High-Level Big Integer Arithmetic in Futhark for GPUs

Cosmin E. Oancea
DIKU, University of Copenhagen
Copenhagen 2100, Denmark
cosmin.oancea@di.ku.dk

Stephen M. Watt
Cheriton School of Computer Science, U. Waterloo
Waterloo, Canada
smwatt@uwaterloo.ca

***Abstract*—We report on GPU implementations of block-level addition, subtraction, multiplication and division for midsize integers, with operands of 2^{15} to 2^{19} bits using the high-level functional language Futhark. Comparing with hand-written C++/CUDA versions and CGBN, we identify which functional constructs compile well, where memory placement and sequentialization are effective, and what compiler support is needed. The results show that high-level code can express the algorithms compactly while approaching competitive performance after certain compiler improvements. In particular, we find that automated placement of arrays in GPU register memory is critical for performance.**

I. Introduction

We examine the use of a high-level functional language, Futhark [29], to produce efficient general purpose GPU code for big integer arithmetic. In earlier work [35], [48], we gave algorithms for addition, subtraction, multiplication and division suitable for integers with 2^{15} to 2^{19} bits. These were implemented in C++ using CUDA, but our longer-term objective has been to obtain high-level functional implementations with competitive performance.

Our motivation is that high-level algorithms can be rapidly deployed and modified, can expose optimization opportunities at a higher semantic level, and can clarify the cost of abstraction. In particular, we ask:

- Can higher order functions and automated function fusion give efficient arithmetic on GPUs?
- Can automated memory allocation and object placement be made sufficiently efficient?
- What primitives are needed for an elegant and efficient solution to a classical problem — big integer arithmetic?
- Can such an implementation provide a competitive library for GPU applications?

More generally, we would like to know how much machine-specific detail can be left to a high-level language compiler without losing the essential performance characteristics of the underlying algorithms. This paper provides first answers to these questions.

GPU arithmetic has been studied from low-level multi-precision integer libraries to exact polynomial arithmetic for computer algebra. We highlight here some work relevant to the design choices in this paper. The most directly comparable integer package is NVIDIA's Cooperative Groups Big Numbers library, CGBN [41], which maps each arithmetic instance to a small cooperative group, typically no larger than a "warp" (see Section II). It exploits fast warp-level communication and gives excellent performance for fixed precisions in its intended range, but is less well matched to the larger regime considered here. Other GPU integer work has explored hardware-adder-style carry propagation, tiled quadratic multiplication, and FFT-based multiplication using real transforms or GPU libraries [2], [16], [17]. Related multiple-precision floating-point libraries include CUMP and CAMPARY [33], [40]; they address a different numerical problem, but illustrate the long-standing interest in GPU arithmetic beyond machine precision.

A related line of computer-algebra work studies dense univariate polynomial arithmetic over finite fields. Earlier work used GPU FFTs over finite fields for polynomial multiplication [36] and later showed that, for important degree ranges, plain multiplication, division and GCD can outperform FFT methods once GPU overheads are included [25]. CUMODP developed this direction into a CUDA library for exact dense polynomial arithmetic over finite fields, including subresultants, multipoint evaluation and interpolation [24]. Later work studied GPU transforms over large prime fields, including generalized Fermat prime characteristics [10]. These computations share with integer arithmetic convolution-like products, finite-field transforms, and tradeoffs among asymptotic work, memory traffic, synchronization and kernel-launch overhead. Integers add the complication that carries and borrows must be propagated.

The present paper asks how well these algorithms can be expressed in Futhark, a high-level purely functional array language, and what compiler support is needed for such code to approach hand-written CUDA performance [29]. The remainder is organized as follows: Section II describes the GPU model; Section III reviews Futhark; Section IV discusses compiler improvements related to automatic placement in register memory; Section V presents the algorithms and their Futhark implementations; Section VI evaluates performance and Section VII concludes.

Research reported here was partially supported by the Novo Nordisk Foundation award NNF24OC0090447 and the Natural Sciences and Engineering Research Council of Canada.

II. Machine Model

Our algorithms are expressed in a high-level language compiled to a vendor-neutral GPU model. We use the OpenCL/SYCL terminology: a computation is launched as a set of lightweight "work-items", organized in "work-groups". In CUDA terminology, these correspond to "threads" and "thread blocks". Work-groups are scheduled on GPU compute units or analogous execution resources. Work-items within a work-group may synchronize and use local memory; different work-groups have no portable synchronization within a kernel and communicate through global memory and kernel boundaries.

Global memory is large but slow. Local memory, called "shared memory" in CUDA and "local memory" in OpenCL/SYCL, is scoped to one work-group and is much faster. Private storage, usually registers, is fastest and belongs to one work-item. Efficient programs reuse data in local and/or private storage before writing back. Limited fast storage determines occupancy and problem size.

In our work, a multi-precision integer is an array of M fixed-width digits in base $B = 2^b$, stored in little-endian order, where b is the digit width. We target midsize integers for which one arithmetic instance fits in one work-group: beyond single-sub-group methods, but small enough for operands and key intermediates to reside in fast storage. A logical digit need not correspond to one work-item: each work-item may process Q logical elements, so T work-items cover QT digit positions. This sequentialization reduces synchronization, improves register reuse, and permits larger integers, at the cost of register pressure.

Thus we rely only on work-group synchronization, local memory, sub-group execution, coalesced global memory, and scarce fast storage. This is weaker than a model exposing vendor-specific primitives, but better matches the portability goals of a high-level language implementation.

III. Futhark

Futhark is a purely functional array language described elsewhere [29]. It is named after Elder Futhark, the oldest known runic script, believed to have originated in the region of present-day Denmark. The name comes from "ᚠᚢᚦᚨᚱᚲ", the first six runes of its alphabet, much as "alphabet" is from the first two Greek letters.

Here we give a few minimal points on syntax to make programs legible. Function application uses the notation popularized by ML, so `f a b` is `f` applied to `a`, giving a function then applied to `b`. This does not preclude giving a single tuple-valued argument, *e.g.* `f(a,b)`. The pipe operator, written ◁ or `<|`, gives the value on the left as an argument to the right, so `a <| f` is `f(a)`.

Data-parallel array languages—APL [31], [32], Accelerate [9], [15], [55], DaCe [3], [4], [59], Futhark, JAX [7], [18], Lift [22], [52], SAC [19]–[21]—express computations by nesting and composing second-order array combinators, such as `map`, `reduce`, `scan` (prefix sum), `scatter` (unstructured write), and sequential loops as well. In short, they aim to support parallel-correctness guarantees and

```
def iota (n: i64) : [n]i64 = ...-- [0,1,...,n-1]
def replicate 't (n:i64) (v:t) : [n]t =...-- [v,...,v]
def zip  [n]'α'β (a: [n]α) (b: [n]β) : [n](α,β)= ...
def unzip[n]'α'β (aos: [n](α,β)) : ([n]α, [n]β)= ...
def map [n] 'α 'β (f: α → β) (a: [n]α) : [n]β
          --  [ f a[0], ..., f a[n-1] ]
def map2[n]'α'β'γ(f:α → β → γ)(a:[n]α)(b:[n]β): [n]γ
      -- [ f a[0] b[0], ..., f a[n-1] b[n-1] ]
def scan[n]'α (⊙:α → α → α)(ne⊙:α)(a:[n]α) : [n]α
    -- [ a[0], a[0]⊙a[1], ..., a[0]⊙ ... ⊙a[n-1] ]

loop x̄ = x̄^init for i < e^n do e^body(i,x̄)  ≡  f 0 x̄^init
 where f i x̄ = if i==e^n then x̄ else f (i+1) e^body(x̄,i)

def write[n]'α(x:*acc([n]α))(i:i64)(v:α):*acc([n]α)
def reduce_by_index_stream [k]'α'β (dest: *[k]α)
 (⊙:α → α → α)(ne⊙:α) (f:*acc([k]α)→ β →acc([k]α))
 (bs: []β) : *[k]α = ...
def scatter_stream [k]'α'β (dest: *[k]α)
  (f: *acc([k]α) → β → acc([k]α)) (b:[]β): *[k]α=...

-- Example of using scatter_stream, write and loop:
def shift[m][q] (n:i64)(xs:[m][q]uint): *[m*q]uint=
  let f (A: *acc([m*q]uint)) tid : acc([m*q]uint) =
    loop A for i < q do
      let off = q * tid + i + n
      let (index,value) = if off >= 0 && off < m*q
         then (off, xs[tid,i]) else (m*q-off+n-1, 0)
      in  write A index value
  in scatter_stream (replicate (m*q) 0) f (iota m)
```

Figure 1. Futhark constructs & demonstration on shifting a 2D array

an elegant expression that is close to the algorithm, while still offering good performance. We refer the reader to [54] for a comparative study of some of these languages.

The types and semantics of the constructs used in this paper are illustrated in Figure 1: `iota` creates an iteration space and `zip` and `unzip` convert between the structure-of-array (SoA) and array-of-structure (AoS) representations. `map` applies a function to each array element, and inclusive `scan` computes all the prefixes of the input array with an *associative* operator ⊙, which has neutral element $ne_\odot$ (exclusive scan has $ne_\odot$ as first element). `map2`, `map3` are like `map`, but apply the function to corresponding elements from two and three input arrays, respectively, e.g.,

```
def map2 f as bs = map (λ(a, b) → f a b) ◁ zip a b
```

A do-loop expression declares a loop counter i taking values in $0 \ldots e^n$ - 1 and a tuple of loop-variant variables $\overline{x}$ and their initializing expressions $\overline{x}^{init}$; the result of the body expression $e^{body}(i, \overline{x})$ is bound to $\overline{x}$ for the next iteration. Since shadowing of let bindings is allowed, if $\overline{x}$ exists in the scope, it serves as implicit initialization for the syntactic sugar notation **loop** $\overline{x}$ **for** $i < n$ **do** ...

In-place updates are supported by an uniqueness type technique [56] using the syntax **let** $x' = x$ **with** $[\overline{e}^{ind}] = e^{val}$ that also accepts the syntactic sugar **let** $x[\overline{e}^{ind}] = e^{val}$.

`reduce_by_index_stream` is essentially a principled, type-safe abstraction [50] of generalized reduction [43]: it is similar to `map (f dest) bs`, except that f can only access its first argument (initialized to `dest`) by means of the `write` function defined above. The latter (atomically) updates the value of some index `i` with the value obtained

by applying the *associative* and *commutative* operator $\odot$ to the existing value at `i` and the new value `v`.

Please note that up to this point all constructs guarantee correct-by-construction parallelism, i.e., data races are not possible. The `scatter_stream` SOAC is similar, except that it overwrites the value at that index instead of accumulating to it (since $\odot$ is not passed as argument). Similar to `scatter`, its use is unsafe—WAW dependencies are possible—and dynamic verification is prohibitively expensive; however recent work [30] has presented promising results for static verification of common cases of scatter.

The bottom of Figure 1 demonstrates the use of loops and `scatter_stream` to flatten and shift the elements of a 2D array by `n` positions, where `n` is also allowed to be negative: the 1D result array of length `m*q` is initialized with zero and each of the `m` threads (`tid`) executes the loop that writes each of its q elements (`xs[tid]`) at the final (shifted) position. Please note the use of sized-dependent types [1], [28]; they are realized by syntactic matching, e.g., passing `replicate (q*m) 0` as first argument to `scatter_stream` would result in a type error, because `[m*q]uint` is syntactically different than `[q*m]uint`.

This can be remedied by runtime verified casting `let x'= x :> [m][q]uint`. Sized types are useful, e.g., in specifying that efficient multiplication requires an even sequentialization factor `[m][2*q]uint`. Finally, Futhark provides a simple FFI to mainstream languages [26], but without support for parameterized components [11], [47].

Futhark's optimizing compiler is primarily aimed at GPU execution. The key code transformation is to map an arbitrary number of levels of application parallelism to the fixed one supported by the hardware. This is achieved by a technique named *incremental flattening* [29], [38] that systematically applies map fission and map-loop interchange to create perfect parallel nests, and generates multiple code versions, which utilize incrementally more levels of application parallelism. These code versions are composed by guarding them with predicates that compare their degree of utilized parallelism to a threshold, which is subject to autotuning. Importantly, some of the created kernels, named *intragroup*, map inner parallelism to the CUDA block level, such that all intermediate arrays are allocated in shared memory, which offers better bandwidth and latency than global memory. In this case, the guard is augmented with a test that succeeds when the shared-memory footprint of the kernel fits the hardware; otherwise a different code version is chosen.

A final relevant optimization is *memory merging* [37], [39], which refers to applying a register-allocation-like algorithm to optimize the reuse of shared-memory buffers, rather than scalars. This is key to enabling large intragroup kernels, since its shared memory must be allocated before running the kernel and a pure setting dictates that arrays are commonly freshly created, e.g., by `map`, `scan`. Notably, we use `opaque` to explicitly prevent consumer-producer fusion and completely disable horizontal fusion [27], because it hinders the reuse of shared memory.

```
1  def demo [n][q] (ass: [n][q]u64) =
2   let (reg: [n][q]u64, shm: [n][2]u64) = unzip
3       ◁ #[toregmem(1)] map f ass
4    ...
5   let x=map(λtid → let z=reg[tid] in g z)(iota n)
6   let yss = map (λ tid →
7      let y= if tid==0 then 0 else shm[tid-1,0] in
8      loop y for i<q do y*y + reg[tid,i]) (iota n)
9    ...
10 def main[m][n][q] (A: [m][n][q]u64) = map demo A
```

Figure 2. Register Demo: `reg` is mapped to registers, `shm` is *not*.

IV. Futhark Compiler Improvements

The intragroup kernels created by incremental flattening allocate all intermediates in shared memory because this is always safe, i.e., accessible by any thread. However, this is inefficient for several reasons: (1) registers have better bandwidth and latency than shared memory, and allow graceful degradation by memory spilling, (2) fast memory is scarce, hence both shared *and* register memory should be used to maximize performance, (3) Futhark performs deep copies when merging arrays across branches and loop iterations–exacerbating shared-memory footprint–possibly resulting in mutually aliased buffers that are not reused.

To remedy this, we have designed a compiler pass that allocates arrays in register memory according to user annotations whenever the analysis can verify that this is safe. The analysis consists of a forward and backward pass that execute in sync during a program traversal: the forward pass extends the context with information referring to scalar expansion, array indirection/slicing, and importantly, with the arrays that are targeted to register allocation. The backward pass checks in a conservative manner that the use (read) of these arrays is compatible with register mapping, namely that (i) the inner-parallel sub-kernel that defined the array has the same parallel dimensions as the (current) one that reads it, and (ii) the outermost indices of the read array correspond to the current sub-kernel indices. If any read violates this pattern, then the array is *not* subject to register allocation.

Other approaches concentrate primarily on improving traffic to shared memory, *e.g.* [53], in particular through the OpenACC [34] or OpenMP [12] frameworks.

Figure 2 shows an example: the `map` at line 10 forms an intragroup kernel, whose parallel sub-kernels correspond to the `map` operations in function `demo`. The first (line 3) advises the compiler to try to allocate results in registers (`#[toregmem(1)]`, where `1` denotes the parallel levels). The backward pass rejects `shm` because its outer index is `tid-1` instead of `tid` at line 7. The analysis succeeds for `reg`, which is indexed with `tid` at lines 8 and 5. Of note, the analysis "remembers" that `z` is the subarray `reg[tid]` but any use of `z` in `g` still conforms to the required pattern.

Ifs and loops have recursive bodies and receive special treatment. A loop variant is mapped to registers if (i) its initializer was meant to be in registers and (ii) its shape is invariant through the loop. If its use in the loop body is not amenable to register mapping, it is

copied to shared memory at the start and loaded to registers at the end—this treatment eliminates the deep-copy and aliasing issues of shared memory. Ifs are treated similarly, but their result `r` needs to be refined through `#[inform_pardim_only(k)] manifest r`, so as to inform the parallel rank `k`. In principle, it seems possible to aggressively map arrays to registers without any annotations.

More advanced analyses than ours were used to disambiguate challenging indexing patterns statically [6], [44], [57] or dynamically [13], [45]. Our analysis treats simpler accesses, which still cover well the common case of Futhark programs. Its scope is not limited to integer arithmetic; in fact, it would significantly accelerate a number of real-world applications written in Futhark [42], [49], [51], and enable efficient radix sort [8] and Flash Attention [14].

V. Big Integer Arithmetic in Futhark

This section discusses Futhark implementations for addition, multiplication, and division. We aim to highlight clean, high-level, hardware-neutral, and pure specifications that (1) are constructed by composing and nesting SOACs, such as map and scan, and guarantee (for most parts) correct-by-construction parallelism, (2) provide good support for abstraction, such as size-dependent types and higher-order functions, and (3) permit optimization of memory placement by register-scheduling directives–such as `#[toregmem(1)]` applied to a map–but are otherwise memory agnostic, thus freeing the programmer from the tedious and challenging task of managing memory. In comparison, CUDA implementations (1) do not guarantee the absence of data races, (2) require manually splitting the application into CPU host code and a number of GPU kernels, and (3) require careful allocation and reuse of arrays into/from the most suited memory level (global, shared, private).

A. Addition

The key idea we build on is that data-parallel addition can be expressed as a map-scan-map composition [5], [48]. Assuming some digit type `uint` and two n-digit unsigned integers x and y, their addition $x+y$ can be computed as in Figure 3. In essence, `map2` $\ominus_1$ `x y` computes whether the per-digit addition overflows or results in the maximal `uint` value. The result (an array of boolean tuples) is passed to an exclusive prefix sum (`scan`^{exc} $\odot$ `(false,true)`) that propagates the carry for each digit, and the last map applies the corresponding carry to the per-digit addition.

Figure 4 shows the optimized code for addition. Function `badd` is intended to compute ipb instances of $n \cdot q$ digit additions within a CUDA block of $ipb \cdot n$ threads, i.e., q is the sequentialization factor. This is hinted by the dependently-sized type `[`$ipb*n$`][`q`]uint` for the array arguments and result at lines 39 and 7. Since arguments *ass* and *bss* are assumed elements of a batch computation—e.g., `map2 badd asss bsss`—they are first copied to register memory at lines 40-41, but only if they reside in

```
1  def ⊖1 a b = (a+b > a, a+b == uint.highest)
2  def ⊙ (o1,h1) (o2,h2) = ((o1&&h2)||o2, h1&&h2)
3  def ⊖3 a b (c, _) = a + b + bool2uint c
4  def badd x y = map3 ⊖3 x y ◁
5            scan^exc ⊙ (false,true) ◁ map2 ⊖1 x y
```

Figure 3. Golden Data-Parallel Addition. The operator ◁ pipes the result of a computation to the last input of the next computation.

```
1  def carryBop (c1: cT) (c2: cT) =
2    if (c2 & 4) != 0 then c2
3    else let res = ( (c1 & (c2 >> 1)) | c2 ) & 1
4         in  res | (c1 & c2 & 2) | (c1 & 4)
5  -- input and result are placed in registers
6  def baddReg [ipb][n][q] (ass: [ipb*n][q]uint)
7      (bss: [ipb*n][q]uint) : [ipb*n][q]uint =
8    let f1 as bs tid : (cT, [q](uint, cT)) =
9      let (carry,rcs) = (2, replicate q (0, 2))
10     let seg_start = tid % n == 0 in
11     loop (carry, rcs) for i < q do
12       let (a, b) = (as[i], bs[i])
13       let r = a + b
14       let c = bool2cT (r < a)
15       let c1= (bool2cT(r == highest_uint))<< 1
16       let c2= (bool2cT(i==0 && seg_start))<< 2
17       let c = c | c1 | c2
18       let carry = carryBop carry c
19       let rcs[i] = (r,c)
20       in  (carry, rcs)
21   let (carry_thds,rcss) = opaque ◁ unzip ◁
22    #[toregmem(1)] map3 f1 ass bss (iota (ipb*n))
23   -- scan across threads, neutral element is 2
24   let carry_thds = scan carryBop 2 carry_thds
25   -- adjust result with thread prefix
26   let f2 rcs tid : [q]uint =
27     let (rs, cs) = unzip rcs
28     let seg_start = tid % n == 0
29     let carry = if seg_start then 2
30                 else carry_thds[tid-1]
31     let rs' = #[scratch] replicate q 0 in
32     (loop (rs', carry) for i < q do
33        let rs'[i]= rs[i] + cT2uint (carry & 1)
34        in (rs', carryBop carry cs[i])).0
35   in  opaque ◁ unzip ◁ #[toregmem(1)]
36       map2 f2 rcss (iota (ipb*n))
37 -- ass and bss may reside in global memory
38 def badd [ipb][n][q] (ass : [ipb*n][q]uint)
39     (bss : [ipb*n][q]uint) : [ipb*n][q]uint =
40   let areg = #[glb2reg_only(1)] manifest ass
41   let breg = #[glb2reg_only(1)] manifest bss
42   in  baddReg areg breg
```

Figure 4. Futhark efficiently-sequentialized code for addition.

global memory, as indicated by the `#[glb2reg_only(1)]` annotation. The latter might not be the case due to fusion.

Lastly, computation is discharged to function `baddReg` (line 6), which maintains its arguments and results in registers and implements the efficiently sequentialized version of the code in Figure 3: $f1$ (lines 8-20) corresponds to $\ominus_1$, except that (1) it processes sequentially q consecutive elements and (2) it returns the per-digit addition and carry results (array `rcs`) together with one carry per-thread, denoting the carry propagation across its q elements.

Line 24 computes the prefix sum of the per-thread carry results. Function `carryBop` has semantics similar to $\odot$, except that (1) it encodes a boolean tuple in the first two least-significant bits of an integer (of some type `cT`), and (2) it lifts $\odot$ to operate across addition instances by setting

the third bit at the start of each new instance. Finally, $f2$ (lines 26-34) corresponds to $\ominus_3$: it computes and applies the final carry to each of the q elements owned by a thread.

Please note that the code of `baddReg` allows the arguments `ass` and `bss` and the intermediate array `rcss` to be safely allocated in registers, because accesses do not overlap across threads—e.g., the producer and consumer maps ($f1$ and $f2$) write and read element `rcss[tid]` only with thread `tid` (i.e., in the iterations numbered `tid`).

Finally, we remark that while the code is hardware neutral, memory agnostic, and relatively high level, it is still significantly more complex, albeit much faster, than the one in Figure 3. Achieving optimal expression *and* performance is, in principle, possible by designing and exposing to the user more advanced scheduling primitives [23], which would generate the code of Figure 4 from Figure 3.

B. Multiplication

Assuming two m-digit integers A and B the quadratic multiplication algorithm is summarized by the formula:

$$C_k = \sum_{\substack{i+j=k \\ 0 \le i,j,k < m}} A_i \cdot B_j$$

which, however, does not handle overflow across C's digits.

Figure 5 highlights the code structure of multiplication in Futhark, which follows the strategy proposed in [48]: Function `bmulReg` (line 43) is intended to receive two register-allocated arguments and also produce their multiplication result in register memory. The implementation consists of (i) performing the quadratic convolution (`convShmQ`), which has arguments and results allocated in shared memory, and (ii) adding the results of convolution (`baddReg`). The conversion between shared and register memory is explicitly performed by the functions `cpShm2Reg` and `cpReg2Shm`, defined at lines 1 and 6. The former uses a sequentialized map to load thread elements from shared memory and the latter uses `scatter_stream`, by which each thread writes its Q registers to consecutive positions in a fresh and flat shared-memory buffer.

Function `convShmQ` (lines 37-41) performs the convolution by mapping `threadConv` to each of the $ipb \cdot n$ threads (line 40). Assuming for simplicity $ipb = 1$, each thread tid sequentially computes $2 \cdot Q$ elements of the result in two steps: (1) Q elements in forward direction, i.e., at indices $tid \cdot Q \ \ldots \ (tid+1) \cdot Q - 1$ (line 32), and (2) their Q symmetric-opposite elements, at indices $n \cdot 2 \cdot Q - (tid+1) \cdot Q \ \ldots \ n \cdot 2 \cdot Q - tid \cdot Q - 1$ (line 34).

This scheduling *requires an even sequentialization factor in order to enforce a perfectly balanced workload across threads, in which each thread computes $n \cdot Q$ scalar multiplications*, e.g., the first and last result elements require 1 and $n \cdot Q - 1$ multiplications, respectively, the second and penultimate require 2 and $n \cdot Q - 2$, and so on.

Function `halfConv` computes Q consecutive elements of the result, i.e., half of a thread's work. The first loop

```
1  def cpShm2Reg[n][Q]'t (shm:[n*Q]t): *[n][Q]t =
2    let ff tid = #[sequential]
3      map (λq→ shm[tid*Q+q]) (iota Q)
4    in  #[toregmem(1)] map ff (iota n)
5
6  def cpReg2Shm[n][Q]'t x (reg:[n][Q]t):*[n*Q]t=
7    let shm = #[scratch] replicate (n*Q) x
8    let f (A: *acc([n*Q]t)) tid : acc([n*Q]t)=
9      loop A for q < Q do
10       write A (tid*Q + q) (reg[tid, q])
11   in scatter_stream shm f (iota n)
12
13 def halfConv [n] (Q: i64) (off: i64) (k: i64)
14     (ash: [n]uint)(bsh: [n]uint) : [Q+2]uint =
15   let cs = #[sequential] replicate Q (0, 0, 0)
16   let cs = loop cs for i < k + 1 do
17         let j = k - i in
18         loop cs for q < Q do
19           cs with [q] = oneConvMul off i (j+q)
20                                  ash bsh cs[q]
21   let cs = loop cs for qm1 < Q-1 do
22         let q = qm1 + 1 in
23         loop cs for i < Q-q do
24           cs with [i+q]=oneConvMul off (k+q) i
25                                 ash bsh cs[i+q]
26   in combineQ cs
27
28 def threadConv [x] (Q:i64) (n:i64) (ash:[]uint)
29  (bsh:[]uint) (tid:i64): ([Q+2]uint,[Q+2]uint)=
30   let offset = n * (2 * Q) * (tid / n)
31   let k1 = Q * (tid % n)
32   let lhcs0 = halfConv Q offset k1 ash bsh
33   let k2 = n * (2*Q) - i32.i64 Q - k1
34   let lhcs1 = halfConv Q offset k2 ash bsh
35   in  (lhcs0, lhcs1)
36
37 def convShmQ[s] (n:i64) (Q:i64) (Ash:*[s]uint)
38         (Bsh: *[s]uint) : ([s]uint, [s]uint) =
39   let (lhcss0,lhcss1)= unzip ◁ #[toregmem(1)]
40     map (threadConv Q n Ash Bsh)(iota (ipb*n))
41   in  manifestShm Ash Bsh lhcss0 lhcss1
42
43 def bmulReg[ipb][n][Q] (Areg:[ipb*n][2*Q]uint)
44     (Breg: [ipb*n][2*Q]uint): [ipb*n][2*Q]uint=
45   let (Lsh,Hsh)=convShmQ n Q (cpReg2Shm 0 Areg)
46                              (cpReg2Shm 0 Breg)
47   in  baddReg (cpShm2Reg Lsh) (cpShm2Reg Hsh)
```

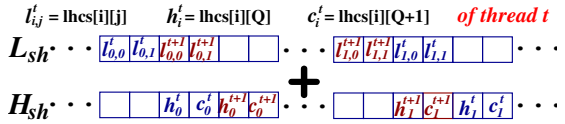

Figure 5. Futhark code for multiplication. The diagram illustrates the placement of thread-private (register) array `lhcs: [2][Q+2]uint` ≅ `(lhcs0,lhcs1)` in shared-memory arrays `Lsh` and `Hsh`, which is achieved by the call to `manifestShm` on line 41. `Lsh` and `Hsh` are then loaded to registers and added at line 47 to complete the algorithm.

nest (lines 16-20) performs a workload equal to that of the first element for all Q elements, and the second loop nest (lines 21-25) performs the missed work for the last $Q-1$ result elements. This structure allows efficient unrolling and scalarization of the inner loop of count Q (line 18) and of the second loop nest, which is performance critical.

The call `oneConvMul o i j A B cs[q]` performs one scalar multiplication $A_{o+i} \cdot B_{o+j}$ that returns a triplet of low, high, and carry digits, which are accumulated to the

current result stored in `cs[q]`, hence `cs: [Q][3]uint`. Importantly, Futhark does not support 128-bit arithmetic: when `uint` has 64 bits we simulate it by performing four 64-bit multiplications, which is less efficient than a CUDA 128-bit multiplication, yielding an application-level slowdown of about 1.33× on highest multi-precision sizes.

After all Q consecutive elements of the result are fully computed, they are aggregated into Q digits plus a high and a carry digit by the call to `combineQ` on line 26, hence the result of a thread's half convolution has type `[Q+2]uint`. It follows that the half-convolution results across all threads, denoted `lhcss0` and `lhcss1` on line 39, have types `[n][Q+2]` and are stored in registers (`#[toregmem(1)]`). Finally, the call to `manifestShm` on line 41 places the `lhcss0` and `lhcss1` arrays in shared-memory arrays `Lsh` and `Hsh`, as illustrated by the bottom picture of Figure 5 and then `Lsh` and `Hsh` are loaded to registers and added at line 47 to complete the algorithm.

C. Division

The Futhark implementation draws inspiration from the CUDA parallelization strategy presented in [35] that implements Watt's algorithm [58]. The latter allows the computation to be carried out entirely in the integral domain and requires at least five full-precision multiplications to compute the quotient and remainder. For more algorithmic detail see the original paper [58].

Figure 6 is intended to give the gist of the Futhark implementation by presenting incomplete code corresponding to the `step` function that is run in a loop and consists of operations on integers whose precision varies through the loop. Since the algorithm cost is dominated by multiplication, we follow the strategy used in [35] that specializes the multiplications to the actual precision of the input and "pads" the linear operations to full precision, i.e., the declared precision of division's input. Examples of the latter include shifting (line 40,41), adding (lines 44,47), subtracting (lines 50), and testing if all least-significant digits up to an index are null (line 45). As a sample, the implementation of `prec`, which computes the precision of a given integer, is shown between lines 25-32: it applies the `reduce_by_index_stream` SOAC (line 31) that uses atomic updates to compute the maximal non-zero index across the per-thread results. The latter are independently aggregated over the `2*Q` elements of each thread in the loop at lines 27-29; a thread performs *an atomic update* only when it owns a non-zero element (using `write` at line 30).

Multiplication in variable precision is implemented by the `varPrecMul` function (lines 8-22), called on line 39 and also in the previous line inside `powDiff` (not shown). The key idea is to *specialize the computation to the required precision of the result*, denoted by the m argument, *by decreasing accordingly the value of the half-sequentialization factor* `Q'`: In the case in which the result precision m is less than or equal to the number of threads n, we use a specialized implementation of multiplication—the `bmul1` call on line 18—that computes an element of the result with each thread, albeit in a thread-unbalanced way.

```
1  def bmulG [n][Q] (Q': i64) (m: i64)
2                   (Ash: *[(1*n)*(2*Q)]uint)
3                   (Bsh: *[(1*n)*(2*Q)]uint) =
4    let (Lsh,Hsh) = convShmQ n Q' Ash Bsh in
5    (cpShm2RegPad m 0 Lsh, cpShm2RegPad m 0 Hsh)
6
7  -- computes the first m digits of A*B
8  def varPrecMul[Q][n] (m:i64)(Areg: [n][2*Q]uint)
9             (Breg: [n][2*Q]uint) : [n][2*Q]uint =
10   let Ash= cpReg2Shm (Areg :> [1*n][2*Q]uint)
11   let Bsh= cpReg2Shm (Breg :> [1*n][2*Q]uint)
12   let Q' = if m <= 1*n then 0 else
13            if m <= 2*n then 1 else
14            if m <= 4*n then 2 else Q
15   let (Lreg,Hreg) =
16    #[inform_pardim_only(1)] manifest
17    (match Q'--bmul1 is specialized to compute
18     case 0 → bmul1 m Ash Bsh--one res/thread
19     case 1 → bmulG 1 m Ash Bsh -- uses Q'= 1
20     case 2 → bmulG 2 m Ash Bsh -- uses Q'= 2
21     case _ → bmulG Q m Ash Bsh)-- uses Q'= Q
22   in (baddReg Lreg Hreg) :> [n][2*Q]uint
23
24 -- computes max non-zero index of xss plus 1
25 def prec [n][q] (xss : [n][q]uint) : i32 =
26   let f (A: *acc([1]i32)) tid : acc([1]i32) =
27     let ind = loop ind = 0i32 for i < q do
28               if xss[tid, i] == 0 then ind
29               else i32.i64 (q * tid + i + 1)
30     in if ind > 0 then write A 0 ind else A
31   in (reduce_by_index_stream (replicate 1 0i32)
32                      (i32.max) 0 f (iota n))[0]
33
34 def step [n][q] (h: i32) (m: i32) (l: i32)
35          (vs: [n][2*q]uint, pv: i32)
36          (ws: [n][2*q]uint) : [n][2*q]uint =
37  let pw = prec ws
38  let (sgn,xs)= powDiff(h-m)(l-2)(vs,pv)(ws,pw)
39  let ys = varPrecMul(pw+prec xs) ws xs-- ws*xs
40  let ws_sft = shift m ws          -- ws << m
41  let ys_sft = shift (2*m - h) ys-- ys << 2*m-h
42  in  #[inform_pardim_only(1)] manifest ◁
43    if ( sgn == 1 )
44    then baddReg' ws_sft ys_sft --ws_sft+ys_sft
45    else let isZero = nullUpToInd (2*m - h) ys
46         let ys_sft'= if isZero then ys_sft
47                      else baddOne ys_sft -- +1
48         let ys_sft'= #[inform_pardim_only(1)]
49                      manifest ys_sft' in
50         bsubReg' ws_sft ys_sft'--ws_sft-ys_sft
```

Figure 6. Futhark code used to implement the whole shifted inverse.

Otherwise, we use the value of `Q'` $\in \{1, 2, 4\}$ that best matches the desired precision—see the calls to `bmulG` at lines 19, 20, 21. The function `bmulG` (i) simply calls `convShmQ`—shown in Figure 5 and discussed in Section V-B—with the reduced sequentialization factor `Q'` and (2) loads the results back in register memory. The latter is accomplished by function `cpShm2RegPad`, which is similar to `cpShm2Reg` except that it sets the indices greater or equal to its first argument (m) with a pad value (0). This is necessary because certain multiplications (in `powDiff`) are performed modulo the m-th power of the base, which requires zeroing out said digits of the result.

VI. Evaluation

a) Hardware, Code Versions, Datasets, Availability: The evaluation was run under the Red Hat Enterprise Linux 8.10 operating system on an NVIDIA A100 GPU, which has 6912 cores, 1555 GB/sec peak global-memory bandwidth and 19.5 TFLOP/s `FP32` peak performance.

CudaP denotes our earlier CUDA implementations of addition, multiplication, and division reported in [48] and [35], while CGBN denotes those using the CGBN library. F-Reg and F-Shm denote the (same) Futhark implementation reported in this paper, compiled with and without the register-placement optimization, i.e., F-Shm allocates intermediate arrays exclusively in shared memory.

The datasets are chosen to vary the integer precision in number of bits (`Num Bits`) from 2^{12} to 2^{19} and the number of parallel instances (`Num Insts`) from 2^{20} down to 2^{13}, such that `Num Bits` $\cdot$ `Num Insts` $= 2^{32}$. The type `uint` is instantiated as a 64-bit unsigned integer. Addition and multiplication use random datasets, since the workload is only sensitive to the integer precision, not its value.

Division borrows the setup used in [35]: denoting by M the dataset declared precision in `uint` words, the actual precision of the dividend is set to $M-2$ and the precision of the divisor is randomly selected between 2 and $M/2$. The rationale is that this configuration always performs the maximal number of iterations in the loop that exhibits precision-variant multiplications. The code and benchmarking setup will be made available at [46].

b) Addition Results: Table I shows the memory throughput in **GB/sec** on the two benchmarks examined in [48]: one in which the per-instance work consists of one addition (**1-Add**) and another in which it consists of six additions (**6-Add**). For both, the number of bytes accessed from global memory is considered 3× that of one input, i.e., each byte of the two inputs/result must be read/written at least once. The computation of **6-Add** is fused so that all intermediates are held in fast memory, hence the global-memory accesses are the same as **1-Add**.

The results in Table I show that CGBN offers poor performance on precisions higher than 2^{13} but excellent scalability, in that the throughput of **6-Add** nearly equals that of **1-Add**, i.e., six additions are performed at the cost of one. The **1-Add** performances of CudaP, F-Reg, and F-Shm are close on all but the largest dataset, where F-Reg is about 1.25× and 1.41× faster than CudaP and F-Shm, respectively. However, on **6-Add**, CudaP outperforms F-Reg with factors between $1.04-1.27\times$. We attribute the slowdown to the Futhark compiler performing (i) suboptimal elimination of redundant barriers, and (ii) internal index calculation in 64-bit arithmetic, which is costly when the computation is not entirely memory bound.

Finally, **6-Add** demonstrates the impact of the register-placement optimization both in terms of (i) scalability: F-Shm runs out of shared memory on the largest dataset, and (ii) performance: F-Reg outperforms F-Shm by $1.66-2.4\times$ factors because registers offer better latency and bandwidth than shared memory.

c) Multiplication Results: Table II uses CudaP as a baseline and presents its runtimes and its speedup in comparison to the other three code versions on the two benchmarks reported in [48]: **1-Mul** denotes $a \cdot b$ and **Poly** denotes $(a \cdot a + b) \cdot (b \cdot b + b) + a \cdot b$, and executes four multiplications and three additions. Since multiplication uses the quadratic algorithm and the precision is increased by a factor of 2 while the number of instances is decreased by the same factor, we expect that the runtime increases by a factor of about $\frac{2^2}{2} = 2\times$ with each larger precision.

Key takeaways are threefold: *First*, CudaP is faster than CGBN on precisions greater than or equal to 2^{15} bits.

Second, F-Reg is slower than CudaP with factors between $0.98-1.53\times$. These are misleading because the main reason for slowdown is that CUDA supports efficient 128-bit multiplications while it remains to be supported in Futhark. Porting Futhark's `oneConvMul` implementation to **CudaP** slows it down by ∼1.33× on the larger datasets. This hints that in fact F-Reg is within 95% of CudaP performance on all datasets and may even win on **1-Mul**.

Third, F-Reg has about the same performance as F-Shm, but starts gaining on the larger datasets. As well, F-Shm cannot run **Poly** in precision 2^{18} and higher. In contrast, F-Reg can actually run 2^{13} instances of precision 2^{19} resulting in *scalable* runtimes of 568 and 2272 *ms* on **1-Mul** and **Poly**, respectively. By *scalable* we mean that the runtimes above are, as expected, about 2× larger than the ones of running 2^{14} instances of precision 2^{18}.

d) Division Results: Table III presents the results for one division operation. Columns 3–4, 5–6 and 9–10 report the runtimes of code versions CudaP, F-Reg and CGBN and the factor by which a division is more expensive than a full multiplication of the same code version, abbreviated DIV-OV. The DIV-OV of CudaP is significantly higher but more accurate than the one reported in [35] because the latter uses as baseline a slower multiplication.

The numbers for CudaP and CGBN indicate that CGBN is using a different algorithm for division, since its DIV-OV is about 2.27×, while Watt's algorithm [58] requires at least 5 full multiplications. Column 11 shows that CGBN is faster than CudaP by large factors (3.64-7.36×) for the lower precisions, but does not support precisions larger than 2^{15}.

F-Reg exhibits smaller DIV-OV than CudaP for all precisions other than 2^{16} and 2^{13}. Moreover, column 7 reports that for all precisions other than 2^{16}, F-Reg is slower than CudaP by factors between 0.98-1.37×. Adjusting for CudaP's advantage of efficient 128-bit multiplication, it seems at least probable that with that, F-Reg performance would fall at least within 95% of CudaP, if not overtake it.

The benefits of the register-placement optimization are evident in column 8 that reports that F-Shm cannot run the highest four precisions, and is slower than F-Reg on the others by 1.31-2.37× factors. As with multiplication, F-Reg can run precision 2^{19} with a scalable runtime, which is under 2× that of precision 2^{18}.

Table I

PERFORMANCE OF ADDITION IN **GB/sec**. THE NUMBER OF BYTES ACCESSED FROM GLOBAL MEMORY FOR BOTH **1-Add** AND **6-Add** IS: $3 \cdot \mathbf{NumInsts} \cdot \mathbf{NumBits}/8$, I.E., EACH DIGIT OF THE TWO INPUTS AND RESULT IS ACCESSED ONCE. A100'S PEAK BANDWIDTH IS **1555** GB/S.

Num Bits	Num Insts	1-Add CGBN	1-Add CudaP	1-Add F-Reg	1-Add F-Shm	6-Add CGBN	6-Add CudaP	6-Add F-Reg	6-Add F-Shm
2^{18}	2^{14}	369	1046	1303	922	362	575	552	—
2^{17}	2^{15}	368	1209	1348	1298	353	826	648	277
2^{16}	2^{16}	376	1358	1338	1219	353	861	693	393
2^{15}	2^{17}	329	1363	1341	1344	321	848	675	407
2^{14}	2^{18}	581	1334	1342	1342	546	875	704	405
2^{13}	2^{19}	1238	1350	1341	1343	1207	881	705	406
2^{12}	2^{20}	1329	1359	1337	1338	1189	882	711	405

Table II

PERFORMANCE OF MULTIPLICATION ON ONE MULTIPLICATION AND A POLYNOMIAL INVOLVING FOUR MULTIPLICATIONS AND THREE ADDITIONS. THE **CudaP** COLUMNS REPORT THE RUNTIME IN *ms* OF THE CUDA IMPLEMENTATION FROM [48]. THE COLUMNS DENOTED **CGBN**, **F-Reg** AND **F-Shm** SHOW THE SLOWDOWN *VS.* **CudaP** OF THE CGBN LIBRARY, AND THE FUTHARK CODE THAT IS COMPILED WITH AND WITHOUT SUPPORT OF ALLOCATING INTERMEDIATE BUFFERS IN REGISTER MEMORY.

Num Bits	Num Insts	1-Mul CudaP *ms*	1-Mul CGBN / CudaP	1-Mul F-Reg / CudaP	1-Mul F-Shm / CudaP	Poly CudaP *ms*	Poly CGBN / CudaP	Poly F-Reg / CudaP	Poly F-Shm / CudaP
2^{19}	2^{13}	420.4	—	1.35	—	1769.9	—	1.28	—
2^{18}	2^{14}	207.3	35.1	1.38	1.45	855.5	49.9	1.34	—
2^{17}	2^{15}	105.5	4.48	1.37	1.41	435.0	21.0	1.35	1.38
2^{16}	2^{16}	57.0	1.21	1.31	1.31	225.1	1.22	1.33	1.34
2^{15}	2^{17}	30.6	1.16	1.30	1.30	119.6	1.00	1.34	1.35
2^{14}	2^{18}	19.3	1.01	1.16	1.16	66.9	0.91	1.35	1.35
2^{13}	2^{19}	12.4	0.93	1.10	1.10	40.7	0.82	1.35	1.37
2^{12}	2^{20}	8.9	0.91	0.82	0.82	22.1	0.79	1.36	1.40

Table III

PERFORMANCE OF DIVISION. COLUMNS 3, 5 AND 9 REPORT THE RUNTIMES OF DIVISION FOR THE CODE VERSION **CudaP**, **F-Reg** AND **CGBN**, RESPECTIVELY. COLUMNS DENOTED **Div/Mul** DENOTE THE FACTOR BY WHICH DIVISION IS SLOWER THAN MULTIPLICATION (DIV-OV) FOR THE CODE VERSION OF THE PREVIOUS COLUMN. THE OTHER COLUMNS REPORT SPEEDUP BETWEEN THE SPECIFIED CODE VERSIONS.

Num Bits	Num Insts	CudaP *ms*	Div / Mul	F-Reg *ms*	Div / Mul	F-Reg / CudaP	F-Shm / F-reg	CGBN *ms*	Div / Mul	CudaP / CGBN
2^{19}	2^{13}	—	—	3166.4	5.58	—	—	—	—	—
2^{18}	2^{14}	1404.3	6.77	1626.9	5.70	1.16	—	—	—	—
2^{17}	2^{15}	702.1	6.66	962.0	6.66	1.37	—	—	—	—
2^{16}	2^{16}	395.2	6.94	606.0	8.10	1.53	—	—	—	—
2^{15}	2^{17}	297.9	9.73	312.2	7.82	1.05	2.37	82.0	2.31	3.64
2^{14}	2^{18}	284.2	14.74	278.7	12.49	0.98	1.31	44.3	2.27	6.42
2^{13}	2^{19}	191.2	15.46	254.5	18.75	1.33	1.42	26.0	2.26	7.36

VII. CONCLUSIONS AND FUTURE WORK

We have provided high-level functional implementations of big integer addition, (classical) multiplication and (Newton iteration) quotient in Futhark. These implementations use higher-order functions and resemble the mathematical formulations of the algorithms. The Futhark compiler generates code for GPUs, automating maps, function fusion and memory use to obtain efficiency.

We have compared the execution efficiency of the resulting code to our hand-coded `C++` CUDA implementations of the same algorithms. The high-level Futhark versions for multiplication and quotient typically take about 1.35× as long as the hand-coded CUDA versions. The Futhark version of addition runs at about the same speed or faster than the CUDA version.

We have also compared our implementations to the standard CGBN library for big integer arithmetic on GPUs. While the Futhark versions are slower for small values, for larger values, the Futhark versions are faster for addition and multiplication ($> 2^{14}$ bits for addition, $> 2^{17}$ bits for multiplication). The performance gain increases as values get larger, for example at 2^{18} bits, addition was 1.5 to 3.5 times as fast, multiplication was 26 times as fast, and CGBN could not perform quotients.

This work has provided concrete examples leading to Futhark compiler improvements of general utility. Allocating arrays in register memory, when possible, has proven particularly useful. We expect that further work on big integers will reveal other useful improvements. A next step would be to implement more sophisticated multiplication algorithms.

References

[1] Lubin Bailly, Troels Henriksen, and Martin Elsman. Shape-constrained array programming with size-dependent types. In *Proceedings of the 11th ACM SIGPLAN International Workshop on Functional High-Performance and Numerical Computing*, FHPNC 2023, page 29–41. ACM, 2023.

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

[3] Tal Ben-Nun, Johannes de Fine Licht, Alexandros N. Ziogas, Timo Schneider, and Torsten Hoefler. Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19. ACM, 2019.

[4] Tal Ben-Nun, Linus Groner, Florian Deconinck, Tobias Wicky, Eddie Davis, Johann Dahm, Oliver D. Elbert, Rhea George, Jeremy McGibbon, Lukas Trümper, Elynn Wu, Oliver Fuhrer, Thomas Schulthess, and Torsten Hoefler. Productive performance engineering for weather and climate modeling with python. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '22. IEEE Press, 2022.

[5] Guy E. Blelloch. Scans as primitive parallel operations. In *International Conference on Parallel Processing, ICPP'87, University Park, PA, USA, August 1987*, pages 355–362. Pennsylvania State University Press, 1987.

[6] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 101–113. ACM, 2008.

[7] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, et al. Jax: composable transformations of python+ numpy programs, 2018.

[8] CCCL Development Team. *CCCL: CUDA C++ Core Libraries*, 2023. https://github.com/NVIDIA/cccl.

[9] Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating haskell array codes with multicore gpus. In *Procs. Workshop on Declarative Aspects of Multicore Programming*, DAMP '11, page 3–14. ACM, 2011.

[10] Liangyu Chen, Svyatoslav Covanov, Davood Mohajerani, and Marc Moreno Maza. Big prime field FFT on the GPU. In *Proceedings of the 2017 ACM International Symposium on Symbolic and Algebraic Computation*, ISSAC 2017, pages 85–92. ACM, 2017.

[11] Y. Chicha, M. Lloyd, C. Oancea, and S. M. Watt. Parametric Polymorphism for Computer Algebra Software Components. In *Procs. 6th Int. Symposium on Symbolic and Numeric Algorithms for Scientific Comput.*, pages 119–130. Mirton Publishing House, 2004.

[12] Leonardo Dagum and Ramesh Menon. OpenMP: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[13] Francis Dang, Hao Yu, and Lawrence Rauchwerger. The R-LRPD Test: Speculative Parallelization of Partially Parallel Loops. In *Int. Par. and Distr. Processing Symp. (PDPS)*, pages 20–29, 2002.

[14] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 16344–16359. Curran Associates, Inc., 2022.

[15] Ivo Gabe de Wolff, David P. van Balen, Gabriele K. Keller, and Trevor L. McDonell. Zero-overhead parallel scans for multicore cpus. In *Proceedings of the 15th International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '24, page 52–61. ACM, 2024.

[16] Adrian P. Dieguez, Margarita Amor, Ramon Doallo, Akira Nukada, and Satoshi Matsuoka. Efficient high-precision integer multiplication on the GPU. *The International Journal of High Performance Computing Applications*, 36(3):356–369, 2022.

[17] Niall Emmart and Charles Weems. High precision integer addition, subtraction and multiplication with a graphics processing unit. *Parallel Processing Letters*, 20(4):293–306, 2010.

[18] Roy Frostig, Matthew James Johnson, and Chris Leary. Compiling machine learning programs via high-level tracing. *Systems for Machine Learning*, pages 23–24, 2018.

[19] C. Grelck and K. Trojahner. Implicit Memory Management for SaC. In C. Grelck and F. Huch, editors, *Implementation and Application of Functional Languages, 16th Int. Workshop, IFL'04*, pages 335–348. University of Kiel, Institute of Computer Science and Applied Mathematics, 2004. Technical Report 0408.

[20] Clemens Grelck and Sven-Bodo Scholz. Merging compositions of array skeletons in SAC. *Journal of Parallel Computing*, 32(7+8):507–522, 2006.

[21] Clemens Grelck and Sven-Bodo Scholz. SAC - A functional array language for efficient multi-threaded execution. *Int. J. Parallel Program.*, 34(4):383–427, 2006.

[22] Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. High performance stencil code generation with lift. In *Int. Symposium on Code Generation and Optimization (CGO)*, CGO 2018, page 100–112. ACM, 2018.

[23] Mary Hall, Cosmin E. Oancea, Anne C. Elster, Ari Rasch, Sameeran Joshi, Amir Mohammad Tavakkoli, and Richard Schulze. Scheduling language chronology: Past, present, and future. *ACM Trans. Archit. Code Optim.*, 22(3), 2025.

[24] Sardar Anisul Haque, Xin Li, Farnam Mansouri, Marc Moreno Maza, Wei Pan, and Ning Xie. Dense arithmetic over finite fields with the CUMODP library. In *Mathematical Software – ICMS 2014*, volume 8592 of *LNCS*, pages 725–732. Springer, 2014.

[25] Sardar Anisul Haque and Marc Moreno Maza. Plain polynomial arithmetic on GPU. *Journal of Physics: Conference Series*, 385:012014, 2012.

[26] Troels Henriksen, Martin Dybdal, Henrik Urms, Anna Sofie Kiehn, Daniel Gavin, Hjalte Abelskov, Martin Elsman, and Cosmin Oancea. Apl on gpus: a tail from the past, scribbled in futhark. In *Procs. of Int. Workshop on Functional High-Performance Computing*, FHPC 2016, page 38–43. ACM, 2016.

[27] Troels Henriksen, Ken Friis Larsen, and Cosmin E. Oancea. Design and gpgpu performance of futhark's redomap construct. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY 2016, page 17–24, New York, NY, USA, 2016. Association for Computing Machinery.

[28] Troels Henriksen and Cosmin E. Oancea. Bounds checking: An instance of hybrid analysis. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY'14, page 88–94, New York, NY, USA, 2014. Association for Computing Machinery.

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

[30] Nikolaj Hey Hinnerskov, Robert Schenck, and Cosmin Oancea. Verifying array properties in pure data-parallel programs. *Proc. ACM Program. Lang.*, 10(PLDI), June 2026.

[31] Aaron W. Hsu and Rodrigo Girão Serrão. U-net CNN in APL: exploring zero-framework, zero-library machine learning. In *Proc. ARRAY'23, Orlando, USA, 18 June 2023*, pages 22–35. ACM, 2023.

[32] Aaron Wen-yao Hsu. *A data parallel compiler hosted on the GPU*. PhD thesis, Indiana University, 2019.

[33] Mioara Joldes, Jean-Michel Muller, Valentina Popescu, and Warwick Tucker. CAMPARY: CUDA multiple precision arithmetic library and applications. In *Mathematical Software – ICMS 2016*, volume 9725 of *LNCS*, pages 232–240, Cham, 2016. Springer.

[34] Seyong Lee and Jeffrey S. Vetter. Openarc: open accelerator research compiler for directive-based, efficient heterogeneous computing. In *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing*, pages 115–120, 2014.

[35] Martin B. Marchioro, Aske N. Raahauge, Marc I. Lovenskjold, Cosmin E. Oancea, and Stephen M. Watt. On GPU implementation for multi-precision integer division. In *Proc. 2026 Computer Algebra in Scientific Computing (CASC 2026)*, LNCS to appear. Springer-Verlag, 2026.

[36] Marc Moreno Maza and Wei Pan. Fast polynomial arithmetic on a GPU. *Journal of Physics: Conference Series*, 256:012009, 2010.

[37] Philip Munksgaard. *Static and Dynamic Analyses for Efficient GPU Execution*. PhD thesis, Department of Computer Science, Faculty of Science, University of Copenhagen, 2023.

[38] Philip Munksgaard, Svend Lund Breddam, Troels Henriksen, Fabian Cristian Gieseke, and Cosmin Oancea. Dataset sensitive autotuning of multi-versioned code based on monotonic properties. In Viktória Zsók and John Hughes, editors, *Trends in Functional Programming. TFP 2021. Lecture Notes in Computer Science*, volume 12834, pages 3–23, Cham, 2021. Springer International Publishing. online at: https://link.springer.com/chapter/10.1007/978-3-030-83978-9_1.

[39] Philip Munksgaard, Troels Henriksen, Ponnuswamy Sadayappan, and Cosmin Oancea. Memory optimizations in an array language. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '22. IEEE Press, 2022.

[40] Takatoshi Nakayama and Daisuke Takahashi. Implementation of multiple-precision floating-point arithmetic for GPU computing. In *Proceedings of the 23rd IASTED International Conference on Parallel and Distributed Computing and Systems*, PDCS 2011, pages 343–349. IASTED, 2011.

[41] NVlabs. Cooperative groups big numbers (CGBN) library. https://github.com/NVlabs/CGBN, 2018.

[42] Cosmin E. Oancea, Christian Andreetta, Jost Berthold, Alain Frisch, and Fritz Henglein. Financial software on gpus: between haskell and fortran. In *Proceedings of the 1st ACM SIGPLAN Workshop on Functional High-Performance Computing*, FHPC '12, page 61–72. Association for Computing Machinery, 2012.

[43] Cosmin E. Oancea and Lawrence Rauchwerger. Logical inference techniques for loop parallelization. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, page 509–520, New York, NY, USA, 2012. Association for Computing Machinery.

[44] Cosmin E. Oancea and Lawrence Rauchwerger. Scalable conditional induction variables (civ) analysis. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 213–224, 2015.

[45] Cosmin E. Oancea, Jason W. A. Selby, Mark Giesbrecht, and Stephen M. Watt. Distributed models of thread-level speculation. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, pages 920–927, 2005.

[46] Cosmin E. Oancea and Stephen M. Watt. Midsize integer arithmetic repository. https://github.com/coancea/midint-arithmetic/futhark-reg.

[47] Cosmin E. Oancea and Stephen M. Watt. Domains and expressions: an interface between two approaches to computer algebra. In *Proceedings of the 2005 International Symposium on Symbolic and Algebraic Computation*, ISSAC '05, page 261–268. ACM, 2005.

[48] Cosmin E. Oancea and Stephen M. Watt. GPU implementations for midsize integer addition and multiplication. In *Languages, Compilers, Analysis – From Beautiful Theory to Useful Practice: Essays Dedicated to Alan Mycroft on the Occasion of His Retirement*, LNCS 15500, pages 51–79. Springer-Verlag, 2025.

[49] Cosmin Eugen Oancea, Ties Robroek, and Fabian Gieseke. Approximate nearest-neighbour fields via massively-parallel propagation-assisted k-d trees. In *2020 IEEE International Conference on Big Data (Big Data)*, pages 5172–5181, 2020.

[50] Robert Schenck, Ola Rønning, Troels Henriksen, and Cosmin E. Oancea. Ad for an array language with nested parallelism. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2022.

[51] Dmitry Serykh, Stefan Oehmcke, Cosmin Oancea, Dainius Masiliūnas, Jan Verbesselt, Yan Cheng, Stéphanie Horion, Fabian Gieseke, and Nikolaj Hinnerskov. Seasonal-trend time series decomposition on graphics processing units. In *IEEE Int. Conference on Big Data (BigData)*, pages 5914–5923, 2023.

[52] M. Steuwer, T. Koehler, B. Köpcke, and F. Pizzuti. Rise & shine: Language-oriented compiler design, 2022. arXiv:2201.03611 [cs.PL]. https://arxiv.org/abs/2201.03611.

[53] Delaram Talaashrafi, Marc Moreno Maza, and Johannes Doerfert. Towards automatic openmp-aware utilization of fast GPU memory. In Michael Klemm, Bronis R. de Supinski, Jannis Klinkenberg, and Brandon Neth, editors, *OpenMP in a Modern World: From Multi-device Support to Meta Programming - 18th International Workshop on OpenMP, IWOMP 2022, Chattanooga, TN, USA, September 27-30, 2022, Proceedings*, volume 13527 of *Lecture Notes in Computer Science*, pages 67–80. Springer, 2022.

[54] David van Balen, Tiziano De Matteis, Clemens Grelck, Troels Henriksen, Aaron W. Hsu, Gabriele K. Keller, Thomas Koopman, Trevor L. McDonell, Cosmin Oancea, Sven-Bodo Scholz, Artjoms Sinkarovs, Tom Smeding, Phil Trinder, Ivo Gabe de Wolff, and Alexandros N. Ziogas. Comparing parallel functional array languages: Programming and performance, 2025. arXiv:2505.08906 [cs.PL] https://arxiv.org/abs/2505.08906.

[55] Lars B. van den Haak, Trevor L. McDonell, Gabriele K. Keller, and Ivo Gabe de Wolff. Accelerating nested data parallelism: Preserving regularity. In Maciej Malawski and Krzysztof Rzadca, editors, *Euro-Par 2020: Parallel Processing*, pages 426–442, Cham, 2020. Springer International Publishing.

[56] Marko van Eekelen, Sjaak Smetsers, and Rinus Plasmeijer. Graph rewriting semantics for functional programming languages. In Dirk van Dalen and Marc Bezem, editors, *Computer Science Logic*, pages 106–128, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[57] Anand Venkat, Mary Hall, and Michelle Strout. Loop and data transformations for sparse matrix code. In *ACM SIGPLAN Conf. on Prog. Lang. Design and Impl. (PLDI)*, PLDI '15, page 521–532. ACM, 2015.

[58] Stephen M. Watt. Efficient generic quotients using exact arithmetic. In *ISSAC '23*, pages 535–544. ACM, 2023.

[59] Alexandros Nikolaos Ziogas, Tal Ben-Nun, Guillermo Indalecio Fernández, Timo Schneider, Mathieu Luisier, and Torsten Hoefler. A data-centric approach to extreme-scale ab initio dissipative quantum transport simulations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19. ACM, 2019.