High-Level Implementation of GPU Algorithms for Midsize Integer Addition and Multiplication

Cosmin E. Oancea[1][0000−0001−5421−6876] and
Stephen M. Watt[2][0000−0001−8303−4983]

[1] DIKU, University of Copenhagen, Copenhagen 2100, Denmark
cosmin.oancea@di.ku.dk
[2] Cheriton School of Computer Science, University of Waterloo, Canada
smwatt@uwaterloo.ca

Abstract. This paper explores practical aspects of using a high-level functional language for GPU-based arithmetic on "midsize" integers. By this we mean integers of up to about a quarter million bits, which is sufficient for most practical purposes. The goal is to understand whether it is possible to support efficient nested-parallel programs with a small, flexible code base. We report on GPU implementations for addition and multiplication of integers that fit in one CUDA block, thus leveraging temporal reuse from scratchpad memories. Our key contribution resides in the simplicity of the proposed solutions: We recognize that addition is a straightforward application of scan, which is known to allow efficient GPU implementation. For quadratic multiplication we employ a simple work-partitioning strategy that offers good temporal locality. For FFT multiplication, we efficiently map the computation in the domain of integral fields by finding "good" primes that enable almost-full utilization of machine words. In comparison, related work uses complex tiling strategies—which feel too big a hammer for the job—or uses the computational domain of reals, which may degrade the magnitude of the base in which the computation is carried. We evaluate the performance in comparison to the state-of-the-art CGBN library, authored by NvidiaLab, and report that our CUDA prototype outperforms CGBN for integer sizes higher than 32K bits, while offering comparable performance for smaller sizes. Moreover, we are, to our knowledge, the first to report that FFT multiplication outperforms the classical one on the larger sizes that still fit in a CUDA block. Finally, we examine Futhark's strengths and weaknesses for efficiently supporting such computations and find out that the significant overheads and scalability issues of the Futhark implementation are mainly caused by the absence of a compiler pass aimed at efficient sequentialization of excess parallelism.

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

1 Introduction

The work presented in this paper is ultimately aimed at extending already-parallel programs, written in high-level languages such as Futhark [10], with support for multi-precision arithmetic that runs efficiently on highly-parallel hardware such as GPGPUs. Accelerating such computations would benefit algorithms from various disciplines, such as computer algebra and cryptography.

The current work focuses on addition and multiplication of midsized integers, by which we mean integers of up to about a quarter million bits. These suffice for most applications and, importantly, they fit in one **CUDA** block of threads, or in a "work group" in OpenCL terminology. This hardware level attention allows the implementation to leverage temporal reuse from fast (register+scratchpad) memory, which has significantly lower latency than global memory. In our experiments on Nvidia hardware, this range covers integers as large as about a quarter million bits.

Normally, writing GPU software attending to this sort hardware consideration is time consuming, detailed and error-prone, with inflexible code using specific machine instructions for a particular platform. Our main interest has been to determine whether we may use a high level language to write GPU code that is elegant and flexible while being sufficiently efficient. We have found this is readily achieved. We have not at this stage been concerned with determining the best algorithms for different integer sizes (i.e. classical, versus Karatsuba, versus FFT), nor with using every trick to maximize performance (e.g. Montgomery representation for FFT).

Related Work The most closely related work is the "Cooperative Groups Big Numbers" (**CGBN**) library [24], authored by NvidiaLab, that offers a high-performance implementation for integers up to 32K bits. The key technique used by **CGBN** to achieve top performance is to map an instance of integer computation on at most one warp of threads in order to leverage specialized Nvidia hardware that allows values to be communicated directly between registers (of the warp), i.e., without passing through scratchpad (shared) memory buffers that have significantly higher latency. Other related work aimed at integer in the target range implement

- the carry propagation of addition by mapping complex VLSI designs of hardware adders into equivalent GPU operations [8,9],
- classical (quadratic time) multiplication using tiling strategies [8], but which require atomic updates to shared memory, or/and
- Strassen's algorithm [29] for log-linear-time multiplication by applying FFT in the domain of reals [1,8], which requires the computation to be carried in bases unfriendly to the underlying machine arithmetic (e.g., base 10).

There have been other, earlier implementations of multiple precision integer arithmetic on CUDA, but with different emphasis:

- CAMPARY [13,14] is a C library based on floating point arithmetic. The main point is to extend precision by representing real numbers as the unevaluated sum of several standard machine precision floating-point values.
- CUMP [22,23] is an older work with a primary purpose of accessing CUDA with GMP. Its stated objective was to outperform the GARPREC library [17]. Other similar works were ancestors of the **CGBN** package, with which we compare directly.
- Isupov [12] uses interval techniques to augment residue number arithmetic for operations that rely on magnitude for numbers with upto 4096 bits.
- Chen et al [6] consider FFT using prime fields with generalized Fermat prime characteristics of size 504 and 992 bits to handle integers of practically unbounded size. These numbers will generally span multiple **CUDA** blocks, hence the work is mostly concerned with making sure that access patterns enable coalesced access to global memory. They aim to improve spatial locality, but not temporal locality (re-use from shared memory).

Contributions This paper presents **CUDA** and Futhark implementations for integer addition and multiplication, including both quadratic and Strassen's log-linear time algorithms. The main contribution of our solutions mainly resides in choosing the simplest tool that does the job: We recognize that addition is a straightforward application of scan [2]—a basic-block of parallel programming—for which efficient GPU implementations are folklore [18]. For classical multiplication we use a simple partitioning technique that assigns to each thread a load-balanced computation of entire elements of the result, so that updates do not need to be atomic and are performed directly in low-latency registers. For Strassen (FFT) multiplication, we conduct the computation in the integral domain by using computer-algebra reasoning to find good prime fields that maximize the utilization of machine-supported arithmetic. This allows for example to represent integers using 15 bits of each half word or 31 bits of each word.

In comparison to **CGBN**, our implementation does not relies on specialized hardware instructions and, we surmise, is likely to translate the performance to other hardware from different vendors, such as AMD. Instead of mapping integer operations to execute at warp level, our implementation allows them to occupy as much as an entire **CUDA** block of threads, and relies on the classical technique of efficiently sequentializing parallelism in excess[3] to amortize each access to shared memory across several register accesses.

We evaluate the performance of our implementations in comparison with **CGBN**—on programs performing one addition, one multiplication and fusion of such operations—and report that **CGBN** is faster on integer ranges up to 2^{13} bits, but our **CUDA** implementation gains the upper hand on ranges of $2^{15} - 2^{16}$ bits, and outperforms **CGBN** on integers consisting of 2^{17} and 2^{18} bits. In fairness, **CGBN** offers near-perfect scalability on fused operations.

Our Futhark implementation exhibits significant overheads in comparison to our **CUDA** prototype. The performance bottlenecks are caused by the absence of

[3] In simple words, this refers to having one thread compute in a sequential-efficient fashion several elements of the result instead of just one.

a compiler pass that automatically performs efficient sequentialization. Implementing it by hand is possible in Futhark, but still sub-optimal in several ways: *First*, intermediate results are always mapped by the compiler to shared-memory buffers and there does not exist a way for the programmer to change the mapping to register memory. *Second*, the shared-memory mapping may restrict the maximal size of the integer that fits in a **CUDA** block. *Finally*, the user code implementing efficient sequentialization is likely to degrade the performance of other semantically-equivalent code versions that, for example, support execution even when the integer is too big to fit in one **CUDA** block.

In summary, the contributions of this paper are:

- a demonstration that high level languages (`C++` and Futhark) can be used to implement big integer arithmetic concisely and efficiently for GPU computation,
- simple and efficient GPU implementations for multi-precision addition and multiplication,
- an experimental evaluation that demonstrates significant performance gains in comparison to **CGBN** library on integer sizes in the range of 2^{15} to 2^{18} bits,
- to our knowledge, the first demonstration that FFT-based multiplication outperforms an efficient implementation of the quadratic algorithm on sizes that fit in a **CUDA** block (by factors as high as $5\times$ on the largest size),
- a presentation that (we hope) allows to reproduce the implementations directly from the information in the paper.

Outline This paper is structured in a straightforward fashion: Sections 2, 3 and 4 present our implementations of addition, quadratic and log-linear time multiplication, respectively. Section 5 discusses the strengths and weaknesses of the current Futhark compiler infrastructure for supporting multi-precision arithmetic. Section 6 reports experimental results, and Section 7 concludes.

2 Integer Addition

We represent a big unsigned integer—referred from now on as an integer—as an array a containing M elements of type *uint*, which allows to store $M \cdot 8 \cdot$ `sizeof(uint)` bits. The elements can be seen as the coefficients of a polynomial in (base) $x = 2^{8\cdot\texttt{sizeof(uint)}}$, i.e., $a = a_0 + a_1 \cdot x + \ldots a_{M-1} \cdot x^{M-1}$. In our implementation of addition and multiplication, the result has the same length and element type as the input integers, e.g., $add : [M]uint \rightarrow [M]uint \rightarrow [M]uint$.

Adding two such integers can be accomplished by a (well known) iterative procedure, illustrated on the left side of figure 1, that adds (in a bigger type of double size) the corresponding elements of a and b together with the carry from the previous operation, and then it computes the result element and the carry for the next iteration as the remainder and quotient of the division of the sum to

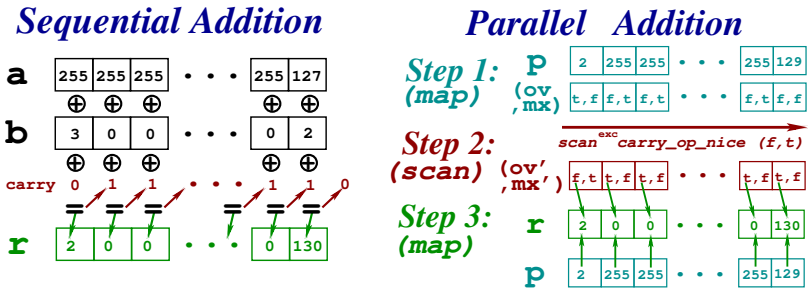

Fig. 1. Sequential and parallel procedures for addition `a+b` base 2^8: linear vs log time.

the integer's base. This is commonly implemented by performing the additions within the domain of *uint* and by checking for overflow.

The procedure described above is inherently sequential, since the carry computation gives rise to a cycle of cross-iteration true dependencies. Furthermore, figure 1 shows a pathological case in which the carry of the first addition is propagated all the way to the last element of the result, which, at first sight, would seem that it does not allow any chunks of computations to be performed in parallel, i.e., independently of each other.

However, as reported in [30] and illustrated on the right side of figure 1, a data-parallel implementation can be straightforwardly obtained by reasoning in terms of the basic blocks of parallel programming—particularly scan [2,4], a.k.a., prefix sum. The procedure requires three parallel steps:

(1) a `map` operation that independently sums up corresponding elements within the *uint* domain, and computes a partial result *uint* $p_i = a_i + b_i$, together with two booleans ov_i and mx_i that record whether (i) the addition has overflow, i.e., there is a carry, and (ii) the result of the addition is the highest element of *uint*, i.e., $ov_i = (p_i < a_i)$ and $mx_i = (p_i == uint.highest)$.
(2) an exclusive `scan` that combines the overflow-highest pairs across elements, and
(3) another `map` that adds the partial result obtained in step (1) with the carry (overflow) result obtained from the scan in step (2).

Before zooming on step (2), we recall the type and semantics of *exclusive* scan:

$$\text{scan}^{exc} \;:\; (\tau \to \tau \to \tau) \;\to\; \tau \;\to [n]\tau \;\to\; [n]\tau$$

$$\text{scan}^{exc} \;\odot\; e_\odot\; [a_1, \ldots, a_n] \;\equiv\; [e_\odot,\; a_1,\; a_1 \odot a_2,\; \ldots,\; a_0 \odot \ldots \odot a_{n-1}]$$

where $\odot$ is an arbitrary *associative* binary operator with neutral element $e_\odot$. Scan has parallel work $O(n)$ and depth $O(\log n)$. Carries can be propagated by means of a scan with an operator that is expressed in a friendly way as below:

```
-- neutral element is (false, true)
def carry_op_nice (ov1: bool, mx1: bool)
                  (ov2: bool, mx2: bool) : (bool, bool) =
    ( (ov1 && mx2) || ov2,  mx1 && mx2 )
```

```
1  -- neutral element is 2 for both carry_op_eff and carry_op_sgm
2  def carry_op_eff (c1: u32) (c2: u32) =
3      (c1 & c2 & 2) | ( ((c1 & (c2 >> 1)) | c2) & 1 )
4
5  def carry_op_sgm (c1: u32) (c2: u32) =
6      if (c2 & 4) != 0 then c2
7      else ( (carry_op_eff c1 c2) | ( (c1 | c2) & 4 ) )
8
9  -- computes IPB instances but without efficient sequentialization
10 def badd [IPB][M] (as : [IPB*M]u32) (bs : [IPB*M]u32) : [IPB*M]u32 =
11     let f a b i =
12         let p = a + b
13         let b = ((u32.bool (i % M == 0)) << 2)
14               | ((u32.bool (p == u32.highest)) << 1)
15               | (u32.bool (p < a))
16         in  (p, b)
17     let (part_res, carry_elms) = map3 f as bs (0..<IPB*M) ▷ unzip
18
19     let carries = scan^exc carry_op_sgm 2 carry_elms -- carry propagation
20
21     let g r c = r + u32.bool ( c & 1 == 1 )
22     in  map2 g part_res carries
23
24 def bbadd [N][IPB][M]
25         (ass : [n][IPB*M]u32) (bss : [n][IPB*M]u32) : [n][IPB*M]u32 =
26     map2 badd ass bss
```

Fig. 2. Futhark pseudocode for performing a batch of `IPB` additions of big numbers, each represented as an array of `M` 32-bit unsigned integers. Function `badd` is supposed to be mapped at CUDA-block level (where arrays are mapped to scratchpad memory). Efficient sequentialization is not shown, albeit it is critical for good performance.

Regarding `(ov1,mx1)` as the current accumulator and `(ov2,mx2)` as the current element of the scanned array, the (sequential) rationale of the operator is:

- An overflow needs to be carried to the next position if either (i) the current addition has produced an overflow, i.e., `ov2` holds, or (ii) the current addition has resulted in a maximal element of `uint` *and* the current accumulator signals an overflow, i.e., `(ov1 && mx2)` holds.
- All elements scanned so far are the maximal element of *uint iff* the current accumulator and the current element are maximal,[4] i.e., `mx1 && mx2`.

Topalovic, Restelli-Nielsen and Olesen prove that the operator is associative [30], and that it accepts `(false,true)` as its neutral element, which is easy to see.

Figure 2 shows a more involved Futhark implementation, denoted `badd`, which is intended to be mapped at the CUDA-block level of parallelism, such that all intermediate arrays are maintained and accessed from fast (scratchpad or register) memory. The pseudocode specializes for simplicity `uint` to 32-bit unsigned integer (`u32`) and computes `IPB` instances of additions in a CUDA block—e.g., to optimize the case when `M` is too small for a good block size. Details are:

[4] While `mx1` is not actually used in the sequential interpretation, it is essential for the parallel execution, which uses a circuit network of depth $O(\log n)$ to compose the results of smaller scanned segments.

- `carry_op_eff` is very similar to `carry_op_nice`, excepts that it packs the tuple of boolean values in the last two bits of an `u32` value. The rationale is that this (i) requires only one scratchpad buffer (instead of two), and (ii) GPU hardware is optimized for `u32` accesses; otherwise one can also use `u8`.
- However, since we aim to compute `IPB` (independent) instances within a CUDA block, step (2) needs to perform a segmented scan instead of a scan. This is typically achieved by lifting the scan's operator to operate over tuples formed by the original datatype and a boolean which, when set, indicates the start of a segment. `carry_op_sgm` encodes the lifted operator of the segmented scan by encoding the start of the segment in the third-last bit.
- the implementation of `badd` follows the three parallel steps mentioned before: the first corresponds to `map3 f` at line 17, which computes the partial result and the input to the (segmented) scan, the second step corresponds to `scan carry_op_sgm 2` at line 19, which propagates the carries, and the third step—`map2 g` at line 22—adds the resulted carries to each partial result.
- the `bbadd` function performs an arbitrary batch (N) of `badd` computations, hence the `map2` at line 26 is intended to be mapped on CUDA's grid of blocks.

A final optimization that we apply (not shown) is the classical *efficient sequentialization of excess parallelism*. In our case, this corresponds to having each thread process independently a parametric (statically-known, smallish) number of elements, rather than just one, as a way of reducing the inter-thread communication overhead for operators such as scan and reduce.[5]

The practical manner of implementing efficient sequentialization in CUDA is by mapping logical arrays to register (thread-private) memory whenever possible, and by using shared memory preferably only as staging buffers—e.g., for copying in coalesced way to/from global to register memory or for storing intermediate results (of reduced size) produced in the internal implementation of `reduce` and `scan`. (Of course, some operations force manifestation of logical arrays in shared memory, e.g., when the same element is read by multiple threads, or in the presence of gather/scatter operations that access statically unknown indices.) This strategy has well known advantages:

- it allows to maximize the utilization of both register and shared memory,
- it promotes accesses from registers, which has lower latency, do not suffer bank conflicts, and are more numerous (in terms of bytes per thread) than shared memory,
- it minimizes the live range of shared-memory buffers, thus promoting their reuse, while register usage is automatically optimized by register allocation.

In summary, efficient sequentialization is a performance critical optimization that reduces inter-thread communication and the latency of memory accesses, generating significant speedups, higher than 2× in cases. More importantly, it

[5] Our CUDA implementation of block-level scan and reduce follows the standard strategy [18] that (de)composes the implementation hierarchically, at each level of the hardware: CUDA thread, warp and block level.

enables the implementation to *efficiently support larger integers*, since in our context, their size is tied with that of the CUDA block: On the one hand, the quantity of resources utilized by a CUDA block is typically proportional with its size, hence a suboptimal mapping will lower the size of the CUDA block that can be launched and thus the magnitude of the integer. One the other hand, CUDA blocks is hard constrained to a maximal number of 1024 threads, hence a one-to-one mapping would limit the integers to $[1024]uint$, while an efficient sequentialization factor $Q = 8$ would support $[8196]uint$. In principle, for addition, the integer size is constrained by registers, not by shared memory, because all logical arrays can be mapped to register memory.

3 Classical, Quadratic Multiplication

Discussion is organized as follows: Section 3.1 provides the high-level rationale of our implementation strategy in comparison with the popular approach of applying tiling to optimize convolution-based code. Section 3.2 gives the birds-eye-view of our CUDA implementation, and section 3.3 zooms in on the main computational step (the convolution).

3.1 Key Insights

The classical algorithm for multiplication corresponds to the formula:

$$C_k = \sum_{\substack{i+j=k \\ 0 \leq i,j,k < M}} A_i \cdot B_j \tag{1}$$

which assumes that the element type $uint'$ of the result C is large enough to prevent overflow. Denoting with $uint$ the element type of A and B, practical implementations commonly perform the product $A_i \cdot B_j$ inside a type $ubig$ which has double the size of $uint$, thus guaranteeing no overflow, and represent the element type of C as a tuple $(ubig, uint32)$ in which the second term denotes the carry that accounts for the potential overflow of summation. Alternatively, one may use a triple $(uint, uint, uint32)$ in which the first two terms correspond to the low and high part of $ubig$.

This section is aimed to highlight the key differences between implementation choices and thus will work directly with formula (1) and ignore the overflow details.

Tiling Approach. Related approaches [1,8,9] predominantly use block tiling to implement formula (1); this results in C-like code similar to the one below, which uses for simplicity the same tile size `T` that is assumed to evenly divide `M`:

```
for(int k = 0; k < M; k++) C[k] = 0;

for (int ii = 0; ii < M); ii+=T)  // mapped on CUDA Grid.y
```

```
1  template<uint, ubig, uint32_t M, uint32_t T> __global__
2  void bmulTiled ( uint* Aglb, uint* Bglb, ubig* Cglb ) {
3      __shared__ uint Ash[T], uint Bsh[T]; __shared__ ubig Csh[2*T];
4      int ii = blockIdx.y*T, i = threadIdx.y; // 0 <= i < T
5      int jj = blockIdx.x*T, j = threadIdx.x; // 0 <= j < T
6      // copy A and B from global to shared memory & initialize Csh
7      if(threadIdx.y == 0) {
8          Ash[j] = Aglb[ii+j]; Csh[j    ] = 0;
9          Bsh[j] = Bglb[jj+j]; Csh[j + T] = 0;
10     }
11     __syncthreads();
12     if(ii+jj + i+j < M) {
13         ubig prod = ((ubig)Ash[i]) * ((ubig)Bsh[j]);
14         atomicAdd(&Csh[i+j], prod);  // atomic in shared memory
15     }
16     __syncthreads();
17     int tid = i*T + j;
18     if(tid < 2*T && ii+jj + tid < M) // atomic in globla memory
19         atomicAdd( &Cglb[ii+jj + tid], Csh[tid] );
20 }
```

Fig. 3. Sketch of a simple CUDA kernel for the tiled version of quadratic multiplication.

```
for (int jj = 0; jj < M; jj+=T) // mapped on CUDA Grid.x
  for (int i = 0; i < T; i++)   // mapped on CUDA Block.y
    for (int j = 0; j < T; j++) // mapped on CUDA Block.x
      if (ii+jj + i+j < M)
        C[ii+jj + i+j] += A[ii+i] * B[jj+j];
```

The tiled code minimizes the temporal reuse distance of the accesses to A, B and C. For example, the read indices of A are invariant to loop `j` (and `jj`). It follows that the slice of `A[ii : ii+T]` can be remapped to a scratchpad memory buffer of length `T` just inside the `jj` loop and reused from there within the body of the loop, i.e., one access to global memory is amortized across T accesses to scratchpad memory. Similar thoughts apply to arrays B and C. However, the loop nest above is a generalized reduction [16,26], whose parallelization requires inter-thread communication, because the same element of `C` may be updated by different threads, hence the additive updates need to be atomic.

Figure 3 sketches a toy CUDA kernel implementing the tiled version, which assumes a two-dimensional block of size `T`×`T`. This approach optimizes temporal locality and enables maximal parallelism but has two shortcomings:

- `C` is not only maintained in shared memory, which has higher latency then registers, but its updates use *expensive* atomic operations: `T`·`T` times from shared memory (line 14 in figure 3) and 2·`T` times from global memory (line 19).
- it prevents producer-consumer fusion—e.g., with following additions and multiplications—because the atomic add in global memory requires a global barrier across all blocks, which is not possible in CUDA other than by ending the execution of the current kernel.

Load Balanced Partitioning of the Result Across Threads. We choose instead a strategy that partitions the elements of the result in a manner that is

$$
\begin{array}{lll}
C_0 & = A_0 \cdot B_0 & \texttt{1 term} \\
C_1 & = A_0 \cdot B_1 + A_1 \cdot B_0 & \texttt{2 terms} \\
C_2 & = A_0 \cdot B_2 + A_1 \cdot B_1 + A_2 \cdot B_1 & \texttt{3 terms} \\
\ldots & \ldots & \\
C_{M-3} & = A_0 \cdot B_{M-3} + \ldots + A_{M-3} \cdot B_0 & \texttt{M-3 terms} \\
C_{M-2} & = A_0 \cdot B_{M-2} + \ldots + A_{M-3} \cdot B_1 + A_{M-2} \cdot B_0 & \texttt{M-2 terms} \\
C_{M-1} & = A_0 \cdot B_{M-1} + A_1 \cdot B_{M-2} + \ldots + A_{M-2} * B_1 + A_{M-1} \cdot B_0 & \texttt{M-1 terms}
\end{array}
$$

Fig. 4. A load-balanced embarrassingly parallel partitioning is to assign thread 0 to compute C_0 and C_{M-1}, thread 1 to compute C_1 and C_{M-2}, thread 2 to compute C_2 and C_{M-3}, and so on. All threads perform a total M multiply-fused add operations.

load balanced, and assigns the computation of an entire partition to the same thread. Figure 4 illustrates the partitioning that computes two elements of the result with each thread, i.e., the identically coloured elements are computed by the same thread and they require the same number (M) of terms. One could increase the sequentialization degree, denoted Q, by computing $Q = 4$ or $Q = 8$ elements of the result per thread, which would result in $2 \cdot M$ and $4 \cdot M$ terms computed by each thread, respectively. This strategy has the advantages that it:

- allows the result C to be mapped to register memory, and to be computed in an embarrassingly parallel fashion, while enabling efficient sequentialization,
- allow multiple addition/multiplication operations to be fused within a CUDA block, such that intermediate results are reused from fast memory.

The downside is that it sequentializes completely an entire parallel dimension of size `M`. We found however that this is a small price to pay, given the advantages, especially when considering that

- we aim at integrating such arithmetic inside programs that are already parallel, which makes it unlikely that the parallel hardware will be starved,
- classical multiplication has suboptimal $O(M^2)$ work, in comparison with the $O(M \ \log M)$ FFT algorithm, hence it makes sense to use it as a niche specialization aimed at squeezing maximal performance from the hardware.

3.2 Birds-Eye View of Implementation

Figure 5 shows the CUDA implementation of the entry function `bmulRegs` that implements the quadratic multiplication algorithm. The implementation denotes by `Q` half the sequentialization factor, i.e., each thread computes 2·`Q` elements of the result, and assumes that 2·`Q` evenly divides `M`, which has been (previously) ensured by padding the numbers to the closest multiple of `Q`.

By this point the input has already been read in coalesced way from global to register memory (not shown), hence the function's input `Areg` and `Breg` and result `Rreg` are stored in register memory. `Ash` and `Bsh` are shared-memory staging buffers of length `[IPB*M]`*uint*. As before, `IPB` denotes the number of instances

```
1  template<class Base,uint32_t IPB,uint32_t M,uint32_t Q> __device__ void
2  bmulRegs( typename Base::uint*Ash,       typename Base::uint*Bsh,
3            typename Base::uint Areg[2*Q], typename Base::uint Breg[2*Q],
4            typename Base::uint Rreg[2*Q]
5  ) {
6      using uint = typename Base::uint;
7      using ubig = typename Base::ubig;
8
9      // 1. copy from register to shared memory
10     cpReg2Shm<uint, 2*Q>( Areg, Ash );
11     cpReg2Shm<uint, 2*Q>( Breg, Bsh );
12     __syncthreads();
13
14     // 2. perform the convolution
15     uint lhcs[2][Q+2];
16     wrapperConv<uint, ubig, M, Q>( Ash, Bsh, lhcs );
17     __syncthreads();
18
19     // 3. publish low parts & high and carry (hcs[:][Q:]) in Lsh & Hsh
20     typename Base::uint *Lsh = Ash, *Hsh = Bsh;
21     publishReg2Shmem<uint, M, Q>( lhcs, Lsh, Hsh );
22     __syncthreads();
23
24     // 4. load back to register and perform the addition of the carries.
25     uint Lrg[2*Q], Hrg[2*Q];
26     cpShm2Reg<uint, 2*Q>( Lsh, Lrg );
27     cpShm2Reg<uint, 2*Q>( Hsh, Hrg );
28     __syncthreads();
29     baddRegs<uint, M, 2*Q, Base::HIGHEST>( Lsh, Lrg, Hrg, Rreg );
30 }
```

Fig. 5. Main CUDA wrapper function that computes quadratic integer multiplication.

solved within a CUDA block, and the integer consists of `M` elements of type `uint`. Thus the size of the CUDA block size is: `IPB*M / (2*Q)`.

The implementation manifests numbers A and B in shared memory (at lines 10-11), because computing the convolution (line 16) corresponding to formula (1) requires multiple threads to access same elements of A and B.

The per-thread result of the convolution (line 16) is the array named `lhcs` which has type $[2][Q+2]uint$. The rationale is that each thread processes two contiguous sub-partitions of Q elements: one from the first half of the integer and its symmetric opposite across the midpoint. The result of each sub-partition is represented as an array of size $Q+2$ of $uint$s that addresses overflow concerns:

- the first Q elements are the low parts of the sequentially aggregated result,
- the next element correspond to the high part of the aggregated result, and the last one to an additional carry (in case the high part overflows).

Next, the function `publishReg2Shmem` manifests the per-thread aggregated `lhcs` result into two shared-memory buffers denoted `Lsh` and `Hsh` in the manner depicted in figure 6. (Note that these are in fact reusing the shared-memory buffers of `Ash` and `Bsh`, which are dead after convolution). Finally, the low, high and carry parts are aggregated across threads by simply adding together, with the procedure from section 2, the integers represented by the arrays L and

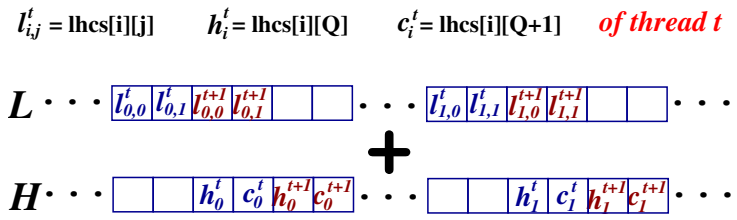

Fig. 6. Illustrating the placement of thread-private array `lhcs`: $[2][Q+2]uint$ in shared-memory arrays L and H for $Q = 2$. L and H are then added to complete the algorithm.

H: This is accomplished by the call to `baddReg` at line 29, which requires its arguments to be remapped as registers (at lines 26 and 27).

3.3 Implementing The Convolution

Figure 7 details the implementation of the convolution step:

- Function `wrapperConv` calls twice function `convolution` at lines 53 and 54, each call processing one of the contiguous sub-partitions corresponding to Q elements of the result,
- Function convolution computes each of the Q results independently in the two loops at lines 35-44. The two loops are necessary since each element of the result requires incrementally more terms.
- Function `convIter`, called at lines 39 and 43 adds one more term to the result. We represent the result at this stage as a tuple between `accum` of type *ubig* (which has double the size of *uint*) and a `carry` of type $uint32_t$. The new term $A_i \cdot B_j$ is computed in the *ubig* type (line 25), which prevents overflow. However the additive update of `accum` may result in overflow, which is accounted for at line 26.
- After all results have been computed, they are aggregated together by the call to `combine` at line 45 which computes half of the `lhcs` array—the other half is computed by the second call to `convolution`.

The reported code allows for all but one loops to be unrolled—the exception is the loop at line 35—thus enabling scalarization of the arrays `lhcs`, `carries`, `accum`. We have found that the best performance is obtained when maximizing the size of *uint*, i.e., when *uint* and *ubig* are instantiated to `uint64_t` and `unsigned __int128`, respectively. This is as expected, since their implementation is sequentially handcrafted for that specific size, and hence offers better performance than our generic algorithm. Finally, our implementation:

- utilizes two shared-memory buffers of size $IPB \cdot M \cdot$ `sizeof(uint)` bytes, which are necessary in order to make the elements of arrays A and B available to all threads during the convolution step.

```
template<class uint,class ubig, uint32_t M,uint32_t Q> __device__
void combine( ubig accums[Q], uint32_t carrys[Q], S lhcs[Q+2] ) {
    const uint32_t SHFT = 8 * sizeof(uint);
    lhcs[0] = (uint) accums[0];
    uint h_res = (uint) (accums[0] >> SHFT);
    uint c_res = carrys[0];

    for(int q=1; q<Q; q++) {
        uint l = (uint) accums[q];
        uint h = (uint) (accums[q] >> SHFT);
        lhcs[q] = l + h_res;
        h_res = h + (c_res + (lhcs[q] < l));
        c_res = carrys[q] + (h_res < h);
    }
    lhcs[Q]   = h_res;
    lhcs[Q+1] = c_res;
}

template<class uint, class ubig> __device__  void
convIter( uint32_t i, uint32_t j, uint* Ash, uint* Bsh,
          ubig& accum, uint32_t& carry
) {
    const uint32_t SHFT = 8*sizeof(S);
    uint accum_prev = (uint) (accum >> SHFT);
    accum += ((ubig)Ash[i]) * ((ubig)Bsh[j]);
    carry += (  ((uint)(accum >> SHFT)) < accum_prev );
}

template<class uint,class ubig, uint32_t M,uint32_t Q> __device__
void convolution(uint32_t k1, uint* Ash, uint* Bsh, uint lhcs[Q+2]) {
    ubig accums[Q];  uint32_t carries[Q];
    for(int q=0; q<Q; q++) {
        accums[q] = 0; carries[q] = 0;
    }
    for(int kk = 0; kk <= k1; kk++) {
        uint32_t i = kk;
        uint32_t j = k1 - i;
        for(int q=0; q<Q; q++)
          convIter<uint,ubig>(i,j+q, Ash,Bsh, accums[q],carries[q]);
    }
    for(int q=1; q<Q; q++) {
        for(int i=0; i<Q-q; i++)
          convIter<uint,ubig>(k1+q,i,Ash, sh,accums[i+q], carries[i+q]);
    }
    combine<uint,ubig,M,Q>(accums, carries, lhcs);
}
template<class uint, class ubig, uint32_t M, uint32_t Q> __device__
void wrapperConv( uint* Ash0, uint* Bsh0, uint lhcs[2][Q+2] ) {
    const uint32_t offset = ( threadIdx.x / (M/(2*Q)) ) * M;
    uint *Ash = Ash0 + offset, *Bsh = Bsh0 + offset;
    uint32_t ltid = threadIdx.x % (M/(2*Q));

    convolution<uint,ubig,M,Q>( Q * ltid,     Ash, Bsh, lhcs[0]); //first half
    convolution<uint,ubig,M,Q>( M-Q*(ltid+1), Ash, Bsh, lhcs[1]); //second half
}
```

Fig. 7. CUDA code for computing the per-thread convolution.

– promotes fusion by holding the logical arrays to registers and using shared-memory buffers transiently for each operation. It follows that fusion can only be hampered by excessive register use.

4 FFT-Based Integer Multiplication

This section presents our implementation for the log-linear time integer multiplication. Several related approaches use the domain of reals to perform the DFFT transformation, which has the potential to affect the accuracy of the computation. Ensuring that errors do not manifest is often handled by restricting the base in which the computation is carried [1,8] to values that are unfriendly to the hardware (e.g., base 10 or 11). Section 4.1 presents the algebraic construction of suitable prime fields that promote efficient DFFT implementation directly in the integral domain by exposing bases that allow near-optimal utilization of the underlying machine arithmetic. While this construction is not necessarily a novelty in the computer-algebra domain, we believe that its application to integer multiplication has merits and it is in the least instructive to the non expert.

Finally, section 4.2 sketches a relatively straightforward implementation of the log-linear time integer multiplication based on the Cooley-Tukey Algorithm [7], which, as before, performs an instance of multiplication with an entire (CUDA) block of threads, such as to allow reuse from fast memory.

4.1 Construction of Integer Prime Fields for DFFT

Finding a good prime field that enables the computation comes down to finding good prime numbers of shape:

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

that accept 2^n distinct roots of unity for a large enough n. Applying the Little Theorem of Fermat, it follows that:

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

By denoting $g = a^k$ for some a, it follows from equation 2 that $g^{2^n} \equiv 1 \pmod{p}$. To compute a g that is a 2^n-th root of unity, one can iterate through the elements a of $\mathbf{Z}_p$ and chose the first one (if any) that also verifies $g^q \neq 1, \ \forall q < 2^n$.

Once such a g was successfully found, one can easily construct a M-th root of unity, named ω, for any M that is a power of 2 and $M < 2^n$:

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

for example, one can easily check that $\omega^M \equiv g^{2^n} \ (mod\ p) \equiv 1 \ (mod\ p)$.

A simple Maple program has revealed in less than 15 minutes of computation the following "good" primes that conform with the desired shape:

*PrimeField*32**:** $(p = 3221225473, k = 3, n = 30, g = 13)$

```
template<typename P> class zmod_t {
    using rep_t  = typename P::uint;
    using ubig_t = typename P::ubig;
    static const rep_t modulus = P::p;
public:
    __host__ __device__ static rep_t norm(const rep_t v)  {
        return (0 <= v && v < modulus) ? v : v % modulus;
    }
    __host__ __device__ static rep_t add (const rep_t x, const rep_t y) {
        ubig_t r = ((ubig_t) x) + ((ubig_t) y);
        if (r >= modulus) r -= modulus;
        return (rep_t)r;
    }
    __host__ __device__ static rep_t sub (const rep_t x, const rep_t y) {
        rep_t r = x;
        if (x < y) r += modulus;
        return (r - y);
    }
    __host__ __device__ static rep_t mul(const rep_t x, const rep_t y) {
        ubig_t r = ((ubig_t) x) * ((ubig_t) y);
        return (r % (ubig_t)modulus);
    } ...
};
```

Fig. 8. A sketch of the prime-field implementation that omits the negation, inversion and power operators.

*PrimeField*64**:** $(p = 4179340454199820289, k = 29, n = 57, g = 21)$

*PrimeField*32 allows (i) to represent an integer (input) as an array of type $[M]uhlf$ with $uhlf$ `= unsigned short`, in which only the first 15 out of 16 bits are utilized, (ii) the FFT computation to be carried on in extended representation $[M]uint$, in which $uint$ `= uint32_t`, and (iii) some of the prime-field computation—such as addition, multiplication, division—need to be performed in a double-sized type $ubig$ = uint64_t. Similarly, *PrimeField*64 uses the following instantiations: $uhlf$ `= uint32_t` such that only the first 31 out of 32 bits are utilized, $uint$ `= uint64_t` and $ubig$ `= unsigned __int128`. The implementation of several of the prime-field operations is shown in figure 8.

4.2 Straightforward Acceleration of Cooley-Tukey Algorithm

Figure 9 gives a relatively-straightforward CUDA implementation of the Cooley-Tukey algorithm, adapted from [15] to use roots of unity in a finite field rather than on the unit circle in the complex plane. The entry function, denoted `bmulFFT`, assumes that M is a power of two and that the total sequentialization factor Q is greater or equal to 2 and it evenly divides M—this is ensured by suitable padding. It follows that the CUDA block size is $\frac{M}{Q}$. The arguments are: `invM` is the (pre-computed) inverse of M in prime field P, `lgM` is the base-2 logarithm of M, `omegas` and `omegas_inv` are (pre-computed) arrays holding the M-roots of unity, i.e.,

```
omegas     = scan^exc (P::mul) 1 (replicate M ω)
omegas_inv = scan^exc (P::mul) 1 (replicate M ω^-1)
```

```
template<typename P, uint32_t M, uint32_t Q> __device__ void
fft ( typename P::uint_t* shmem, uint32_t lgM, typename P::uint_t* omegas,
      typename P::uint_t Areg[2*Q], typename P::uint_t Rreg[2*Q]
) {
    using uint_t = typename P::uint_t; using PF = zmod_t<P>;
    cpReg2Shm<uint_t,2*Q>( Arg, shmem );
    __syncthreads();

    for( int32_t q = 0; q < 2*Q; q++ ) {
        int32_t vtid = threadIdx.x + q*blockDim.x;
        permute<uint_t>( vtid, lgM, shmem );
    }
    __syncthreads();

    for(int32_t t = 1; t <= lgM; t++) {
        uint32_t L = 1 << t,   Ld2 = L >> 1,   r = M >> t;
        for(int32_t q = 0; q < Q; q++) {
            int32_t vtid = threadIdx.x + q*blockDim.x;
            int32_t k = vtid >> (t-1);
            int32_t j = vtid &  (Ld2 - 1);
            int32_t kLj = k*L + j;
            uint_t omega_pow = omegas[r*j];
            uint_t tau       = PF::mul( omega_pow , shmem[kLj + Ld2] );
            uint_t x_kLj     = shmem[kLj];
            shmem[kLj]       = PF::add(x_kLj, tau);
            shmem[kLj + Ld2] = PF::sub(x_kLj, tau);
        }
        __syncthreads();
    }
    cpShm2Reg<uint_t,2*Q>( shmem, Rrg );
    __syncthreads();
}

template<typename P, uint32_t M, uint32_t Q> __device__ void
ifft( typename P::uint_t  invM,     uint32_t lgM,
      typename P::uint_t* shmem,    typename P::uint_t* omegas_inv,
      typename P::uint_t Areg[2*Q], typename P::uint_t Rreg[2*Q]
) {
    fft<P,M,Q>( shmem, lgM, omegas_inv, Areg, Rreg );
    for(int i=0; i<2*Q; i++) Rreg[i] = zmod_t<P>::mul(invM, Rreg[i]);
}

template<typename P, uint32_t M, uint32_t Q> __device__ void
bmulFFT ( typename P::uint  invM,   uint32_t lgM,
          typename P::uint* omegas, typename P::uint_t* omegas_inv,
          typename P::uint* shmem,  typename P::uhlf Ahlf[Q],
          typename P::uhlf Bhlf[Q], typename P::uhlf Rhlf[Q]
) {
    using uint = typename P::uint; using uhlf = typename P::uhlf;
    uint_t Areg[Q], Afft[Q], Breg[Q], Bfft[Q], Treg[Q], Rreg[Q];

    for(int q=0; q<Q; q++) Areg[q] = Ahlf[q];
    fft<P,M,Q/2>(shmem, lgM, omegas, Areg, Afft);

    for(int q=0; q<Q; q++) Breg[q] = Bhlf[q];
    fft<P,M,Q/2>(shmem, lgM, omegas, Breg, Bfft);

    for(int q=0; q<Q; q++) Treg[q] = zmod_t<P>::mul(Afft[q], Bfft[q]);
    ifft<P,M,Q/2>(shmem, invM, lgM, omegas_inv, Treg, Rreg);

    uhlf_t Rlw[Q], Rhc[Q];
    splitFftReg<P,Q>(Rreg, (uhlf*)shmem, Rlw, Rhc);
    baddRegMul2Fft<P, M, 2*Q, 0>( (uhlf*)shmem, Rlw, Rhc, Rhlf );
}
```

Fig. 9. Main CUDA wrapper function that computes FFT multiplication, where the input (`Ahlf`, `Bhlf`) and result (`Rhlf`) are stored in register memory.

Finally, `shmem` is a shared-memory staging buffer of type $[M]uint$, and `Ahlf` and `Bhlf` represent the input arrays of element type $uhlf$, which were already copied in coalesced way from global to register memory. `Bhlf` is the place-holder for the result and it is mapped to register memory as well.

The implementation applies FFT to the input arrays (lines 52-56), then multiplies together the FFT results within the given prime field (line 58), and applies the inverse FFT transform (line 59), whose result is stored in `Rreg`.

Function `splitFftReg`, whose implementation is not shown, is called at line 62 to change the base in which the computation is carried from $2^{8\cdot\texttt{sizeof(uint)}}$ back to $2^{8\cdot\texttt{sizeof(uhlf)}-1}$:

- it first aggregates the Q per-thread results of element type $uint$ into Q low parts, one high part and a carry, all of element type $uhlf$; the procedure is similar to the one described in section 3.2 for classical multiplication,
- then it places the result in a shared-memory buffer in a manner similar to the one depicted in figure 6—except that there is no "symmetrical" part,
- finally, it loads the results back again to register memory, denoted `Rlw` and `Rhc`—such that each thread holds the elements at the same indices from the logical arrays `Rlw` and `Rhc`.

The numbers represented by logical arrays `Rlw` and `Rhc` are finally summed up (carry propagation included) by the call to function `baddRegMul2Fft` at line 63. Since the addition needs to be performed in base $2^{8\cdot\texttt{sizeof(uhlf)}-1}$, we multiply all elements of `Rlw` and `Rhc` by two, perform the addition in the machine base $2^{8\cdot\texttt{sizeof(uhlf)}}$, then divide back by 2 while reapplying the carries. (An element resulted from addition was subject to a carry if its value is odd.)

The implementation of `fft` and `ifft` are standard. FFT-based multiplication can operate with integers having (close to) the same size as the classical multiplication ($\mathcal{CM}$): on the one hand its input uses half the element type of $\mathcal{CM}$ (i.e., $uhlf$ *vs.* $uint$), but on the other hand it uses only one memory buffer of size $M \cdot$ `sizeof(uint)` instead of two buffers with $\mathcal{CM}$. To be exact, the maximal supported size is a bit smaller than the one of $\mathcal{CM}$, because only 15 out of 16 bits can be used with $PrimeField32$ and 31 out of 32 bits with $PrimeField64$.

5 Futhark's Strengths and Weaknesses

5.1 Futhark Optimizations Relevant for Big Integer Arithmetic

Incremental Flattening. Futhark supports expression of parallel programs that operate on regular multi-dimensional arrays. The arbitrarily-nested application parallelism is flattened [3,5], by a technique dubbed "incremental flattening" [11] that utilizes map fission and map-loop interchange to create semantically-equivalent code versions that systematically map more and more levels of application parallelism to the hardware. Essentially, when a new `map f` operation is discovered in the top-down traversal of the program, the analysis:

(1) creates a first code version corresponding to a CUDA kernel in which each thread executes (independently) an application of f.
(2) creates a second code version in which the `map` parallelism discovered so far is mapped on the CUDA grid, and the parallelism inside function f is recursively flattened and mapped to CUDA block level, such that intermediate arrays are mapped to shared memory. This is dubbed an *intra-group kernel*.
(3) the current `map` is added to the parallel context—that represents a perfect nest of `map` operations—and incremental flattening continues recursively.

The resulted code versions are independently optimized and combined into one program by guarding each of them with a predicate that compares compare a dynamic program measure[6] with a threshold. Threshold values are autotuned—so as to select the best combination of code versions—based on deterministic procedure that is guaranteed to produce a near-optimal result in minimal number of runs, as long as the dynamic measure conforms with a monotonic property [20].

We have found that incremental flattening is essential to supporting multi-precision computations in an high-level architecture-neutral language such as Futhark: On the one hand, choice (2) produces the intra-group kernel that we are aiming for, i.e., that leverages the use of fast (shared) memory and supports efficient fusion of such operations. One the other hand, it provides a fail-safe platform: the computation will still be carried on, albeit less efficiently so, on large integers that do not fit the intra-group kernel, by means of the versions generated by choices (3) and (1).

Memory Optimizations. A set of analyses that come handy in our context refer to reducing the memory footprint [19] and at eliminating unnecessary copy operations, dubbed short-circuiting analysis [21]. The former corresponds to applying register-like allocation to operate on memory buffers instead of registers, thus allowing buffers' reuse once their liveness ended. This reduces the shared-memory requirements of the kernel and enables larger integers and fusion.

Short-circuiting analysis [21] addresses an inefficiency common to functional languages whose type systems enforce correct-by-construction parallelism: some parallel loops—e.g., appearing in LUD or Needleman-Wunsch algorithms—cannot be expressed directly because they both read and write (non-overlapping slices of) elements of the same matrix. The typical type system does not perform dependence-analysis on arrays [27,28] and will demand to separate the parallel loop into two parallel operations, typically: a `map` that reads from the original matrix and results into a temporary buffer, and another parallel-write operation that updates the corresponding slice of the matrix with the buffer elements.

The analysis introduces a notion of memory and attempts to map the memory space of the buffer directly to the corresponding memory space of the matrix—whenever it can guarantee safety—such that the parallel write becomes a `noop` and the buffer does not actually allocates any extra space. A trivial example is:

$$\texttt{let}\ x\ =\ \texttt{concat}\ a\ b$$

[6] In practice, the dynamic measure is the degree of parallelism utilized by a kernel.

```
1  let badd [ipb][n] (as: [ipb*(4*n)]u32) (bs: [ipb*(4*n)]u32) : [ipb*(4*n)]u32 =
2    let g = ipb * n
3    let cpGlb2Sh (i : i64) = #[unsafe]
4          ( ( as[i], as[g + i], as[2*g + i], as[3*g + i] )
5          , ( bs[i], bs[g + i], bs[2*g + i], bs[3*g + i] ) )
6
7    let ( ass, bss ) = map cpGlb2Sh (0...<g) ▷ unzip
8    let (a1s, a2s, a3s, a4s) = unzip4 ass
9    let (b1s, b2s, b3s, b4s) = unzip4 bss
10   let ash = a1s ++ a2s ++ a3s ++ a4s
11   let bsh = b1s ++ b2s ++ b3s ++ b4s
12   ...
```

Fig. 10. Futhark code illustrating coalesced copying from global to shared memory for an efficient sequentialization factor $Q = 4$.

If a or b are lastly used in the above statement, then their memory is allocated directly in the corresponding memory space of x and the `concat` becomes a `noop`.

We have used this compiler feature to support efficient sequentialization in our Futhark implementations. Figure 10 shows the prelude of the function that performs integer addition, which is intended to be mapped at CUDA block level. The `map cpGlb2Sh` operation on line 7 reads four elements of `as` and `bs` with each of the `g` threads (of the block) in coalesced way from global to shared memory—i.e., consecutive threads access consecutive words in global memory. The following `concat` operations at lines 10 and 11 put the results together in the correct order in arrays `ash` and `bsh`, which will be mapped to shared memory. Short-circuiting analysis ensures that only two shared-memory buffers are allocated (for the final `ash` and `bsh`), and that the `concat` operation cost nothing—since `a1s`...`a4s` are allocated directly in the memory space of `ash`.

5.2 Shortcomings of Futhark's Compiler Infrastructure

The experimental evaluation, reported in the next section, shows that our Futhark implementation has sub-optimal performance and scalability in comparison to our **CUDA** prototype and the **CGBN** library. The central reason is the absence of a compiler pass aimed at supporting efficient sequentialization. Rationale is:

First, performing efficient sequentialization by hand is not only "un-natural" and results in less elegant code, but, more importantly it has the potential of degrading the performance of the other semantically-equivalent code versions. For example, if the integer size is too large to fit in the intra-group kernel, then the code in figure 10—which was intended to copy in coalesced way from global to shared memory—performs in a convoluted way two expensive and completely unnecessary copies (global-to-global memory).

Second, logical array created inside the intra-group kernel are currently mapped by the compiler to shared-memory only, since this guarantees that their elements are accessible to any threads. This mapping has serios performance implications since shared memory has higher latency than registers, and there is no manner in which this mapping can be altered by the programmer. It also follows that Futhark kernels will requires more shared memory than necessary, which limits

the magnitude of the supported integer. For example, in the case of addition, our **CUDA** prototype utilizes one memory buffer (and this can be further reduced), while Futhark requires twice as much.

Third, the Futhark FFT implementation currently uses a `scatter` (parallel write) operation as the result of a loop (body), which requires two shared-memory buffers that are aliased across the loop (double buffering). This circular aliasing prevents the current compiler to reuse the space of these buffers for subsequent operations, even when their liveness interval has ended. We observe that if the corresponding array was mapped to register memory instead, then only one shared-memory buffer would be necessary, and furthermore the double buffering of that array would be efficiently supported by the register allocation of the underlying compiler (e.g., `nvcc`).

Finally, Futhark does not yet supports an 128-bit integer, which would offer a significant boost to the performance of the classical multiplication.

6 Experiments

This section evaluates the performance of our implementations for addition, multiplication and fusion of such operations. We compare our results with those of the "Cooperative Groups Big Numbers" library,[7] (**CGBN**) authored by NVlabs. that offers a framework for performing unsigned multiple precision integer arithmetic in CUDA. The current release (XMP 2.0 Beta) offers state-of-the-art performance on small to medium sized integers: 2^5 bits through 2^{15} bits, but also seems to support larger integers, albeit without top-performance guarantees.

The key design decision in **CGBN** is that one operation is performed within (at most) one warp of threads, such that the implementation can leverage specialized hardware (instructions) that enable very efficient (low-latency) communication of register values within a warp. This also promotes the scalability of fused operations. In comparison, our implementations does not rely on specialized hardware instructions, and maps an integer instance to be solved by at most one CUDA block of threads. We thus expect to achieve higher performance on larger integers, where **CGBN** is likely to be affected by high-register pressure.

6.1 Hardware, Benchmark, Performance Measures, Methodology

Our evaluation uses an Nvidia A100 GPU that offers 6912 cores, peak global-memory bandwidth of 1.555 TB/sec and `FP32` peak performance of 19.49 Tflops.

We evaluate our implementations in comparison with **CGBN** by running programs that perform batches of (i) one addition **1-Add**, (ii) six additions **6-Add**, (iii) one multiplication **1-Mul**, and (iv) a polynomial computation **Poly** involving four multiplications and two additions. All four programs corresponds to the execution of one kernel such that related sequences of additions and multiplications are performed inside the same CUDA block—i.e., **6-ADD** and **Poly** are

[7] https://github.com/NVlabs/CGBN

intended to evaluate the scalability of block-level fusion, in which intermediate results are maintained in fast memory. These programs are evaluated on eight combinations of values for the size in bits *NumBits* of the integer and the total number of (integer) instances *NumInsts*, such that $NumBits \cdot NumInsts = 2^{32}$.

We report the performance of

addition: in GB/sec—because addition it is memory bound—and we compute the number of bytes accessed for both **1-Add** and **6-Add** with the formula:[8]

$$\texttt{number-of-bytes-accessed} \;=\; 3NumInsts \cdot \frac{NumBits}{8}$$

multiplication: in terms of Giga 32-bit unit operations per second (Gu32ops/sec), since multiplication is compute bound. **1-Mul** uses as number of operations:

$$\texttt{1-Mul-num-u32-ops} = 300 \cdot NumInsts \cdot m \cdot \log m, \text{ where } m = \frac{M \cdot \texttt{sizeof(uint)}}{4}$$

For **Poly** we consider the number of unit operations to be four times that of **1-Mul**, i.e., we only consider the four multiplications and ignore the two additions. The rationale for the constant 300 is that the algorithm performs three FFT transformations, each of them using about $100 \cdot m \cdot \log m$ unit operations.[9] However, the constant does not matter much: the key is that the measure implements a normalized runtime that allows meaningful comparison across different implementations and also across different datasets.

Since all Cuda programs consists of one kernel call, we measure the runtime as the average of 500 kernel runs for **1-Add**, **6-Add** and **1-Mul** and of 125 runs for **Poly**. For Futhark, we use the option `bench -backend=cuda` that (i) measures all overheads except for device initialization, kernel compilation and data transfers between host and device, and (ii) performs enough runs until the 95%-confidence percentile average stabilizes.

6.2 Performance of Addition

Table 1 shows the performance of integer addition expressed as memory bandwidth, i.e., in GB/sec. The peak global-memory bandwidth of the Nvidia A100 hardware is 1.555TB/sec. The first two columns correspond to the number of bits of the integer (*NumBits*) and the total number of instances performed *NumInsts*. The columns denoted **CGBN** correspond to the performance of the **CGBN** library, while the columns denoted by **Our-Cuda** and **Futhark** correspond to the performance of our Cuda and Futhark implementations.

[8] Ideally, both programs read two integers from global memory and write one as result.

[9] We have used test programs to measure the latency of 32 and 64 bit operations such as addition, multiplication, modulo, in comparison with 32-bit integer addition (as the unit), and we have counted that FFT multiplication instantiated to `FftPrime32` requires about 100 units inside its $M \cdot \log M$ loop nest. In fact the modulo operation on `uint64_t` alone accounts for about 78 additions.

Table 1. Performance of Addition in **GB/sec**. A100's peak bandwidth is **1555** GB/s.

Num Bits	Num Insts	1-Add CGBN	1-Add Our-Cuda	1-Add Futhark	6-Add CGBN	6-Add Our-Cuda	6-Add Futhark
2^{18}	2^{14}	369	**1320**	737	362	**570**	294
2^{17}	2^{15}	368	**1331**	1172	353	**803**	350
2^{16}	2^{16}	376	**1358**	**1343**	353	**853**	313
2^{15}	2^{17}	329	**1363**	**1363**	321	**856**	431
2^{14}	2^{18}	581	**1334**	**1370**	546	**836**	434
2^{13}	2^{19}	1238	**1350**	**1364**	**1207**	816	435
2^{12}	2^{20}	**1329**	**1359**	**1364**	**1189**	856	435
2^{11}	2^{21}	1275	**1366**	**1364**	**1167**	855	435

For **CGBN** we set the thread-per-instance parameter for *NumBits* equal to 2^{11} and 2^{12} to 16 and 8, respectively, and to 32 for the rest of *NumBits* values; we have observed that best performance is achieved for these instantiations (for both addition and multiplication). For our Cuda and Futhark implementations, we instantiate *uint* to `uint64_t` and the sequentialization factor Q to 4, i.e., each thread computes $4 \cdot 64 = 256$ bits sequentially.

Key observations derived from table 1 are:

1-Add: both our Cuda and Futhark implementations outperform CGBN on integers whose number of bits are in the interval $2^{14} \ldots 2^{18}$. For example, CGBN commonly achieves less than 25% of the peak bandwidth, while our implementations commonly achieve higher than 85% peak performance. CGBN offers competitive performance on $\mathit{NumBits} = 2^{11} \ldots 2^{13}$.

6-Add: CGBN offers near-perfect scalability, i.e., it takes about the same amount of time to perform six addition as it takes to perform one. We attribute this to the low-latency of specialized-register instructions for transferring values within a warp of threads. However, while **CGBN** offers excellent performance on integers of size $2^{11} \ldots 2^{13}$ bits (up to 1.5× faster than ours), **Our-Cuda** still holds the upper hand on sizes $2^{14} \ldots 2^{18}$ (up to 2.7× faster than CGBN).

6-Add: our CUDA implementation offers decent scaling: except for 2^{18} bits, computing six additions takes less than 1.65× the time of one addition.

6-Add: the scalability of the Futhark implementation is severely handicapped by the layout that maps intermediate arrays in shared-memory buffers rather than registers: six additions require (more than) 3× the time of one addition.

6.3 Performance of Multiplication

Table 2 shows the performance of multiplication expressed in Gu32ops/sec (see section 6.1), where the best two numbers are displayed in bold text—the higher the number the better the performance. Cells filled with **----** denote that kernel

Table 2. Performance of multiplication in Gu32ops/sec; the number of 32-bit operations for **1-Mul** is computed as $300 \cdot \mathit{NumInsts} \cdot m \cdot \log m$, where $m = \frac{M \cdot \texttt{sizeof(uint)}}{4}$. **Poly** computes $(a \cdot a + b) \cdot (b \cdot b + b) + a \cdot b$ using four multiplications and two additions; its number of operations is considered to be four times that of **1-Mul**.

Num Bits	Num Insts	1-Mul CGBN	1-Mul Cu-Q	1-Mul Fu-Q	1-Mul Cu-F	1-Mul Fu-F	Poly CGBN	Poly Cu-Q	Poly Fu-Q	Poly Cu-F	Poly Fu-F
2^{18}	2^{14}	72	----	**1813**	**11590**	----	49	----	**1795**	**11351**	----
2^{17}	2^{15}	997	**4471**	3296	**11789**	----	192	**4027**	3259	**12130**	----
2^{16}	2^{16}	6482	**7843**	5901	**11466**	----	5837	**7148**	5820	**11679**	----
2^{15}	2^{17}	11640	**13460**	10091	**12779**	10187	**12246**	**12247**	9889	**12621**	8395
2^{14}	2^{18}	**21461**	**21608**	15856	14297	11259	**21454**	**19455**	15093	13899	10742
2^{13}	2^{19}	**34004**	**31658**	24267	15791	11651	**34165**	**27876**	20677	15173	11620
2^{12}	2^{20}	**54465**	**49328**	39861	15264	11051	**53846**	**42673**	30569	11189	9641
2^{11}	2^{21}	**86252**	**70661**	61333	13554	10099	**86182**	**60819**	44094	10884	8801

launch failed due to out of resources.[10] As before, the first two columns report the integer size in bits and the number of instances performed, and the columns denoted **CGBN** correspond to the performance of the CGBN library. The columns denoted **Cu-Q** and **Cu-F** correspond to our CUDA implementation for the classical (quadratic time) and FFT (log-linear time), respectively, and similarly, the columns denoted **Fu-Q** and **Fu-F** correspond to Futhark.

Our implementation of classical multiplication specializes *uint* to `uint64_t` and use a total sequentialization factor of 4, i.e., each thread computes two elements from the first half and their two symmetric opposites across the middle, as in figure 6. For FFT multiplication we use (i) the smallest sequentialization factor (greater than two) that allows the computation to fit in a CUDA block, which is constrained to 1024 threads, and (ii) the `FftPrime32` field, in which *uint* is `uint32_t` and *ubig* is `uint64_t`. Using `FftPrime64` is a bit slower mainly because it requires a modulo operation on 128-bit integers, which is very expensive. Key observations derived from table 2 are:

(1) On both **1-Mul** and **Poly**, our CUDA implementation of classical (quadratic) multiplication is faster than **CGBN** for integer sizes in the range $2^{15} \ldots 2^{18}$. Size 2^{15} is also very close to the split point from which on, our CUDA FFT implementation starts outperforming the quadratic implementation.
(2) **CGBN** is faster on the smaller integer sizes $2^{11} \ldots 2^{13}$ by factors as high as $1.2\times$ and $1.4\times$ on **1-Mul** and **Poly**, respectively, but our **Cu-Q** is faster on sizes $2^{15} \ldots 2^{17}$ by factors as high as $4.5\times$ and $21\times$.
(3) Our CUDA FFT implementation is faster than the best quadratic running implementation (i) by factors of $6.3\times$ on integer size 2^{18}, and (ii) by $2.6\times$ and $3.0\times$ factors on integer size 2^{17} on **1-Mul** and **Poly**, respectively.

[10] We prevented Futhark from switching to the slower versions described in section 5.1.

(4) **CGBN** demonstrates excellent (super-linear) scalability on integer sizes between $2^{11} \ldots 2^{15}$, since its performance on **Poly** is very-close to or better than the one on **1-Mul**, even when the two additions are not counted.
(5) Our CUDA FFT implementation also demonstrates excellent scalability on sizes higher than or equal to 2^{15}, which is all that matters because 2^{15} seems to be the split point from which point on FFT gains the upper hand.
(6) We attribute the performance gap between our CUDA and Futhark quadratic implementations on **1-Mul** to the fact that Futhark lacks support for 128-bit integers, and hence it uses less-efficient 64-bit arithmetic that computes the high and low parts. Similar to addition, Futhark's scalability (for **Poly**) is worse due to logical arrays being always mapped to shared memory.
(7) Finally, the Futhark FFT implementation runs out of resource for sizes of 2^{16} to 2^{18} bits, due to the last issue reported in section 5.2.

7 Conclusions

We have shown that level languages (`C++` and Futhark) can be used to implement big integer addition and multiplication concisely and efficiently for GPU computation. These implementations are simple and efficient for big integers of practical size, comparing favourably to the **CGBN** library for integers of size from 2^{15} to 2^{18} bits (i.e. up to about 79,000 digits). We have seen that an FFT-based multiplication can, by factors as high as $5\times$, outperform an efficient implementation of the classical multiplication on sizes that fit in a **CUDA** block. This is achieved using a naive implementation of finite field arithmetic — further improvement would be expected using Montgomery representation. The paper has presented the implementation in sufficient detail to be reproduced as desired by others. For `C++` template CUDA code has been provided.

We have measured the performance of these implementations against the high quality **CGBN** library, testing up to sizes of 2^{18} bits. For addition, our CUDA code outperforms the **CGBN** library by a factor of about $2\times$ to $4\times$ for integers of more than about 2^{14} bits. For most tests, our functional Futhark code also outperforms the **CGBN** library. For classical quadratic multiplication, our simple CUDA code is comparable to the **CGBN** library for numbers with upto 2^{14} bits and superior to **CGBN** for larger sizes. The Futhark implementation is comparable over most of the range of sizes. The CUDA FFT implementation of multiplication is superior for sizes greater than 2^{15} bits, becoming about 160 times faster at 2^{18} bits. For tests involving a combination of operations (the "Poly" tests), our CUDA implementation using classical arithmetic performs significantly better than **CGBN** for sizes above 2^{15} bits and within a factor of 2 below that size. While the Futhark implementation meets the criterion of being concise and flexible, furhter compiler support is required to approach the efficiency of our CUDA code. The present investigation has identified specific areas of Futhark compiler enhancement that together may lead to performance comparable to our CUDA code.

Acknowledgments. The authors were originally inspired to consider non-uniform memory architectures in a collaboration with Alan Mycroft [25], without whom we never would have ended up here!

Disclosure of Interests. The authors have no conflicts of interest to declare.

References

1. Bantikyan, H.: Big integer multiplication with cuda fft(cufft) library. International Journal of Innovative Research in Computer and Communication Engineering **2**, 6317–6325 (2014), https://api.semanticscholar.org/CorpusID:14759606
2. Blelloch, G.E.: Scans as Primitive Parallel Operations. Computers, IEEE Transactions **38**(11), 1526–1538 (1989)
3. Blelloch, G.E.: Vector models for data-parallel computing, vol. 75. MIT press Cambridge (1990)
4. Blelloch, G.E.: Programming Parallel Algorithms. Communications of the ACM (CACM) **39**(3), 85–97 (1996)
5. Blelloch, G.E., Hardwick, J.C., Sipelstein, J., Zagha, M., Chatterjee, S.: Implementation of a Portable Nested Data-Parallel Language. Journal of parallel and distributed computing **21**(1), 4–14 (1994)
6. Chen, L., Covanov, S., Mohajerani, D., Moreno Maza, M.: Big prime field FFT on the GPU. In: Proc. 2017 International Symposium on Symbolic and Algebraic Computation (ISSAC 2017). pp. 85–92. ACM Press (2017)
7. Cooley, J.W., Tukey, J.W.: An algorithm for the machine calculation of complex fourier series. Mathematics of Computation **19**(90), 297–301 (1965), http://www.jstor.org/stable/2003354
8. Dieguez, A.P., Amor, M., Doallo, R., Nukada, A., Matsuoka, S.: Efficient high-precision integer multiplication on the gpu. The International Journal of High Performance Computing Applications **36**(3), 356–369 (2022). https://doi.org/10.1177/10943420221077964, https://doi.org/10.1177/10943420221077964
9. Emmart, N., Weems, C.: High precision integer addition, subtraction and multiplication with a graphics processing unit. Parallel Processing Letters **20**, 293–306 (12 2010). https://doi.org/10.1142/S0129626410000259
10. Henriksen, T., Serup, N.G.W., Elsman, M., Henglein, F., Oancea, C.E.: Futhark: Purely functional GPU-programming with nested parallelism and in-place array updates. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 556–571. PLDI 2017, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3062341.3062354, http://doi.acm.org/10.1145/3062341.3062354
11. Henriksen, T., Thorøe, F., Elsman, M., Oancea, C.: Incremental flattening for nested data parallelism. In: Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming. pp. 53–67. PPoPP '19, ACM, New York, NY, USA (2019). https://doi.org/10.1145/3293883.3295707, http://doi.acm.org/10.1145/3293883.3295707
12. Isupov, K.: Using floating-point intervals for non-modular computations in residue number system. IEEE Access **8**, 58603–58619 (2020)
13. Joldes, M., Muller, J., Popescu, V., Tucker, W.: CAMPARY: Cuda multiple precision arithmetic library and applications. In: Greuel, G., Koch, T., Paule, P., Sommese, A. (eds.) Mathematical Software – ICMS 2016. LNCS 9725. pp. 232–240. Springer Cham (2016)

14. Joldes, M., Muller, J., Popescu, V., Tucker, W.: CAMPARY library (2017), https://homepages.laas.fr/mmjoldes/campary/
15. van Loan, C.: Computational Frameworks for the Fast Fourier Transform. SIAM (1992)
16. Lu, B., Mellor-Crummey, J.: Compiler optimization of implicit reductions for distributed memory multiprocessors. In: Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing. pp. 42–51 (1998). https://doi.org/10.1109/IPPS.1998.669887
17. Lu, M., He, B., Luo, Q.: Supporting extended precision on grahics processors. In: Proc. Sixth International Workshop on Data Management on New Hardware (DaMoN '10). pp. 19–26. ACM (2010)
18. Merrill, D., Garland, M.: Single-pass Parallel Prefix Scan with Decoupled Lookback. Technical report nvr-2016-002, NVIDIA Corporation (March 2016), https://research.nvidia.com/sites/default/files/pubs/2016-03_Single-pass-Parallel-Prefix/nvr-2016-002.pdf
19. Munksgaard, P.: Static and Dynamic Analyses for Efficient GPU Execution. Ph.D. thesis, Department of Computer Science, Faculty of Science, University of Copenhagen (2023), https://di.ku.dk/english/research/phd/phd-theses/2023/Philip_Munksgaard_Thesis.pdf
20. Munksgaard, P., Breddam, S.L., Henriksen, T., Gieseke, F.C., Oancea, C.: Dataset sensitive autotuning of multi-versioned code based on monotonic properties. In: Zsók, V., Hughes, J. (eds.) Trends in Functional Programming. pp. 3–23. Springer International Publishing, Cham (2021)
21. Munksgaard, P., Henriksen, T., Sadayappan, P., Oancea, C.: 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). https://doi.org/10.1109/SC41404.2022.00036
22. Nakayama, T., Takahashi, D.: Implementation of multiple-precision floating-point arithmetic for GPU computing. In: Proc 23rd IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2011). pp. 343–349. IASTED (2011)
23. Nakayama, T.: CUMP library (2017), https://github.com/skystar0227/CUMP
24. NVlabs: Cooperative Groups Big Numbers (CGBN) Library (2018), https://github.com/NVlabs/CGBN
25. Oancea, C.E., Mycroft, A., Watt, S.M.: A new approach to parallelising tracing algorithms. In: Proc. 2009 International Symposium on Memory Management (ISMM 2009). pp. 10–19. ACM Press (2009)
26. Oancea, C.E., Rauchwerger, L.: Logical inference techniques for loop parallelization. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 509–520. PLDI '12, ACM, New York, NY, USA (2012). https://doi.org/10.1145/2254064.2254124, http://doi.acm.org/10.1145/2254064.2254124
27. Oancea, C.E., Rauchwerger, L.: A Hybrid Approach to Proving Memory Reference Monotonicity. In: Rajopadhye, S., Mills Strout, M. (eds.) Languages and Compilers for Parallel Computing. pp. 61–75. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
28. Oancea, C.E., Rauchwerger, L.: Scalable conditional induction variables (civ) analysis. In: Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization. pp. 213–224. CGO '15, IEEE Computer Society, Washington, DC, USA (2015), http://dl.acm.org/citation.cfm?id=2738600.2738627

29. Strassen, V., Schönhage, A.: Schnelle multiplikation großer zahlen. Computing **7**(3-4), 281–292 (1971)
30. Topalovic, A., Restelli-Nielsen, W., Olesen, K.: Multiple-precision Integer Arithmetic. Final project of the “Data Parallel Programming" MSc-level course, Department of Computer Science, University of Copenhagen (2022), https://futhark-lang.org/student-projects/dpp21-mpint.pdf