mapsto t_2 & & c_{m-3} & = \sum_{i+j=m-3} a_i \cdot b_j & \mapsto t_1 & & m-3 \text{ terms} \\
c_{m-2} & = \sum_{i+j=m-2} a_i \cdot b_j & \mapsto t_1 & & c_{m-2} & = \sum_{i+j=m-2} a_i \cdot b_j & \mapsto t_0 & & m-2 \text{ terms} \\
c_{m-1} & = \sum_{i+j=m-1} a_i \cdot b_j & \mapsto t_0 & & c_{m-1} & = \sum_{i+j=m-1} a_i \cdot b_j & \mapsto t_0 & & m-1 \text{ terms}
\end{array}
$$

Fig. 2. Embarrassingly-parallel load-balanced scheduling of quadratic/classical multiplication proposed by [32]: The left schedule uses $\frac{m}{2}$ threads each computing $q = 2$ elements of the result by performing m digit multiplications. The one on the right uses $m/4$ threads, each computing $q = 4$ elements by performing $2 \cdot m$ digit multiplications.

Listing 1.5 shows the implementation for subtracting `as - bs`, where inputs `as` and `bs` and the result `rs` are maintained in register memory and `as` is assumed larger than `bs`. The combine operator $\odot$, represented by class `CarryOP`, is optimized as in [32] to encode the two boolean values as the least-significant two bits of a 32-bit integer. Each thread performs the scan sequentially across its Q elements (the loop at lines 13-19) and publishes the final carry in shared memory (line 20). A block level scan (`scanBlk`) computes and propagates the carries (line 22), and the subtraction result is updated accordingly at lines 23-28. For very small numbers of digits, hardware-inspired carry-lookahead or carry-select schemes may have lower constant overhead, but our target range is larger multi-precision integers fitting within a CUDA block; accordingly, the implementation uses sequential propagation within each thread and a block-level scan only across the per-thread summaries.

3.4 Implementation of Multi-Precision Multiplication

The shift-inverse Algorithm 1 uses operations of different (increasing) sizes inside the loop in the `Refine` function. The operations that have cheap (linear) cost—e.g., comparison, subtraction—are implemented for simplicity by padding to the multi-precision size of the input. However, the multiplications have quadratic cost and need to be adjusted to the actual size of the arguments in order to allow the whole computation to have the same asymptotic as one full multiplication.

Our implementation of size-variant multiplication builds on the one proposed in [32] that addresses the fixed multi-precision case. Figure 2 visually depicts the scheduling proposed in [32] that minimizes inter-thread communication and

ensures a load-balanced execution. Denoting by m the multi-precision size and by q the sequentialization factor and assuming m and q powers of two, the schedule assigns to each thread the computation of q digits from the first half of the result and another q symmetrical-opposite digits from the second half. It follows that each thread performs a fixed number $\frac{Q}{2} \cdot m$ of base (digit) multiplications. Figure 2 shows the cases $q = 2$ and $q = 4$ on the left- and right-hand sides, respectively.

Our approach dispatches at runtime variant-sized multiplications to a number of statically-specialized instantiations of the multi-precision size:

```
if (m <= BLOCK) {
    smallMul<uint>(m, Ash, Bsh, Csh);
} else if (m <= 2 * BLOCK) {
    effMul<uint, BLOCK, 2>(Ash, Bsh, Csh);
} else if (m <= 4 * BLOCK) {
    effMul<uint, BLOCK, 4>(Ash, Bsh, Csh);
} else if (m <= 8 * BLOCK) ...
```

The pseudocode above assumes that input `Ash` and `Bsh` and the result `Csh` are allocated in shared memory (where the result `Csh` overlaps with an input) and are relocated to registers inside the specializations. A small multiplication `smallMul` refers to the case when its precision is smaller than or equal to the CUDA block size `BLOCK`. This is essentially computed by allocating each digit of the result to a different thread or possibly by a procedure that maximizes thread utilization by using atomic accumulations in shared memory. The other cases call `effMul` that uses the efficient scheduling depicted in Figure 2: the second and third template arguments denote the CUDA block size and the sequentialization factor q; it follows that the result is computed in precision $q \cdot \texttt{BLOCK} \geq m$. In practice we use $q = 4$ as the maximal sequentialization factor.

We recognize that a polished library would not use just one multiplication method throughout — that at different sizes one would transition between classical multiplication, Karatsuba or Toom-Cook multiplication and a number theoretic transform (NTT/FFT) method. For this work, however, we are content to study division in terms of whatever multiplication is used.

4 Empirical Evaluation

Hardware. The evaluation was run under the Red Hat Enterprise Linux 8.10 operating system on 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 TFLOP/s.

Methodology. The benchmarking setup is available at:

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

and extends the framework presented in [32] with the implementation of division $\frac{u}{v}$. Each problem instance is initialized with randomly generated integers. The precision of u is fixed to $M - 2$, such as to account for the two guard digits in the `Refine` function of Algorithm 1. The precision of the divisor v is randomly

Num Bits	Num Insts	Our Mul time (ms)	Our Mul / Cgbn Mul	Our Div / Our Mul	Our Div / Cgbn Div	Gmp Div / Our Div
2^{18}	2^{14}	271.205	0.037×	5.17×	–	11.3×
2^{17}	2^{15}	130.994	0.268×	5.38×	–	14.0×
2^{16}	2^{16}	69.059	1.041×	5.73×	–	18.9×
2^{15}	2^{17}	38.048	1.124×	7.83×	3.63×	12.6×
2^{14}	2^{18}	24.619	1.512×	8.95×	4.98×	7.4×
2^{13}	2^{19}	19.198	2.112×	9.96×	7.36×	11.1×

Table 1. Comparing the performance of multiplication and division with CGBN. The first two columns indicate the characteristics of the dataset: the integer precision in number of bits and the batch size, i.e., number of instances that run in parallel. The third column shows the runtime of our multiplication. The fourth column shows the speedup of the CGBN multiplication *vs.* ours. The fifth column shows the factor by which our division is slower than our multiplication. The sixth column shows the speedup of CGBN division *vs.* our implementation of division; "–" indicates that CGBN does not support that dataset. The last column shows the speedup of our CUDA division *vs.* the GMP division, run sequentially on an AMD EPYC 7352 CPU.

selected between 2 and $\frac{M}{2}$. This configuration ensures that the refinement loop (in `Refine`) always performs the maximum number of iterations.

The evaluation instantiates the digit type (*uint*) to 64-bit unsigned integer and the (maximal) sequentialization factor to $Q = 4$. The results are averaged across 25 runs to minimize statistical variance. We compare the performance of our implementation with that of the CGBN library [29] (namely `cgbn_div_rem`), which is also averaged over 25 runs. CGBN does not allow customization of the digits' word size or of the sequentialization factor.

Datasets. The datasets are chosen to vary the integer precision in number of bits (`Num Bits`) from 2^{13} to 2^{18} and the number of (parallel) division instances (`Num Insts`) from 2^{19} down to 2^{14}, such that $\texttt{Num Bits} \cdot \texttt{Num Insts} = 2^{32}$.

Validation. The results of our CUDA implementation match those of the GNU Multi-Precision Library (GMP) on all considered datasets. Beyond the performance datasets, correctness testing used 1000 randomized instances with random values and effective precisions, plus manual tests of known edge cases, including the shifted-inverse over-approximation that motivated Theorem 2. The tests also covered all multiplication sizes. All results agreed with GMP, providing strong evidence for correctness.

Performance Comparison with CGBN. Table 1 compares our multiplication and division with CGBN. For multiplication, our implementation is faster on the two highest-precision datasets by factors of 27× and 3.7×, roughly breaks even on the next two datasets, and is slower on the last two by factors of 1.5× and 2.1×. This is consistent with the different design points: CGBN uses a warp-level organization, whereas our implementation targets larger precisions with one arithmetic instance per CUDA block. A precise attribution of the gap to register pressure, scheduling, memory transactions, or occupancy would require a separate counter-level study, which is outside the scope of this paper.

The fifth column shows the factor by which our implementation of division is slower than one full multiplication that is used in its implementation. The highest-precision dataset (2^{18}) exhibits close to optimal performance for division, being only 5.2× slower than a full multiplication—recall that Section 2.3 argued that the target division algorithm requires at least 5 full multiplications.

The next two datasets slightly enlarge the gap to 5.4× and 5.7× slower than one full multiplication, which is still close to optimal. The last three datasets exhibit significant loss of performance as high as 2× away from the optimal, i.e., from 7.8× to 10× slower than one full multiplication. The performance loss is due to:

- assigning a suboptimal number of threads per CUDA block—the current implementation computes one division instance with a CUDA block of threads, which, for example, has suboptimal size 32 for the last dataset.
- the fact that the weight of the five full multiplications in the total runtime decreases with the integer precision, i.e., the other operations become significant.

The sixth column compares the performance of our division with that of CGBN. Importantly, CGBN does not support division at precisions higher than 2^{15} bits, although multiplication is supported suboptimally up to 2^{18} bits. On the last three lower-precision datasets, CGBN division is faster by significant factors ranging from 3.6× to 7.4×. In particular CGBN's division is only around 2.5× slower than one CGBN full multiplication. This indicates that, for the lower precision range up to 2^{15} bits, the CGBN implementation is better matched to the problem size than the block-level division method studied in this paper,[3] while the block-level method covers precisions higher than 2^{15} and up to at least 2^{18} bits. The last column shows the speedup of our division *vs.* GMP.

The present comparison is therefore intended to test the multiplication-based cost model and to identify the precision range where the shifted-inverse method is promising, rather than to provide a complete production-library benchmark. Further engineering work could combine a counter-level analysis with refinements such as clipped products [27], which could reduce the cost of the double-sized multiplication used in line 2 of Algorithm 3.

5 Related Work

This paper reports a CUDA implementation of the multi-precision division algorithm proposed in [38], which extends the Newton method to be carried out entirely in the integral domain by means of data-parallel operations such as shifting and multiplying multi-precision integers. This algorithm can in principle be further improved with the refinement of clipped products [27], which would enable cheaper computation of the middle digits of a multiplication result.

The implementation presented in this paper uses the CUDA implementations for addition and multiplication reported in [32], which also inspires the

[3] This is also why we did not optimize our implementation of division for the last three datasets—even reaching the optimal cost of five full multiplications would still not close the performance gap with CGBN.

parallelization strategy used for division. The work also reports matching implementations in the high-level data-parallel Futhark language. The Futhark compiler provides a set of useful optimizations, e.g., related to autotuning the degree of exploited parallelism [23], mapping computations to CUDA block level with allocation of intermediate results in shared memory [15], reusing memory buffers [24], automatic differentiation [4,35], static [16] or dynamic [14] verification of memory safety. These have enabled efficient acceleration of applications from various domains [11,33,36], which were exported as python libraries [13]. However, Futhark's multiplication suffers significant loss of performance because Futhark lacks code transformations aimed at mapping parallel arrays in register rather than shared memory. We have implemented the target division algorithm in Futhark as well; its performance is poor because the mentioned shortcomings are exacerbated by the nature of the division algorithm that requires parallel operations whose sizes vary through a loop, e.g., shifts and multiplications.

The closest related work to the one in this paper is the "Cooperative Groups Big Numbers" (CGBN) library [29], authored by NvidiaLab, that offers high-performance implementation for multi-precision integers up to 32K bits, including multiplication and division. The key technique enabling high performance in CGBN is to map an instance of integer computation on at most one warp of threads in order to leverage specialized NVIDIA hardware that allows values to be communicated directly between registers (of the warp), i.e., avoiding the overhead of accessing the shared memory, which has significantly higher latency. However, CGBN does not support division on precisions higher than 32K bits. Other less related works targeting parallel multi-precision arithmetic include:

- Previous work has investigated the feasibility of retargeting the multi-precision algorithms used by GMP for GPU execution. The conclusion has been that the architectural differences do not allow easy and/or efficient porting, which motivated development of new, inherently parallel algorithms [8].
- Efficient parallel algorithms for division have been devised to cover specialized input [10]. For example, the case where the divisor is a digit has been efficiently solved with a modified version of Jebelean's exact division algorithm, which has parallel complexity $O(n/p + \log p)$, where n denotes the precision and p the number of processors.
- CAMPARY [19,20] supports sequential CPU and GPU execution of multi-precision floating-point arithmetic up to a few hundred bits, e.g., addition, multiplication, division, square root. The key idea is to represent real numbers as unevaluated sums of multiple machine-precision floating point values.
- Older libraries aimed at supporting the functionality of GMP in CUDA include GARPREC [21] and CUMP [25,26]. The latter offers precision up to about 62 decimal digits and its objective was to outperform GARPREC. Similar works were ancestors of CGBN, with which we compare directly.
- Isupov [18] uses interval techniques to augment residue number arithmetic for operations that rely on magnitude for numbers with up to 4096 bits.
- A rich body of work [1,5,7,9] was aimed at accelerating multi-precision integer multiplication on GPUs using adaptations of the classical quadratic algorithm and of Strassen's log-linear algorithm [37].

Libraries supporting multi-precision arithmetic expose highly-parameterized components (tower of generic types). For example, the division algorithm is parameterized over the underlying multiplication algorithm—e.g., classical, Karatsuba or Strassen—which in turn imposes different representations for the multi-precision integers. Interoperating such libraries with various DSLs and mainstream computer-algebra systems [2,17], which support different mechanisms for parametrization, remains a direction of interest [6,30,31].

We note that algorithms for multi-precision integer arithmetic are closely related to algorithms for univariate polynomials with coefficients in $\mathbb{Z} \mod m$. Of particular interest are those with m a word-sized prime, for which there are good GPU algorithms. An important complication with multi-precision integers is the need to handle carries and borrows.

6 Future Work and Conclusions

This paper has examined multi-precision integer division on GPUs in a midsize precision range: larger than the range where warp-level cooperative libraries are most effective, but still small enough that a complete division instance can be assigned to a single CUDA block. In this regime, the whole-shifted-inverse algorithm of Watt [38] is a good structural match for GPU execution. Its Newton refinement is expressed using integer multiplication, shifting, comparison, subtraction, and related data-parallel operations, avoiding the need to move into a floating-point reciprocal domain.

The implementation maps one division instance to a CUDA block, keeps active operands in registers where possible, uses shared memory for communication and staging, and dispatches variable-size multiplications to statically specialized implementations. These choices reduce global memory traffic and expose the compiler transformations needed for a high-level data-parallel language such as Futhark to generate similar code.

The empirical results support the cost model developed in Section 2.3. For the largest tested precision, division is only slightly more expensive than the predicted lower bound of five full multiplications. This indicates that, at these sizes, the overheads from shifts, comparisons, subtractions, and control structure have been kept small relative to the dominant multiplication costs. At smaller precisions these overheads become more visible, and CGBN remains faster where its division implementation is available. Thus the present implementation should be viewed not as a replacement for warp-level libraries in their strongest range, but as evidence that a block-level strategy can effectively cover larger midsize precisions.

Several directions remain open. The most immediate algorithmic improvement is to use clipped products [27]. In the current implementation, the quotient approximation requires a double-precision multiplication even though only a selected range of product digits is ultimately needed. A clipped-product implementation could reduce this cost and bring the observed runtime closer to the idealized multiplication count.

A second direction is better specialization across operand sizes. The current implementation uses a fixed block-level strategy and a maximal sequentialization factor chosen to work well for the largest tested inputs. This is not optimal for smaller inputs, where the number of active threads per division instance can be too low and the relative cost of non-multiplication operations increases. More aggressive autotuning of block size, sequentialization factor, and multiplication specialization would likely improve performance across the full tested range.

A third direction is integration with high-level GPU programming systems. The CUDA implementation identifies several transformations that are important for performance: keeping short arrays in registers rather than shared memory, staging global-memory accesses to preserve coalescing, specializing operations by effective precision, and avoiding unnecessary materialization of intermediate arrays. Division is a useful stress test for such compiler work because it combines scans, shifts, comparisons, variable-size multiplications, and iterative refinement in a single exact arithmetic computation.

Finally, larger precisions raise the question of asymptotically faster multiplication. The shifted-inverse division algorithm is parameterized by multiplication, and can therefore in principle benefit from Karatsuba, FFT, or NTT-based methods. On GPUs, however, these methods introduce their own memory-management constraints. Near the fast-memory limit, CRT or NTT-based products may require several modular products for the largest multiplication, while shorter products arising during the Newton refinement may be able to use fewer moduli or longer transform vectors. This suggests that fast multiplication should not be treated as a black-box replacement only; it should be integrated with the changing precision requirements of the division algorithm.

In summary, the results show that exact multi-precision integer division based on whole shifted inverses can be implemented efficiently on GPUs for midsize operands. The implementation reaches near-model performance at the largest tested precisions, covers sizes beyond those supported by CGBN division in our experiments, and clarifies the low-level operations and compiler transformations needed for future high-level implementations.

References

1. Bantikyan, H.: Big integer multiplication with cuda fft (cufft) library. Int. J. Innovative Research in Computer and Communication Engineering **2**, 6317–6325 (2014)
2. Bernardin, L., Chin, P., DeMarco, P., Geddes, K.O., Hare, D.E.G., Heal, K.M., Labahn, G., May, J.P., McCarron, J., Monagan, M.B., Ohashi, D., Vorkoetter, S.M.: Maple Programming Guide, Maplesoft, a division of Waterloo Maple Inc., 1996-2023.
3. Blelloch, G.E.: Scans as primitive parallel operations. In: International Conference on Parallel Processing, ICPP'87, University Park, PA, USA, August 1987. pp. 355–362. Pennsylvania State University Press (1987)
4. Bruun, L.M., Larsen, U.S., Hinnerskov, N.H., Oancea, C.E.: Reverse-mode ad of multi-reduce and scan in futhark. In: Procs. of the 35th Symposium on Implementation and Application of Functional Languages. IFL '23, ACM (2024)

5. Chen, L., Covanov, S., Mohajerani, D., Moreno Maza, M.: Big prime field FFT on the GPU. In: Proc. 2017 International Symposium on Symbolic and Algebraic Computation (ISSAC 2017). pp. 85–92. ACM Press (2017)
6. Chicha, Y., Lloyd, M., Oancea, C., Watt, S.M.: Parametric Polymorphism for Computer Algebra Software Components. In: Procs. 6th Int. Symposium on Symbolic and Numeric Algorithms for Scientific Comput. pp. 119–130. Mirton Publishing House (2004)
7. Dieguez, A.P., Amor, M., Doallo, R., Nukada, A., Matsuoka, S.: Efficient high-precision integer multiplication on the gpu. The Int. Journal of High Perf. Computing Applics. **36**(3), 356–369 (2022)
8. Emmart, N.: A Study of High Performance Multiple Precision Arithmetic on Graphics Processing Units. Ph.D. thesis, University of Massachusetts (2018), https://hdl.handle.net/20.500.14394/17353
9. Emmart, N., Weems, C.: High precision integer addition, subtraction and multiplication with a graphics processing unit. Parallel Processing Letters **20**, 293–306 (12 2010)
10. Emmart, N., Weems, C.: Parallel multiple precision division by a single precision divisor. In: 18th Int. Conf. on High Performance Computing. pp. 1–9 (2011)
11. Gieseke, F., Rosca, S., Henriksen, T., Verbesselt, J., Oancea, C.E.: Massively-parallel change detection for satellite time series data with missing values. In: 2020 IEEE 36th International Conference on Data Engineering (ICDE). pp. 385–396 (2020)
12. Haque, S.A., Maza, M.M., Xie, N.: A many-core machine model for designing algorithms with minimum parallelism overheads. In: Joubert, G.R., Leather, H., Parsons, M., Peters, F.J., Sawyer, M. (eds.) Parallel Computing: On the Road to Exascale, Proceedings of the International Conference on Parallel Computing, ParCo 2015, 1-4 September 2015, Edinburgh, Scotland, UK. pp. 35–44. Advances in Parallel Computing, IOS Press (2015)
13. Henriksen, T., Dybdal, M., Urms, H., Kiehn, A.S., Gavin, D., Abelskov, H., Elsman, M., Oancea, C.: Apl on gpus: a tail from the past, scribbled in futhark. In: Procs. of the 5th Int. Workshop on Functional High-Performance Computing. p. 38–43. FHPC 2016, ACM (2016)
14. Henriksen, T., Oancea, C.E.: Bounds checking: An instance of hybrid analysis. In: Procs. of ACM SIGPLAN Int. Workshop on Libraries, Languages, and Compilers for Array Programming. pp. 88:88–88:94. ARRAY'14, ACM, New York, NY, USA (2014)
15. Henriksen, T., Thorøe, F., Elsman, M., Oancea, C.: Incremental flattening for nested data parallelism. In: Procs. of the 24th Symp. on Principles and Practice of Parallel Programming. pp. 53–67. PPoPP '19, ACM (2019)
16. Hinnerskov, N.H., Schenck, R., Oancea, C.: Verifying array properties in pure data-parallel programs. In: Procs. of ACM SIGPLAN Conf. on Programming Language Design and Implementation. PLDI 2026, ACM, New York, NY, USA (2026)
17. Research, Inc., W.: Mathematica, Version 14.0, https://www.wolfram.com/mathematica, Champaign, IL, 2024
18. Isupov, K.: Using floating-point intervals for non-modular computations in residue number system. IEEE Access **8**, 58603–58619 (2020)
19. Joldes, M., Muller, J., Popescu, V., Tucker, W.: CAMPARY: Cuda multiple precision arithmetic library and applications. In: Greuel, G., Koch, T., Paule, P., Sommese, A. (eds.) Mathematical Software – ICMS 2016. LNCS 9725. pp. 232–240. Springer Cham (2016)

20. Joldes, M., Muller, J., Popescu, V., Tucker, W.: CAMPARY library (2017), https://homepages.laas.fr/mmjoldes/campary/
21. Lu, M., He, B., Luo, Q.: Supporting extended precision on grahics processors. In: Proc. Sixth International Workshop on Data Management on New Hardware (DaMoN '10). pp. 19–26. ACM (2010)
22. Ma, L., Agrawal, K., Chamberlain, R.D.: A memory access model for highly-threaded many-core architectures. Future Generation Computer Systems **30**, 202–215 (2014), https://www.sciencedirect.com/science/article/pii/S0167739X13001349, special Issue on Extreme Scale Parallel Architectures and Systems, Cryptography in Cloud Computing and Recent Advances in Parallel and Distributed Systems, ICPADS 2012 Selected Papers
23. Munksgaard, P., Breddam, S.L., Henriksen, T., Gieseke, F.C., Oancea, C.: Dataset sensitive autotuning of multi-versioned code based on monotonic properties: Autotuning in futhark. In: Trends in Functional Programming (TFP 2021). pp. 3–23. Springer (2021)
24. Munksgaard, P., Henriksen, T., Sadayappan, P., Oancea, C.: Memory optimizations in an array language. In: Procs. of the Int. Conf. on High Performance Computing, Networking, Storage and Analysis. SC '22, IEEE Press (2022)
25. Nakayama, T., Takahashi, D.: Implementation of multiple-precision floating-point arithmetic for GPU computing. In: Proc 23rd IASTED Int. Conf. on Parallel and Distributed Computing and Systems (PDCS 2011). pp. 343–349. IASTED (2011)
26. Nakayama, T.: CUMP library (2017), https://github.com/skystar0227/CUMP
27. Norman, A.C., Watt, S.M.: Computing clipped products. In: Computer Algebra in Scientific Computing (CASC 2024). pp. 273–291. Springer (2024)
28. Nvidia: 1. Introduction; CUDA C++ Programming Guide — docs.nvidia.com. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model (2016)
29. NVlabs: Cooperative Groups Big Numbers (CGBN) Library (2018), https://github.com/NVlabs/CGBN
30. Oancea, C.E., Watt, S.M.: Domains and expressions: an interface between two approaches to computer algebra. In: ISSAC '05. pp. 261–268. ACM (2005)
31. Oancea, C.E., Watt, S.M.: Parametric polymorphism for software component architectures. In: OOPSLA '05. pp. 147–166. ACM (2005)
32. Oancea, C.E., Watt, S.M.: GPU Implementations for Midsize Integer Addition and Multiplication, pp. 51–79. Springer (2026)
33. Oancea, C.E., Robroek, T., Gieseke, F.: Approximate nearest-neighbour fields via massively-parallel propagation-assisted k-d trees. In: IEEE BigData. pp. 5172–5181 (2020)
34. Raahauge, A.N., Marchioro, M.B., Løvenskjold, M.I.: Efficient GPU Implementation of Multi-Precision Integer Division. Master's thesis, University of Copenhagen (2025), https://futhark-lang.org/student-projects/MSc-Bigint-Div-Aske-Mark-Martin.pdf
35. Schenck, R., Rønning, O., Henriksen, T., Oancea, C.E.: Ad for an array language with nested parallelism. In: SC '22 (2022)
36. Serykh, D., Oehmcke, S., Oancea, C., Masiliūnas, D., Verbesselt, J., Cheng, Y., Horion, S., Gieseke, F., Hinnerskov, N.: Seasonal-trend time series decomposition on graphics processing units. In: IEEE BigData. pp. 5914–5923 (2023)
37. Strassen, V., Schönhage, A.: Schnelle multiplikation großer zahlen. Computing **7**(3-4), 281–292 (1971)
38. Watt, S.M.: Efficient generic quotients using exact arithmetic. In: ISSAC '23. pp. 535–544. ACM (2023)