Addition and Multiplication of Medium Size Integers in Cuda and Futhark

Cosmin E. Oancea and Stephen M. Watt
cosmin.oancea@diku.dk smwatt@uwaterloo.ca

Department of Computer Science
University of Copenhagen

School of Computer Science
University of Waterloo

1st of December 2023, MycroftFest Presentation

High-Level Motivation

- we decided that we need to have a new project dedicated specially to honor Alan's outstanding career

- something simple and beautiful combining math, languages, compilers and parallel computing

- ***arithmetic on big integers fits the bill***:
  - focus is on medium-size integers ($\leq$ 4096 32-bit words)
  - to what extent can it be supported efficiently in a high-level hardware-independent parallel language such as Futhark?
  - (we aim to use big integer inside already parallel programs, not only to perform one computation on huge integers.)

- Please excuse:
  - the very non-specific abstract
  - the very in-progress status of the work
    (we did what we could in between way too much teaching)

Related Work: Nvidia’s CGBN Library

https://github.com/NVlabs/CGBN

- aimed at medium-size integers ($\leq$ 4096 32-bit words)
- **promises to be fast (and it is!)**
  - operations are performed within a warp of (32) threads
  - data is maintained in registers (for most parts)
  - aggressively utilizes specialized-hardware instructions
- limitations:
  - open source but still hard to read
  - big-integer size needs to be a statically-known multiple of 32
  - does not scale to larger integers
  - sub-optimal composibility: suitable for extending simple Cuda kernels (that perform scalar operations)
  - some common operations are not supported, e.g., summing up big integers
  - obviously not usable with AMD GPUs.

A Large Body of Related Work

Among the ones aimed specifically at medium-size integers [1,3,5], the need for fast “technology” pollutes the algorithm, e.g.,

- the use of cuFFT imposes restrictions to the base (e.g., 10, 11) and the size of integer,
- the $O(n^2)$ implementation is based on tiling convolutions,
- mysterious operation for handling carries that, e.g., “adapts hardware adders in VLSI design into an equivalent GPU software operation”.

Most approaches are implemented in Cuda, but are not feasible to extend an already parallel program with support for big numbers.

[1] N Emmart and C Weems, “High precision integer addition, subtraction and multiplication with a graphics processing unit”, Parallel Processing Letters, 2010
[2] M. Moreno Maza and W. Pan, “Fast polynomial arithmetic on a GPU.”, J. of Physics, 2010.
[3] H. Bantikyan, “Big Integer Multiplication with CUDA FFT (cuFFT) Library”, 2014
[4] L. Chen, S. Covanov, D. Mohajerani, M. Moreno Maza, “Big Prime Field FFT on the GPU”, ISSAC, 2017.
[5] A.P. Dieguez, M. Amor, R. Doallo, A. Nukada, S. Matsuoka, “Efficient high-precision integer multiplication on the GPU”, Int. Journal of High Performance Computing Applications, 2022

Challenges

Context: extending a functional (high-level) parallel language to efficiently support big integers on GPUs.

Challenges:

1. nested parallelism (supported),
2. efficient utilization of (fast) memory (somewhat supported),
3. overcoming the (Cuda) technology wall, e.g., specialized hardware instructions on registers, cuFFT, convolution-like computations, the mysterious carry operation
    - **perhaps these are too big a hammer for such a small nail (?)**

Potential benefits:

- now one can actually write in a simple way parallel programs using big integers, that can run on various GPUs, e.g., AMD
- can be used together with other tools, e.g., AD.

Incremental Flattening: One Hat Does Not Fit All Sizes

How much of the application parallelism should be mapped to the hardware and how much should be efficiently sequentialized?

The key idea is to adapt the compilation process to the hardware and dataset characteristics by generating a sequence of semantically-equivalent but differently optimized code versions that exploit more and more levels of application parallelism.

Top-down analysis: each time a new map is encountered we

- V1: generate a version that utilizes all parallelism encountered so far and sequentializes everything inside the current map nest;
- V2: generate a code version in which the current parallelism is mapped on the grid, and the inner parallelism is flattened and mapped to workgroup level (in fast memory);
- V3: recursively continue flattening by loop-like distribution & interchange

The driver for multi-versioning is the degree of utilized parallelism.

[1] T. Henriksen, F. Thorøe, M. Elsman, C. Oancea.“Incremental Flattening for Nested Data Parallelism”, PPoPP, 2019.

Futhark Code Example (Contrived)

```
let f [m][n] (matrix₀: [m][n]i32) : [m][n]i32 =
    map2 (\(row₀: [n]i32) (i: i32) : [n]i32 ->
            loop (row : [n]i32) = (row₀)
              for _k = 0..63 do
                let row' = scan (+) 0 row
                let row''= map (* i) row'
                in  row''
         )
         matrix₀ (0..<m)
```

- **map** $f\ [a_0, \ldots, a_{n-1}] = [f\ a_0, \ldots, f\ a_{n-1}]$
- **scan** $\odot\ e_\odot\ [a_0, \ldots, a_{n-1}] = [a_0,\ a_0 \odot a_1, \ldots,\ a_0 \odot \ldots \odot a_{n-1}]$

Futhark Code Example (Contrived)

```
let f [m][n] (matrix_0: [m][n]i32) : [m][n]i32 =
    map2 (\(row_0: [n]i32) (i: i32) : [n]i32 ->
            loop (row : [n]i32) = (row_0)
              for _k = 0..63 do
                let row' = scan (+) 0 row
                let row''= map (* i) row'
                in  row''
         )
         matrix_0 (0..<m)
```

- **map** $f\ [a_0, \ldots, a_{n-1}] \;=\; [f\ a_0, \ldots, f\ a_{n-1}]$
- **scan** $\odot\ e_\odot\ [a_0, \ldots, a_{n-1}] \;=\; [a_0,\ a_0 \odot a_1, \ldots,\ a_0 \odot \ldots \odot a_{n-1}]$

Contrived, but it illustrates a valid computational pattern,
e.g., batch radix sort and batch matrix inversion by Gauss Jordan.

- `matrix: [m][n]i32`: an $m \times n$ matrix of 32-bit integers;
- sized types: the result has the same shape as the input (`[m][n]i32`);
- **map2**: each parallel iteration uses a `row` of the matrix and its index `i`;
- loops use an SSA-like notation;
- nesting: **map2** – **loop** – **map** o **scan**.

Code Version 1 (V1)

One can argue that if `n` is small (e.g., 4), V1 could be "optimal".

```
let f [m][n] (matrix₀: [m][n]i32) : [m][n]i32 =
     map2 (\(row₀: [n]i32) (i: i32) : [n]i32 ->
               loop (row : [n]i32) = (row₀)
                 for _k = 0..63 do
                   -- let row' = scan (+) 0 row
                   -- let row''= map (* i) row'
                   let row' = scratch n i32
                   in  loop (row') for i = 0..n-1 do
                         let row'[i] = ...
                         in  row'
          )
          matrix₀ (0..<m)
```

Exploit only the outer parallelism of degree `m`
(sequentialize the inner map-scan computation):

+ eliminates inter-thread communication;
- `m` might not be big enough to fully utilize hardware.

Code Version 2 (V2)

One can reasonably argue that if n is a reasonable (multiple of a) block size, and m not large enough, V2 could be "optimal".

```
let f [m][n] (matrix₀: [m][n]i32) : [m][n]i32 =
  map2 (\row₀ i : [n]i32 -> -- grid level
            loop (row: [n]i32) = (row₀)
              for _k = 0..63 do
                let row' = scan (+) 0 row -- workgroup
                let row''= map (* i) row' -- shared mem
                in  row''
        ) matrix₀ (0..<m)
```

Map the outer parallelism of degree m at the grid level
and the inner parallelism of degree n at the workgroup level:

+ exploits temporal locality (in shared memory);
+ exploits more/all parallelism;
- bad for very small values of n;
- might run out of shared memory for large n.

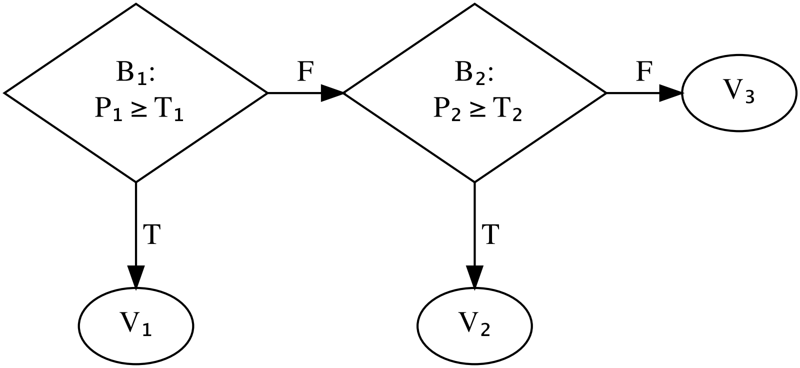
Code Version 3 (V3)

Flattening step performs map-loop interchange, then map fission:

```
let f [m][n] (matrix₀: [m][n]i32) : [m][n]i32 =
    loop (matrix: [m][n]i32) = (matrix₀)
      for _k = 0..63 do
        let matrix' = map (\ row -> scan (+) 0 row) matrix
        let matrix''= map2(\row' i-> map (*i) row') matrix' (0..<m)
        in  matrix''
```

One can reasonably argue that if m **is small and** n **is too big for a block size, then V3 is "optimal".**

Assembling Code Versions into One Program



P_i: the degree of utilized parallelism of code version *V_i*;

T_i: thresholds determined by a one-time autotuning process [2], one new threshold per code version;

[2] P. Munksgaard, S. Breddam, T. Henriksen, F. Gieseke, C. Oancea, “Dataset Sensitive Autotuning of Multi-Versioned Code based on Monotonic Properties”, TFP’21.

Memory Optimizations in Futhark

Supporting a correct-by-construction parallel representation typically introduces memory and copying overheads.

$$\begin{aligned}&\mathbf{let}\ X\ =\ \mathbf{map2}\ f\ \ A[R^{vert}]\ \ A[R^{horiz}]\\&\mathbf{let}\ A[W]\ =\ X\end{aligned}$$

In Futhark they are alleviated by switching to an “unsafe” memory-based IR together with

- a register-allocation like technique on memory buffers,
- an array short-circuiting analysis [3] that performs dependency-analysis on arrays to determine if it can safely allocate X directly in the memory space of A

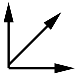
[3] P. Munksgaard, T. Henriksen, P. Sadayappan, C. Oancea, “Memory Optimizations in an Array Language”, SC, 2022.

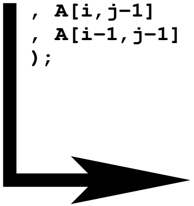
Illustration on Needleman-Wunsch (NW)

$$\texttt{let}\ X = \texttt{map2}\ f\ \ A[R^{vert}]\ \ A[R^{horiz}]$$
$$\texttt{let}\ A[W] \ = \ X$$

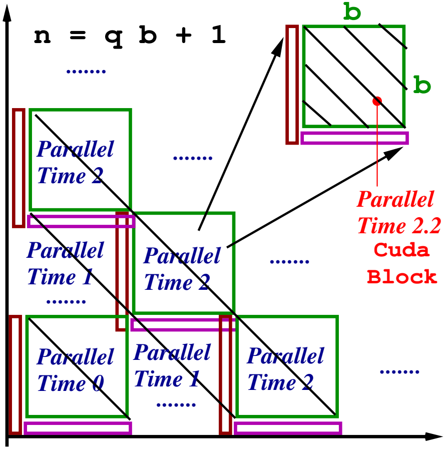
```
// Golden Sequential
// Implementation of NW
for (i=1; i<n; i++)
 for (j=1; j<n; j++)
   A[i,j] = f( A[i-1,j]
             , A[i,j-1]
             , A[i-1,j-1]
             );
```

Dependence Pattern:





Parallelized by block tiling and loop skweing



Demystifying Carry Propagation

ov	whether the pairwise addition had suffer overflow
mx	whether the pairwise addition yielded the maximal integer

```
def carry_prop (ov1 : bool, mx1: bool)
               (ov2 : bool, mx2: bool)
             : (bool, bool) =
  ( (ov1 && mx2) || ov2,  mx1 && mx2 )
```

Associative operator for propagating carries:

- we have overflow if the latter addition has overflow or if the previous addition has overflow and the latter addition resulted in the maximal integer
- the neutral element is `(false, true)`
- [4] A. Topalovic, W. Restelli-Nielsen, K. Olesen, “Multiple-precision Integer Arithmetic”, project for the Data-Parallel Programming course, 2022.

Can be implemented more efficiently as:

```
def carry_prop (c1: u32) (c2: u32) =
  (c1 & c2 & 2) | (( (c1 & (c2 >> 1)) | c2) & 1)
-- neutral element is 2
```

Ideal Big-Integer Addition (badd)

```
def imap2 as bs f = map2 as bs f

def badd [n] (as : [n]u32) (bs : [n]u32) : [n]u32 =
  let (pres, cs) =
    imap2 as bs
      (\ a b -> let s = a + b
                let b = u32.bool (s < a)
                let b = b | ((u32.bool (s == u32.highest)) << 1)
                in  (s, b)
      ) |> unzip

  let carries = scan carry_prop 2 cs  -- propagating the carries

  in  imap2 (0..<n) pres
        (\ i r -> r + u32.bool (i > 0 && ( carries[i-1] & 1 == 1)) )

entry oneAdd [m][n] (ass: [m][n]u32) (bss: [m][n]u32) : [m][n]u32 =
  imap2 ass bss badd
```

Unfortunately this implementation is sub-optimal, because it does not "efficiently sequentializes the parallelism in excess".

- in-progress work to support this in the compiler
- until then we can achieve this by hand

Big-Integer Addition with Efficient Sequentialization

- each "thread" processes 4 consecutive elements
- each block/workgroup computes `ipb` big additions

```
let badd [ipb][n] (as : [ipb*(4*n)]u64) (bs : [ipb*(4*n)]u64) : [ipb*(
  let g = ipb * n
  let cp2sh (i : i32) =
        let g = i32.i64 g in
        ( ( as[i], as[g + i], as[2*g + i], as[3*g + i] )
        , ( bs[i], bs[g + i], bs[2*g + i], bs[3*g + i] ) )
  -- copy from global to shared memory in coalesced fashion
  let ( ass, bss ) = (0..<g) |> map i32.i64 |> map cp2sh  |> unzip
  let (a1s, a2s, a3s, a4s) = unzip4 ass -- by shortcircuiting
  let (b1s, b2s, b3s, b4s) = unzip4 bss -- a1s ... a4s and
  let ash = a1s ++ a2s ++ a3s ++ a4s    -- b1s ... b4s are
  let bsh = b1s ++ b2s ++ b3s ++ b4s    -- allocated in ash and bsh
                                        -- (i.e., no overhead here)
  in  (badd0 ipb n ash bsh) :> [ipb*(4*n)]u64
```

Big-Integer Addition with Efficient Sequentialization

- each ”thread” processes 4 consecutive elements
- each block/workgroup computes `ipb` big additions

```
let badd [ipb][n] (as : [ipb*(4*n)]u64) (bs : [ipb*(4*n)]u64) : [ipb*(
  let g = ipb * n
  let cp2sh (i : i32) =
        let g = i32.i64 g in
        ( ( as[i], as[g + i], as[2*g + i], as[3*g + i] )
        , ( bs[i], bs[g + i], bs[2*g + i], bs[3*g + i] ) )
 -- copy from global to shared memory in coalesced fashion
  let ( ass, bss ) = (0..<g) |> map i32.i64 |> map cp2sh  |> unzip
  let (a1s, a2s, a3s, a4s) = unzip4 ass -- by shortcircuiting
  let (b1s, b2s, b3s, b4s) = unzip4 bss -- a1s ... a4s and
  let ash = a1s ++ a2s ++ a3s ++ a4s    -- b1s ... b4s are
  let bsh = b1s ++ b2s ++ b3s ++ b4s    -- allocated in ash and bsh
                                        -- (i.e., no overhead here)
  in  (badd0 ipb n ash bsh) :> [ipb*(4*n)]u64
```

- `ash` and `bsh` are allocated in shared memory
- the many concatenations are noops because shortcircuit analysis allocates `a1s .. a4s` and `b1s ..  b4s` directly in the memory space of `ass` and `bss`.
- the rest is ”efficiently” sequentialized in a similar fashion in `badd0`
- we perform some redundant computation to save buffer space
- scan is translated to a segmented scan to support several instances per block.

Performance of Big Additions; Higher is Better)

- the total size in bytes of a batch of big-numbers is kept constant 536870912
- we vary *n*: the number of bits in the big word.
- six additions computes $6 * a + 10 * b$
- performance is reported in GB/sec, but for six additions we multiply by 6 the number of bytes.
- Our-Cuda is the Cuda prototype that we developed.
- **Hardware: Nvidia A100, with peak bandwidth 1555 GB/sec**

GB/sec	Performance of One Addition			Performance of Six Additions		
#Bits	CGBN	Our-Cuda	Futhark	CGBN	Our-Cuda	Futhark
2^{16}	376	1275	1233	2199	2265	2033
2^{15}	317	1318	1228	1871	2387	2039
2^{14}	556	1312	1222	3104	2396	2019
2^{13}	1136	1277	1217	6655	2397	2023
2^{12}	1307	1315	1215	6843	2450	2021
2^{11}	1171	1324	1219	6446	2402	2019

CGBN performs poorly on one addition, but scales almost perfectly on six additions.

Hints that the default storage place for arrays should be registers instead of shared memory!

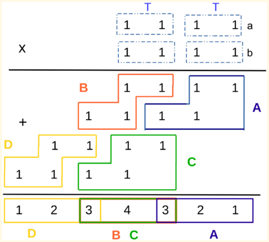
$O(n^2)$ Big Multiplication using Convolutions [5]

$$C_k = \Sigma_{i+j=k}\ low(a_i \cdot b_j)\ +\ \Sigma_{i+j=k-1}\ high(a_i \cdot b_j)$$

Vectorization Example: Remember Vector Machines?

```
for (k from 0 to 2*N-1)
    c[k] = 0

for (i from 0 to N-1)
  for (j from 0 to N-1)
    c[i+j] += a[i] * b[j]
```



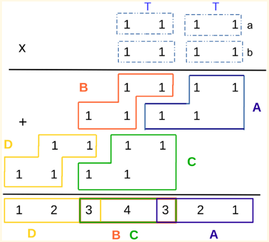
$O(n^2)$ Big Multiplication using Convolutions [5]

$$C_k = \Sigma_{i+j=k}\ low(a_i \cdot b_j)\ +\ \Sigma_{i+j=k-1}\ high(a_i \cdot b_j)$$

Vectorization Example: Remember Vector Machines?

```
for (k from 0 to 2*N-1)
    c[k] = 0

for (i from 0 to N-1)
  for (j from 0 to N-1)
    c[i+j] += a[i] * b[j]
```



- apply aggressively block and register tiling, but ...
- either do twice the work or introduce if statement
- merging tiles require atomic operations (high overhead).

[5] A.P. Dieguez, M. Amor, R. Doallo, A. Nukada, S. Matsuoka, “Efficient high-precision integer multiplication on the GPU”, Int. Journal of High Performance Computing Applications, 2022

$O(n^2)$ Big Multiplication the Easy Way

$$C_k = \Sigma_{i+j=k}\ low(a_i \cdot b_j) \ + \ \Sigma_{i+j=k-1}\ high(a_i \cdot b_j)$$

Thread 0 computes $k = 0$ and $k = n - 1$:

$(i,j)_{i+j=0}$ $(0,0)$

$(i,j)_{i+j=n-1}$ $(0,n-1),\ (n-1,0),\ (1,n-2)\ (n-2,1),\ldots$

Thread 1 computes $k = 1$ and $k = n - 2$:

$(i,j)_{i+j=1}$ $(0,1),\ (1,0)$

$(i,j)_{i+j=n-2}$ $(0,n-2),\ (n-2,0),\ (1,n-3)\ (n-3,1),\ldots$

...

- actually we compute 4 elements per thread
- thread divergence can be optimized
- simple, no tiling, no atomic operations, ...

$O(n^2)$ Big Multiplication the Easy Way

```
let convolution4 (n: i32) (ash: []u64) (bsh: []u64) (tid: i32)
               : ( (u64,u64,u64,u64), (u64,u64,u64,u64)) =

  let (instance, vtid) = ( tid / n, tid % n )
  let off = instance * (4*n)

  let k1 = 2*vtid
  let (lhc0, lhc1) = -- first half (e.g., first and second)
    loop (lhc0, lhc1) = ((0u64,0u64,0u32), (u64,u64,u32))
    for kk < k1 do
        let i = kk
        let j = k1 - i
        let lhc0 = computeIter64 ash[off+i] bsh[off+j] lhc0
        let lhc1 = computeIter64 ash[off+i] bsh[off+j+1] lhc1
        in  (lhc0, lhc1)
  let lhc1 = computeIter64 ash[off+k1+1] bsh[off+0] lhc1
  let (l0, l1, h2, c3) = combine2 lhc0 lhc1

  let k2 = 4*n - k1 - 2 -- second half (e.g., ante-last and last)
  let (lhc2, lhc3) = ...
  let (l_nm2, l_nm1, h_n, c_np1) = combine2 lhc2 lhc3
  in ( (l0, l1, h2, c3), (l_nm2, l_nm1, h_n, c_np1) )
```

$O(n^2)$ Big Multiplication the Easy Way

```
let computeIter64 (ai: u64) (bj: u64) (l: u64, h: u64, c: u32)
                                              : (u64, u64, u32) =
    let ck_l = ai * bj
    let n_l = l + ck_l
    let c_l = (u32.u64 (n_l >> 32)) < (u32.u64 (ck_l >> 32))
          |> u64.bool
    let n_h = h + c_l
    let ck_h = u64.mulhigh ai bj
    let n_h = n_h + ck_h
    let c_h = (u32.u64 (n_h >> 32)) < (u32.u64 (h >> 32))
          |> u32.bool
    let n_c = c + c_h
    in  (n_l, n_h, n_c)
```

FFT-based Big Multiplication

Stephen did some number-theory magic and found some nice prime numbers of form $p = k * 2^n + 1$ that allow 2^n roots of unity:

- $p = 3221225473$ allows working in base $2^1$5
- $p = 4179340454199820289$ alows working in base $2^3$1

Then implemented big multiplication based on Cooley-Tukey FFT, but without the final carry propagation step. In Futhark:

- the FFT version wins for the largest size (2048 words)
- albeit it suffers from some significant performance bugs
- hypothesize that with a Cuda implementation the FFT version will win handily.
- we do not report performance results for the FFT multiplication yet.

Performance of Big Multiplications; Higher is Better

- the total size of a batch is kept constant (536870912); we vary the number of bits
- four multiplications compute $(a^2 + b) * (b^2 + b) + a * b$
- performance is reported in Giga-u32-ops/sec:
  - for one: $q * 4 * m * m / (\text{runtime in us} * 1000)$
  - for four: $q * 4 * 4 * m * m / (\text{runtime in us} * 1000)$
  - $m = \frac{n}{32}$ the big-integer length in 32-bit words and q is the batch size
- Our-Cuda is the Cuda prototype that we developed.
- **Hardware: Nvidia A100**

Gops/sec	**Performance of One Multiplication**			**Performance of Four Multiplications**		
#Bits	**CGBN**	**Our-Cuda**	**Futhark**	**CGBN**	**Our-Cuda**	**Futhark**
2^{16}	16281	19706	14279	14063	18759	14322
2^{15}	15903	18290	13337	16767	17495	13369
2^{14}	16211	16234	11779	16099	15542	11374
2^{13}	14487	13122	10048	14523	12820	8813
2^{12}	13450	12045	9658	13206	11547	7593
2^{11}	12622	10220	8683	12860	9654	6496

- **Our Cuda prototype wins on the bigger sizes 2048 and 1024 words, although it does not use specialized hardware instructions or tiled convolutions.**
- Futhark is $< 1.4\times$ slower, partly because it does not support 128-bit integers.
- CGBN wins on smaller sizes.

Narrowing the Performance Gap & Conclusion

- support for 128-bit integers,
- ”efficient sequentialization” analysis for intra-group kernels,
- default storage for arrays created at intra-group (Cuda block) level should perhaps be registers instead of shared memory.
- various other improvements

Conclusion:

- promising results obtained in a relatively short time,
- How?
  By keeping things clean & using the right hammer for the job,
- Our Cuda performance competitive with the CGBN library,
- Futhark performance not too far behind — there is hope.

Thank you Alan!

Thank you Dominic, Tomas and Jeremy for organizing this.