Efficient Big Integer Arithmetic Using GPGPU

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

Thorbjørn Bülow Bringgaard

University of Copenhagen
Department of Computer Science

June 18, 2024

Outline

Introduction

Big Integers

Positional number system

An integer $u \in \mathbb{N}$ can be expressed in base $B \in \mathbb{N} > 1$ with m digits $u_{i \in \{0,1,\ldots,m-1\}} \in \{0, 1, \ldots, B-1\}$ by the sum:

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

E.g. the number 256 in base $B = 10$ is $[6, 5, 2]$.

Big integers in the positional number system maps to an array of unsigned machine words.

E.g. the number 4294967298 in base $B = 2^{32}$ is $[2, 1]$.

Introduction

Big Integers

Positional number system

An integer $u \in \mathbb{N}$ can be expressed in base $B \in \mathbb{N} > 1$ with m digits $u_{i \in \{0,1,\ldots,m-1\}} \in \{0, 1, \ldots, B-1\}$ by the sum:

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

E.g. the number 256 in base $B = 10$ is $[6, 5, 2]$.

Big integers in the positional number system maps to an array of unsigned machine words.

E.g. the number 4294967298 in base $B = 2^{32}$ is $[2, 1]$.

Introduction

Big Integers

Positional number system

An integer $u \in \mathbb{N}$ can be expressed in base $B \in \mathbb{N} > 1$ with m digits $u_{i \in \{0,1,\ldots,m-1\}} \in \{0, 1, \ldots, B-1\}$ by the sum:

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

E.g. the number 256 in base $B = 10$ is $[6, 5, 2]$.

Big integers in the positional number system maps to an array of unsigned machine words.

E.g. the number 4294967298 in base $B = 2^{32}$ is $[2, 1]$.

Introduction

Big Integers

Positional number system

An integer $u \in \mathbb{N}$ can be expressed in base $B \in \mathbb{N} > 1$ with m digits $u_{i \in \{0,1,\ldots,m-1\}} \in \{0, 1, \ldots, B-1\}$ by the sum:

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

E.g. the number 256 in base $B = 10$ is $[6, 5, 2]$.

Big integers in the positional number system maps to an array of unsigned machine words.

E.g. the number 4294967298 in base $B = 2^{32}$ is $[2, 1]$.

Introduction

Implementation

The idea is to process integers at CUDA block-level, requiring:

- Sequentialization factor.

- Coalesced transactions to global memory.

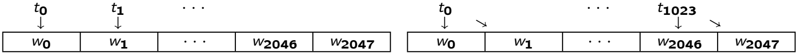
- Multiple instances per block.

Introduction

Implementation

The idea is to process integers at CUDA block-level, requiring:

- Sequentialization factor.



- Coalesced transactions to global memory.

- Multiple instances per block.

Introduction

Implementation

The idea is to process integers at CUDA block-level, requiring:

- Sequentialization factor.

t_0	t_1	$\cdots$		
w_0	w_1	$\cdots$	w_{2046}	w_{2047}

t_0		$\cdots$	t_{1023}	
w_0	w_1	$\cdots$	w_{2046}	w_{2047}

- Coalesced transactions to global memory.

t_0		$\cdots$	t_{1023}	
w_0	w_1	$\cdots$	w_{2046}	w_{2047}

t_0	t_1	$\cdots$	t_{1022}	t_{1023}
w_0	w_1	$\cdots$	w_{2046}	w_{2047}

- Multiple instances per block.

t_0	t_1	$\cdots$	t_{14}	t_{15}
w_0	w_1	$\cdots$	w_{14}	w_{15}

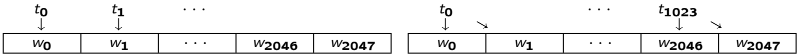
t_0	t_1	$\cdots$	t_{30}	t_{31}
w_0	w_1	$\cdots$	u_{14}	u_{15}

Introduction

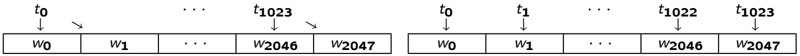
Implementation

The idea is to process integers at CUDA block-level, requiring:

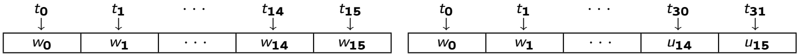
- Sequentialization factor.



- Coalesced transactions to global memory.



- Multiple instances per block.



Addition

Algorithm

badd – bitwise-optimized addition of big integers

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

```
(ws, cs) = map2 ⊕ us vs
pcs = scan_exc ⊗ 2 cs
w = map2 (λ w c → w + (c & 1) ) ws pcs
```

where

$$x \oplus y := (r,\ \mathtt{uint}(r < x) \mid (\mathtt{uint}(r == B-1) \ll 1)), \quad \textbf{where } r = x + y \tag{2}$$

$$x \otimes y := (((x \,\&\, (y \gg 1)) \mid y) \,\&\, 1) \mid (x \,\&\, y \,\&\, 2) \tag{3}$$

Addition

Implementation 1/2

Optimizations boils down to:

- Sequentialize the parallelism in excess.
- Segmented scan with flags integrated in the bitwise carry-overflow representation.

Addition

Implementation 1/2

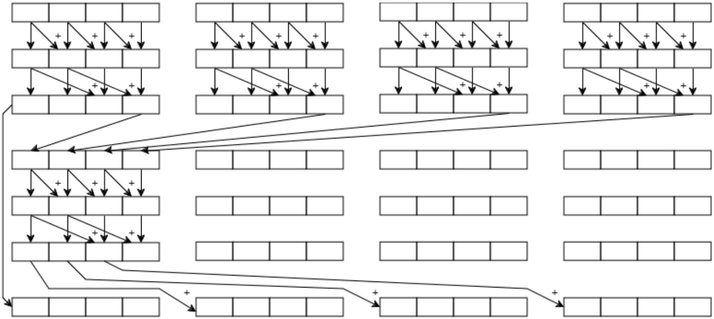
Optimizations boils down to:

- Sequentialize the parallelism in excess.
- Segmented scan with flags integrated in the bitwise carry-overflow representation.

Addition

Implementation 2/2

Implementation revolves around a scan:



Addition

CUDA 1/3

Main addition kernel:

```
template<class B, uint32_t m, uint32_t q, uint32_t ipb>
__global__ void
baddKer3(typename B::uint_t* as, typename B::uint_t* bs,
                                 typename B::uint_t* rs) {
    using uint_t  = typename B::uint_t;
    using carry_t = typename B::carry_t;
    uint_t ass[q];
    uint_t bss[q];
    cpGlb2Reg<uint_t,m,q,ipb>(as, ass);
    cpGlb2Reg<uint_t,m,q,ipb>(bs, bss);
    __syncthreads();

    uint_t rss[q];
    __shared__ carry_t shmem[m*ipb];
    baddKer3Run<B,m,q,ipb>(ass, bss, rss, shmem);   * badd
    cpReg2Glb<uint_t,m,q,ipb>(rss, rs);
}
```

Addition

CUDA 2/3

Step 1.

```
uint_t css[q];
carry_t acc = threadIdx.x % (m/q) == 0
    ? SegCarryProp<B>::setFlag(SegCarryProp<B>::identity())
    : SegCarryProp<B>::identity();
#pragma unroll
for(int i=0; i<q; i++) {
    rss[i] = ass[i] + bss[i];
    css[i] = ((carry_t) (rss[i] < ass[i]))
               | (((carry_t) (rss[i] == B::HIGHEST)) << 1);
    acc = SegCarryProp<B>::apply(acc, css[i]);
}
shmem[threadIdx.x] = acc;
__syncthreads();
```

Addition

CUDA 3/3

Step 2.

```
acc = scanExcBlock< SegCarryProp<B> >(shmem, threadIdx.x);
acc = threadIdx.x % (m/q) == 0 ? SegCarryProp<B>::identity()
                                : acc;
```

Step 3.

```
#pragma unroll
for(int i=0; i<q; i++) {
    rss[i] += (acc & 1);
    acc = SegCarryProp<B>::apply(acc, css[i]);
}
```

Addition

Futhark 1/3

Main addition function:

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

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

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

  in baddV3Run ush vsh :> [ipb*(4*m)]ui  ⋆ badd
```

Addition

Futhark 2/3

Step 1.

```
let (ws, cs, accs) = unzip3 <| imap (0..<ipb*m) (\ i ->
  let i4 = i*4
  let (u1,u2,u3,u4) = (us[i4], us[i4+1], us[i4+2], us[i4+3])
  let (v1,v2,v3,v4) = (vs[i4], vs[i4+1], vs[i4+2], vs[i4+3])
  let (w1,w2,w3,w4) = (u1 + v1, u2 + v2, u3 + v3, u4 + v4)
  let (c1,c2,c3,c4) = (carryAug w1 u1, carryAug w2 u2,
                       carryAug w3 u3, carryAug w4 u4)
  let c1 = (boolToCt (i % m == 0)) << 2 | c1
  let acc = carryProp c1 <| carryProp c2 <| carryProp c3 c4
  in ((w1, w2, w3, w4), (c1, c2, c3, c4), acc))
```

Step 2.

```
let pcs = scanExc carryPropSeg carryPropE accs
```

Addition

Futhark 3/3

Step 3.

```
let (wi1s, wi2s, wi3s, wi4s) = imap4 ws cs pcs (0..<ipb*m)
  (\ (w1, w2, w3, w4) (c1, c2, c3, _) acc1 i ->
      let acc1 = if i % m == 0 then carryPropE else acc1
      let acc2 = carryProp acc1 c1
      let acc3 = carryProp acc2 c2
      let acc4 = carryProp acc3 c3
      in ((w1 + fromCt (acc1 & 1), i*4),
          (w2 + fromCt (acc2 & 1), i*4+1),
          (w3 + fromCt (acc3 & 1), i*4+2),
          (w4 + fromCt (acc4 & 1), i*4+3))) |> unzip

let (ws, inds) = unzip <| wi1s ++ wi2s ++ wi3s ++ wi4s
in scatter (replicate (ipb*(4*m)) 0) inds ws
```

Addition

Evaluation

One addition and ten additions of base `uint64_t` in GB/s:

Bits	Insts	CGBN1	CUDA1	Fut1	CGBN10	CUDA10	Fut10
2^{18}	2^{14}	62	161	–	25	92	–
2^{17}	2^{15}	67	163	–	60	109	–
2^{16}	2^{16}	19	166	146	73	124	24
2^{15}	2^{17}	19	166	168	45	124	29
2^{14}	2^{18}	84	166	168	97	124	29
2^{13}	2^{19}	164	165	168	162	123	29
2^{12}	2^{20}	165	166	168	164	123	29
2^{11}	2^{21}	164	166	168	161	124	29
2^{10}	2^{22}	156	166	168	152	124	29
2^{9}	2^{23}	118	167	168	113	124	29

Classical Multiplication

Algorithm 1/2

Classical multiplication of big integers

Multiplying integer $u \in \mathbb{N}$ by $v \in \mathbb{N}$ in base B and m digits, is classically computed by:

$$u \cdot v = \sum_{k=0}^{m-1} \left(\sum_{\substack{0 \leq i,j < m \\ i+j=k}} u_i \cdot v_j \right) B^k \tag{4}$$

$$\begin{array}{|c|c|c|c|c|}
\hline
w_0 & w_1 & w_2 & \cdots & w_{m-1} \\
\hline
\end{array}$$

$$\begin{array}{ccccccccc}
= & & = & & = & & & & = \\
0 & \searrow & c_0 & \searrow & c_1 & \searrow & \cdots & \searrow & c_{m-2} \\
+ & & + & & + & & & & + \\
u_0 v_0 & & u_0 v_1 & & u_0 v_2 & & \cdots & & u_0 v_{m-1} \\
 & & + & & + & & & & + \\
 & & u_1 v_0 & & u_1 v_1 & & \cdots & & u_1 v_{m-2} \\
 & & & & + & & & & + \\
 & & & & u_2 v_0 & & \cdots & & u_2 v_{m-3} \\
 & & & & & & \ddots & & \vdots \\
 & & & & & & & & u_{m-1} v_0
\end{array}$$

Classical Multiplication

Algorithm 2/2

convmul – work-balanced classical multiplication by convolutions

Represent each convolution (column) by a *low*, *high*, and *carry* part.

Thread $k \in \{0, 1, \ldots, (m/2) - 1\}$ handles convolution $k_1 = k$ and $k_2 = m - 1 - k$.

The convolutions are added in the following pattern:

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

Classical Multiplication

Implementation

Optimizations boils down to:

- Double word-sizes to reduce work (for CUDA).
- Less communication but more sequentialization.

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

Classical Multiplication

Implementation

Optimizations boils down to:

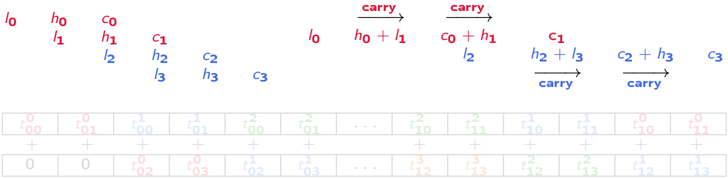
- Double word-sizes to reduce work (for CUDA).
- Less communication but more sequentialization.

Classical Multiplication

Implementation

Optimizations boils down to:

- Double word-sizes to reduce work (for CUDA).
- Less communication but more sequentialization.



Classical Multiplication

Implementation

Optimizations boils down to:

- Double word-sizes to reduce work (for CUDA).
- Less communication but more sequentialization.

l_0	h_0	c_0			
	l_1	h_1	c_1		
		l_2	h_2	c_2	
			l_3	h_3	c_3

	$\xrightarrow{\text{carry}}$	$\xrightarrow{\text{carry}}$			
l_0	$h_0 + l_1$	$c_0 + h_1$	c_1		
		l_2	$h_2 + l_3$	$c_2 + h_3$	c_3
			$\xrightarrow[\text{carry}]{}$	$\xrightarrow[\text{carry}]{}$	

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

Classical Multiplication

CUDA 1/3

The lower convolution:

```
ubig_t lh0[2]; lh0[0] = 0; lh0[1] = 0;
ubig_t lh1[2]; lh1[0] = 0; lh1[1] = 0;

int k1 = threadIdx.x*2;
int k1_start = (k1/m) * m;

for (int i=k1_start; i<=k1; i++) {
    int j = k1 - i + k1_start;
    uint_t a = shmem_as[i];
    iterate<B>(a, shmem_bs[j], lh0);
    iterate<B>(a, shmem_bs[j+1], lh1);
}
iterate<B>(shmem_as[k1+1], shmem_bs[k1_start], lh1);

combine2<B>(lh0, lh1, lhck1);
```

Classical Multiplication

CUDA 2/3

```
// iterate(uint_t a, uint_t b, ubig_t* lh)
  ubig_t ab = ((ubig_t) a) * ((ubig_t) b);
  lh[0] += ab & ((ubig_t) B::HIGHEST);
  lh[1] += ab >> B::bits;

// combine2(ubig_t* lh0, ubig_t* lh1, uint_t* lhc)
  uint_t h0t = (uint_t) lh0[1];
  uint_t h0  = h0t + ((uint_t) (lh0[0] >> B::bits));
  uint_t c0  = ((uint_t) (lh0[1] >> B::bits)) + (h0 < h0t);

  uint_t h1t = (uint_t) lh1[1];
  uint_t h1  = h1t + ((uint_t) (lh1[0] >> B::bits));
  uint_t c1  = ((uint_t) (lh1[1] >> B::bits)) + (h1 < h1t);

  lhc[0] = (uint_t) lh0[0];
  lhc[1] = h0 + ((uint_t) lh1[0]);
  lhc[2] = c0 + h1 + (lhc[1] < h0);
  lhc[3] = c1 + (lhc[2] < h1);
```

Classical Multiplication

CUDA 3/3

Memory layout:

t^0_{00} $\downarrow w$	t^1_{00} $\downarrow w$	t^1_{10} $\downarrow w$	t^0_{10} $\downarrow w$	t^0_{01} $\downarrow w$	t^1_{01} $\downarrow w$	t^1_{11} $\downarrow w$	t^0_{11} $\downarrow w$
l_0	l_2	l_3	l_4	lh_0	lh_1	lh_2	lh_3
$\uparrow r$ t^0_{00}	$\uparrow r$ t^0_{12}	$\uparrow r$ t^1_{00}	$\uparrow r$ t^1_{12}	$\uparrow r$ t^0_{01}	$\uparrow r$ t^0_{13}	$\uparrow r$ t^1_{01}	$\uparrow r$ t^1_{13}
ls				hls			

t^0_{02} $\downarrow w$	t^1_{02} $\downarrow w$	t^1_{12} $\downarrow w$	t^0_{12} $\downarrow w$	t^0_{03} $\downarrow w$	t^1_{03} $\downarrow w$	t^1_{13} $\downarrow w$	t^0_{13} $\downarrow w$
hc_0	hc_1	hc_2	hc_3	cc_0	cc_1	cc_2	cc_3
0_r t^0_{10}	$\nwarrow r$ t^0_{02}	$\nwarrow r$ t^1_{10}	$\nwarrow r$ t^1_{02}	0_r t^0_{11}	$\nwarrow r$ t^0_{03}	$\nwarrow r$ t^1_{11}	$\nwarrow r$ t^1_{03}
chs				ccs			

Classical Multiplication

Futhark 1/3

Presented solution was sub-optimal.

Fixed by tagging parts with their index in the convolution function:

```
let s1 = (k1+2) % (4*m) != 0 |> i64.bool |> (\i -> i - 1)
let s2 = (k2+1) % (4*m) != 0 |> i64.bool |> (\i -> i - 1)
in
((l1,  lh1,  l2,   lh2), (hc1,      cc1,      hc2,      cc2     ),
 (k1,  k1+1, k2-1, k2),  (s1|k1+2,  s1|k1+3,  s2|k2+1,  s2|k2+2))
```

However, the operator is still slow for larger integers.

Inspired by [1], the shared memory is piped to opaque-function.

Significant speedup, but still slower than the basic version.

Classical Multiplication

Futhark 1/3

Presented solution was sub-optimal.

Fixed by tagging parts with their index in the convolution function:

```
let s1 = (k1+2) % (4*m) != 0 |> i64.bool |> (\i -> i - 1)
let s2 = (k2+1) % (4*m) != 0 |> i64.bool |> (\i -> i - 1)
in
((l1,  lh1,  l2,   lh2), (hc1,     cc1,     hc2,     cc2    ),
 (k1,  k1+1, k2-1, k2),  (s1|k1+2, s1|k1+3, s2|k2+1, s2|k2+2))
```

However, the operator is still slow for larger integers.

Inspired by [1], the shared memory is piped to opaque-function.

Significant speedup, but still slower than the basic version.

Classical Multiplication

Futhark 1/3

Presented solution was sub-optimal.

Fixed by tagging parts with their index in the convolution function:

```
let s1 = (k1+2) % (4*m) != 0 |> i64.bool |> (\i -> i - 1)
let s2 = (k2+1) % (4*m) != 0 |> i64.bool |> (\i -> i - 1)
in
((l1,  lh1,   l2,   lh2), (hc1,      cc1,      hc2,      cc2     ),
 (k1,  k1+1,  k2-1, k2),  (s1|k1+2,  s1|k1+3,  s2|k2+1,  s2|k2+2))
```

However, the operator is still slow for larger integers.

Inspired by [1], the shared memory is piped to opaque-function.

Significant speedup, but still slower than the basic version.

Classical Multiplication

Futhark 2/3

```
let CONV (us: []ui) (vs: []ui) (tid: i64) = #[unsafe]
  let k1 = tid
  let k1_start = (k1 / (2*m)) * (2*m)
  let lhc1 : (ui, ui, ui) =
  loop (l, h, c) = (0, 0, 0)
  for i < k1 + 1 - k1_start
  do let j = k1 - i
     let lr = us[i+k1_start] * vs[j]
     let hr = mulHigh us[i+k1_start] vs[j]
     let ln = l + lr
     let hn = h + hr + (fromBool (ln < l))
     let cn = c + (fromBool (hn < h))
     in (ln, hn, cn)

  let k2 = ipb*2*m-1 - k1
  ...
  in (lhc1, lhc2)
```

Classical Multiplication

Futhark 3/3

```
let ush = map (\i -> us[i]) (0..<ipb*(2*m))
let vsh = map (\i -> vs[i]) (0..<ipb*(2*m))

let (lhcs1, lhcs2) = map (CONV ush vsh) (0..<ipb*m) |> unzip
let (ls1, hs1, cs1) = unzip3 lhcs1
let (ls2, hs2, cs2) = unzip3 <| reverse lhcs2
let ls = ls1 ++ ls2 :> [ipb*(2*m)]ui
let hs = hs1 ++ hs2 :> [ipb*(2*m)]ui
let hs = map (\ i -> if i % (2*m) == 0 then 0 else hs[i-1] )
             (0..<ipb*(2*m))
let cs = cs1 ++ cs2 :> [ipb*(2*m)]ui
let cs = map (\ i -> if i % (2*m) <= 1 then 0 else cs[i-2] )
             (0..<ipb*(2*m))

in baddV4 ls hs |> baddV4 cs
```

Multiplication

Evaluation 1/2

One multiplication of base `uint64_t` in Gu32ops:

Bits	Insts	CGBN	CUDA	FutOldQ2	FutOldQ4	FutNewQ4	FutNewQ2
2^{18}	2^{14}	–	–	–	–	–	–
2^{17}	2^{15}	1	1150	–	–	–	–
2^{16}	2^{16}	35	2039	974	–	1263	1453
2^{15}	2^{17}	116	3471	1674	482	2108	2423
2^{14}	2^{18}	217	5515	2671	693	3264	3697
2^{13}	2^{19}	340	8082	3880	984	4559	4947
2^{12}	2^{20}	526	10475	4931	1281	5467	5786
2^{11}	2^{21}	793	15745	3836	1899	6946	5990
2^{10}	2^{22}	822	16554	2352	2492	7830	6203
2^{9}	2^{23}	496	16888	1122	2798	8134	5646

Multiplication

Evaluation 2/2

Six multiplications of base `uint64_t` in Gu32ops:

Bits	Insts	CGBN	CUDA	FutOldQ2	FutOldQ4	FutNewQ4	FutNewQ2
2^{18}	2^{14}	–	–	–	–	–	–
2^{17}	2^{15}	11	–	–	–	–	–
2^{16}	2^{16}	888	1747	921	–	–	–
2^{15}	2^{17}	2832	2602	1595	350	–	–
2^{14}	2^{18}	4960	2696	1656	513	1211	1609
2^{13}	2^{19}	8625	4961	1872	778	1264	2194
2^{12}	2^{20}	13924	8981	3307	1029	1507	2616
2^{11}	2^{21}	23424	13717	3028	1505	2068	2681
2^{10}	2^{22}	37500	17513	2180	1946	2452	2859
2^{9}	2^{23}	70093	17079	1225	2156	2626	2678

Division

Algorithm 1/3

The intuition behind the division algorithm:

- Multiply the dividend with the of inverse divisor.
- Use shifts to represent the inverse as a big integer.
- Approximate the shifted inverse by Newton's Method using [2].
- Compute quotient and remainder to adjust approximation.

Quotient of big integers by shifted inverse

We define the quotient of big integers $u \leq B^{h \in \mathbb{N}}$ and v in base B using [2] as:

$$u \texttt{ quo } v = \texttt{shift}_{-h}\ (u \cdot \texttt{shinv}_h\ v) + \delta, \quad \textbf{where } \delta \in \{0, 1\} \tag{5}$$

$$\texttt{shift}_{n \in \mathbb{Z}}\ u = \lfloor u \cdot B^n \rfloor \qquad \texttt{shinv}_{n \in \mathbb{N}}\ v = \lfloor B^n / v \rfloor \tag{6}$$

Division

Algorithm 1/3

The intuition behind the division algorithm:

- Multiply the dividend with the of inverse divisor.
- Use shifts to represent the inverse as a big integer.
- Approximate the shifted inverse by Newton's Method using [2].
- Compute quotient and remainder to adjust approximation.

Quotient of big integers by shifted inverse

We define the quotient of big integers $u \leq B^{h \in \mathbb{N}}$ and v in base B using [2] as:

$$u \text{ quo } v = \texttt{shift}_{-h} \left(u \cdot \texttt{shinv}_h \; v \right) + \delta, \quad \textbf{where } \delta \in \{0, 1\} \tag{5}$$

$$\texttt{shift}_{n \in \mathbb{Z}} \; u = \lfloor u \cdot B^n \rfloor \qquad \texttt{shinv}_{n \in \mathbb{N}} \; v = \lfloor B^n / v \rfloor \tag{6}$$

Division

Algorithm 1/3

The intuition behind the division algorithm:

- Multiply the dividend with the of inverse divisor.
- Use shifts to represent the inverse as a big integer.
- Approximate the shifted inverse by Newton's Method using [2].
- Compute quotient and remainder to adjust approximation.

Quotient of big integers by shifted inverse

We define the quotient of big integers $u \leq B^{h \in \mathbb{N}}$ and v in base B using [2] as:

$$u \text{ quo } v = \texttt{shift}_{-h} \left(u \cdot \texttt{shinv}_h \ v \right) + \delta, \quad \textbf{where } \delta \in \{0, 1\} \tag{5}$$

$$\texttt{shift}_{n \in \mathbb{Z}} \ u = \lfloor u \cdot B^n \rfloor \qquad \texttt{shinv}_{n \in \mathbb{N}} \ v = \lfloor B^n / v \rfloor \tag{6}$$

Division

Algorithm 1/3

The intuition behind the division algorithm:

- Multiply the dividend with the of inverse divisor.
- Use shifts to represent the inverse as a big integer.
- Approximate the shifted inverse by Newton's Method using [2].
- Compute quotient and remainder to adjust approximation.

Quotient of big integers by shifted inverse

We define the quotient of big integers $u \leq B^{h \in \mathbb{N}}$ and v in base B using [2] as:

$$u \text{ quo } v = \texttt{shift}_{-h}\ (u \cdot \texttt{shinv}_h\ v) + \delta, \quad \textbf{where } \delta \in \{0, 1\} \tag{5}$$

$$\texttt{shift}_{n \in \mathbb{Z}}\ u = \lfloor u \cdot B^n \rfloor \qquad \texttt{shinv}_{n \in \mathbb{N}}\ v = \lfloor B^n / v \rfloor \tag{6}$$

Division

Algorithm 1/3

The intuition behind the division algorithm:

- Multiply the dividend with the of inverse divisor.
- Use shifts to represent the inverse as a big integer.
- Approximate the shifted inverse by Newton's Method using [2].
- Compute quotient and remainder to adjust approximation.

Quotient of big integers by shifted inverse

We define the quotient of big integers $u \leq B^{h \in \mathbb{N}}$ and v in base B using [2] as:

$$u \ \mathtt{quo} \ v = \mathtt{shift}_{-h} \ (u \cdot \mathtt{shinv}_h \ v) + \delta, \quad \textbf{where } \delta \in \{0, 1\} \tag{5}$$

$$\mathtt{shift}_{n \in \mathbb{Z}} \ u = \lfloor u \cdot B^n \rfloor \qquad \mathtt{shinv}_{n \in \mathbb{N}} \ v = \lfloor B^n / v \rfloor \tag{6}$$

Division

Algorithm 2/3

The algorithm of [2] is based on the Netwon iteration:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} = x_i + \left(x_i - \frac{v}{u}x_i^2\right), \quad \textbf{where } x \in \mathbb{R} \textbf{ and } f(x) = \frac{u}{x} - v \tag{7}$$

It is modified in [2] w.r.t. three aspects:

- It is discretized to integers, so we compute $w \in \mathbb{Z}$ rather than $x \in \mathbb{R}$.
- u is specialized to B^h (where B is the base and $u \leq B^h$).
- Use a shift rather than a division since $v/(B^h) = v \cdot B^{-h} \geq \texttt{shift}_{-h}\, v$.

The Newton iteration becomes:

$$w_{i+1} = w_i + shift_{-h}\left(shift_h\ w_i - v \cdot w_i^2\right) = w_i + w_i \lfloor B^h - v \cdot w_i \rfloor B^{-h} \tag{8}$$

Division

Algorithm 2/3

The algorithm of [2] is based on the Netwon iteration:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} = x_i + \left(x_i - \frac{v}{u}x_i^2\right), \quad \textbf{where } x \in \mathbb{R} \textbf{ and } f(x) = \frac{u}{x} - v \qquad (7)$$

It is modified in [2] w.r.t. three aspects:

- It is discretized to integers, so we compute $w \in \mathbb{Z}$ rather than $x \in \mathbb{R}$.
- u is specialized to B^h (where B is the base and $u \leq B^h$).
- Use a shift rather than a division since $v/(B^h) = v \cdot B^{-h} \geq \texttt{shift}_{-h}\ v$.

The Newton iteration becomes:

$$w_{i+1} = w_i + shift_{-h}\left(shift_h\ w_i - v \cdot w_i^2\right) = w_i + w_i\lfloor B^h - v \cdot w_i \rfloor B^{-h} \qquad (8)$$

Division

Algorithm 2/3

The algorithm of [2] is based on the Netwon iteration:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} = x_i + \left(x_i - \frac{v}{u}x_i^2\right), \quad \textbf{where } x \in \mathbb{R} \textbf{ and } f(x) = \frac{u}{x} - v \qquad (7)$$

It is modified in [2] w.r.t. three aspects:

- It is discretized to integers, so we compute $w \in \mathbb{Z}$ rather than $x \in \mathbb{R}$.
- u is specialized to B^h (where B is the base and $u \leq B^h$).
- Use a shift rather than a division since $v/(B^h) = v \cdot B^{-h} \geq \texttt{shift}_{-h}\ v$.

The Newton iteration becomes:

$$w_{i+1} = w_i + shift_{-h}\left(shift_h\ w_i - v \cdot w_i^2\right) = w_i + w_i \lfloor B^h - v \cdot w_i \rfloor B^{-h} \qquad (8)$$

Division

Algorithm 2/3

The algorithm of [2] is based on the Netwon iteration:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} = x_i + \left(x_i - \frac{v}{u}x_i^2\right), \quad \textbf{where } x \in \mathbb{R} \textbf{ and } f(x) = \frac{u}{x} - v \qquad (7)$$

It is modified in [2] w.r.t. three aspects:

- It is discretized to integers, so we compute $w \in \mathbb{Z}$ rather than $x \in \mathbb{R}$.
- u is specialized to B^h (where B is the base and $u \leq B^h$).
- Use a shift rather than a division since $v/(B^h) = v \cdot B^{-h} \geq \texttt{shift}_{-h}\ v$.

The Newton iteration becomes:

$$w_{i+1} = w_i + shift_{-h}\left(shift_h\ w_i - v \cdot w_i^2\right) = w_i + w_i \lfloor B^h - v \cdot w_i \rfloor B^{-h} \qquad (8)$$

Division

Algorithm 2/3

The algorithm of [2] is based on the Netwon iteration:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} = x_i + \left(x_i - \frac{v}{u} x_i^2\right), \quad \textbf{where } x \in \mathbb{R} \textbf{ and } f(x) = \frac{u}{x} - v \qquad (7)$$

It is modified in [2] w.r.t. three aspects:

- It is discretized to integers, so we compute $w \in \mathbb{Z}$ rather than $x \in \mathbb{R}$.
- u is specialized to B^h (where B is the base and $u \leq B^h$).
- Use a shift rather than a division since $v/(B^h) = v \cdot B^{-h} \geq \mathtt{shift}_{-h}\ v$.

The Newton iteration becomes:

$$w_{i+1} = w_i + shift_{-h}\left(shift_h\ w_i - v \cdot w_i^2\right) = w_i + w_i \lfloor B^h - v \cdot w_i \rfloor B^{-h} \qquad (8)$$

Division

Algorithm 3/3

Instead of showing the algorithm for computing the division and the shifted inverse, let us look at the Futhark implementation.

In the thesis, the Futhark implementation was invalid.

Now it validates!

(Without the divisor prefixes and shorter iterates optimizations.)

Hence all arithmetics are in full length.

The analysis in [2] gives work $O(\log(h-k)(M(h)+M(|h/2-k|)))$.

If we assume $h = m + k$, we get $O(\log(m)M(m))$.

Division

Algorithm 3/3

Instead of showing the algorithm for computing the division and the shifted inverse, let us look at the Futhark implementation.

In the thesis, the Futhark implementation was invalid.

Now it validates!

(Without the divisor prefixes and shorter iterates optimizations.)

Hence all arithmetics are in full length.

The analysis in [2] gives work $O(\log(h-k)(M(h)+M(|h/2-k|)))$.

If we assume $h=m+k$, we get $O(\log(m)M(m))$.

Division

Algorithm 3/3

Instead of showing the algorithm for computing the division and the shifted inverse, let us look at the Futhark implementation.

In the thesis, the Futhark implementation was invalid.

Now it validates!

(Without the divisor prefixes and shorter iterates optimizations.)

Hence all arithmetics are in full length.

The analysis in [2] gives work $O(\log(h-k)(M(h)+M(|h/2-k|)))$.

If we assume $h = m + k$, we get $O(\log(m)M(m))$.

Division

Implementation 1/5

Revisions to section 9.2 and section 9.3 of the thesis:

- 2 guard digits are sufficient.
- The thesis states that $v > B^h$ corresponds to:

$$\exists i \in \mathbb{N}.\ (h < i < m \wedge v[i] \neq 0) \vee (h = i < m \wedge v[i] > 1) \qquad (9)$$

This is not true. E.g. it fails on $[1, 0, 0, 1] > B^3$.

Instead, define $v > B^h$ as $\neg(v < B^h \vee v = B^h)$.

Define $v < B^h$ as $\forall i \in \mathbb{N}.\ i \geq h \vee v[i] = 0$.

Define $v = B^h$ as $\forall i \in \mathbb{N}.\ (i = h \wedge v[i] = 1) \vee (i \neq h \wedge v[i] = 0)$.

Division

Implementation 1/5

Revisions to section 9.2 and section 9.3 of the thesis:

- 2 guard digits are sufficient.
- The thesis states that $v > B^h$ corresponds to:

$$\exists i \in \mathbb{N}.\ (h < i < m \wedge v[i] \neq 0) \vee (h = i < m \wedge v[i] > 1) \qquad (9)$$

This is not true. E.g. it fails on $[1, 0, 0, 1] > B^3$.

Instead, define $v > B^h$ as $\neg(v < B^h \vee v = B^h)$.

Define $v < B^h$ as $\forall i \in \mathbb{N}.\ i \geq h \vee v[i] = 0$.

Define $v = B^h$ as $\forall i \in \mathbb{N}.\ (i = h \wedge v[i] = 1) \vee (i \neq h \wedge v[i] = 0)$.

Division

Implementation 1/5

Revisions to section 9.2 and section 9.3 of the thesis:

- 2 guard digits are sufficient.
- The thesis states that $v > B^h$ corresponds to:

$$\exists i \in \mathbb{N}.\ (h < i < m \wedge v[i] \neq 0) \vee (h = i < m \wedge v[i] > 1) \tag{9}$$

This is not true. E.g. it fails on $[1, 0, 0, 1] > B^3$.

Instead, define $v > B^h$ as $\neg(v < B^h \vee v = B^h)$.

Define $v < B^h$ as $\forall i \in \mathbb{N}.\ i \geq h \vee v[i] = 0$.

Define $v = B^h$ as $\forall i \in \mathbb{N}.\ (i = h \wedge v[i] = 1) \vee (i \neq h \wedge v[i] = 0)$.

Division

Implementation 2/5

```
def div [m] (u: [m]ui) (v: [m]ui) : ([m]ui, [m]ui) =
  let h = findh u -- u ≤ B^h
  let k = findk v -- B^k ≤ v < B^(k+1)

  let p = 2*(m + (i64.bool (k <= 1)) + (i64.bool (k == 0)))
  let up = map (\ i -> if i < m then u[i] else 0 ) (iota p)
  let vp = map (\ i -> if i < m then v[i] else 0 ) (iota p)

  let (h, k, up, vp) =
         if k == 1 then (h+1, k+1, shift 1 up, shift 1 vp)
    else if k == 0 then (h+2, k+2, shift 2 up, shift 2 vp)
    else                (h,   k,   up,         vp)

  let q = shinv k vp h |> mul up |> shift (-h) |> take m
  let r = mul v q |> sub u |> fst
  in if not (lt r v)
     then (add q (singleton m 1), fst (sub r v)) else (q, r)
```

Division

Implementation 3/5

```
def shinv [m] (k: i64) (v: [m]ui) (h: i64) : [m]ui =
  assert (k > 1) (
         if gtBpow v h                then new m
  else if gtBpow (muld v 2) h then singleton m 1
  else if eqBpow v k                then bpow m (h - k)
  else
    let V = (toQi v[k-2])
            + (toQi v[k-1] << (i64ToQi bits))
            + (toQi v[k]   << (i64ToQi (2*bits)))
    let W = ((0 - V) / V) + 1
    let w = map (\i -> if i <= 1
                       then fromQi (W >> (i64ToQi (bits*i)))
                       else 0) (iota m)

    in if h - k <= 2 then shift (h - k - 2) w
                     else refine v w h k      )
```

Division

Implementation 4/5

```
def refine [m] (v:[m]ui) (w:[m]ui) (h:i64) (k:i64) : [m]ui =
  let g = 1
  let h = h + g
  let (w, _) =
    loop (w, l) = (shift (h-k-2) w, 2) while h - k > l do
    let w = step h v w 0 l 0
    let l = i64.min (2*l-1) (h-k)
    in (w, l)
  in shift (-g) w
```

```
def step [m] (h: i64) (v: [m]ui) (w: [m]ui)
             (n: i64) (l: i64)   (g:i64)   : [m]ui =
  let (pwd, sign) = powdiff v w (h-n) (l-g)
  let wpwdS = shift (2*n - h) (mul w pwd)
  let wS = shift n w
  in if sign then fst (sub wS wpwdS) else add wS wpwdS
```

Division

Implementation 5/5

```
def powdiff [m] (v: [m]ui) (w: [m]ui)
                (h: i64)   (l: i64)   : ([m]ui, bool) =
  let L = (prec v) + (prec w) - l + 1
  in if (ez v) || (ez w) then (bpow m h, false)
     else if L >= h then sub (bpow m h) (mul v w)
     else let P = multmod v w L
          in if ez P then (P, false)
             else if P[L-1] == 0 then (P, true)
             else sub (bpow m L) P
```

```
def multmod [m] (v: [m]ui) (w: [m]ui) (e: i64) : [m]ui =
  let vw = mul (take e v) (take e w)
  in map (\ i -> if i < e then vw[i] else 0 ) (iota m)
```

Division

Evaluation 1/2

The implementation is not efficient:

- Batch processing results in error:

  "Known compiler limitation encountered. Sorry."

  Can be circumvented with attribute `#[sequential_outer]`.

- It succeeds in generating intra-block version when run with:

  `#[incremental_flattening(only_intra)]`

  It runs significantly slower for sizes greater than 256 digits.

Division

Evaluation 1/2

The implementation is not efficient:

- Batch processing results in error:

  "Known compiler limitation encountered. Sorry."

  Can be circumvented with attribute `#[sequential_outer]`.

- It succeeds in generating intra-block version when run with:

  `#[incremental_flattening(only_intra)]`

  It runs significantly slower for sizes greater than 256 digits.

Division

Evaluation 2/2

Furthermore, runtimes depend on precision rather than size.

Thus, incomparable to the evaluation method for other arithmetics.

However, the implementation has no problems compiling to C code.

Hence, we could use `multicore` backend and compare to GMP.

The difference in runtimes are so big that results are meaningless.

Conclusion: It is inefficient and not comparable to GMP or CGBN.

Division

Evaluation 2/2

Furthermore, runtimes depend on precision rather than size.

Thus, incomparable to the evaluation method for other arithmetics.

However, the implementation has no problems compiling to C code.

Hence, we could use `multicore` backend and compare to GMP.

The difference in runtimes are so big that results are meaningless.

Conclusion: It is inefficient and not comparable to GMP or CGBN.

Division

Evaluation 2/2

Furthermore, runtimes depend on precision rather than size.

Thus, incomparable to the evaluation method for other arithmetics.

However, the implementation has no problems compiling to C code.

Hence, we could use `multicore` backend and compare to GMP.

The difference in runtimes are so big that results are meaningless.

Conclusion: It is inefficient and not comparable to GMP or CGBN.

Conclusion

This thesis has shown:

- How to compute exact addition, classical multiplication, and division of big integers in parallel.
- How to implement the arithmetic operators in Futhark and CUDA C++.
- Efficient implementations of addition and multiplication.
- Correct but inefficient implementation of division, while outlining the required steps to efficiency.

Conclusion

This thesis has shown:

- How to compute exact addition, classical multiplication, and division of big integers in parallel.
- How to implement the arithmetic operators in Futhark and CUDA C++.
- Efficient implementations of addition and multiplication.
- Correct but inefficient implementation of division, while outlining the required steps to efficiency.

Conclusion

This thesis has shown:

- How to compute exact addition, classical multiplication, and division of big integers in parallel.
- How to implement the arithmetic operators in Futhark and CUDA C++.
- Efficient implementations of addition and multiplication.
- Correct but inefficient implementation of division, while outlining the required steps to efficiency.

Conclusion

This thesis has shown:

- How to compute exact addition, classical multiplication, and division of big integers in parallel.
- How to implement the arithmetic operators in Futhark and CUDA C++.
- Efficient implementations of addition and multiplication.
- Correct but inefficient implementation of division, while outlining the required steps to efficiency.

Conclusion

References

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

Stephen M. Watt.
Efficient Generic Quotients Using Exact Arithmetic.
2023.
arXiv:2304.01753.