On the CUDA implementation of Reduce and Scan

Cosmin Oancea and Troels Henriksen

September 2025

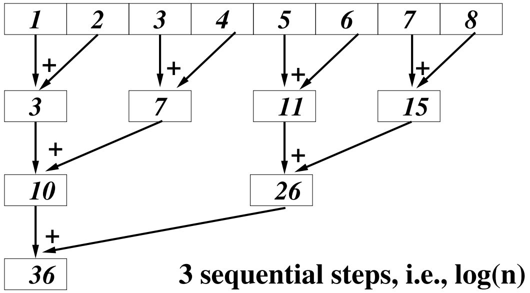
Course Contents

Summing an Integer Array

$$\sum_{i<n} x[i]$$

Binary Tree Reduction

The idea: each thread reads two neighbouring elements, adds them together, and writes one element. This halves the array in size. Continue until only a single element is left.



- Each level becomes a kernel invocation, with number of threads equal to half the number of array elements.
- $O(n)$ work and $O(\log(n))$ span (optimal).
- **Why is this not efficient?**

Improving the Tree Reduction

The idea: instead of shrinking the array by a factor of two for each level, shrink it by the CUDA-block size.

- Same asymptotic performance.
- Avoids kernels with very few threads. E.g with block size 256:
  $10000000 \rightarrow 39063 \rightarrow 153 \rightarrow 1$.

Improving the Tree Reduction

The idea: instead of shrinking the array by a factor of two for each level, shrink it by the CUDA-block size.

- Same asymptotic performance.
- Avoids kernels with very few threads. E.g with block size 256: $10000000 \rightarrow 39063 \rightarrow 153 \rightarrow 1$.

Implementation	$n = 1000$	$n = 1000000$
Tree reduction	$77\mu s$	$363\mu s$
Block reduction	$17\mu s$	$179\mu s$

Applying Brent's Lemma

The idea: instead of letting the thread count depend on the input size, always launch the same number of threads, and have each thread perform an efficient sequential summation of a *chunk* of the input.

- GPUs have a maximum (hardware/problem-dependent) capacity for exploiting parallelism. Beyond that limit, parallelism is at best worthless, and usually comes with overhead (e.g. excessive synchronisation).
- *A straightforward implementation of this idea only works if the operator is commutative, and also allows fusing a* `map` *producer!*

Applying Brent’s Lemma

The idea: instead of letting the thread count depend on the input size, always launch the same number of threads, and have each thread perform an efficient sequential summation of a *chunk* of the input.

- GPUs have a maximum (hardware/problem-dependent) capacity for exploiting parallelism. Beyond that limit, parallelism is at best worthless, and usually comes with overhead (e.g. excessive synchronisation).
- *A straightforward implementation of this idea only works if the operator is commutative, and also allows fusing a* `map` *producer!*

Implementation	$n = 1000$	$n = 1000000$
Tree reduction	$77\mu s$	$363\mu s$
Block reduction	$17\mu s$	$179\mu s$
Chunked reduction	$70\mu s$	$103\mu s$

Using Atomics

The idea: GPUs have special hardware support for performing certain memory updates atomically. In CUDA, this is exposed through *atomic operations*.

```
int atomicAdd( volatile __global int *p
             , int val)
```

- Concise parallel reduction: each thread reads an element and uses `atomicAdd()` to update the same location in memory.
- **Why is this slow for large inputs?**

Using Atomics

The idea: GPUs have special hardware support for performing certain memory updates atomically. In CUDA, this is exposed through *atomic operations*.

```
int atomicAdd( volatile __global int *p
             , int val)
```

- Concise parallel reduction: each thread reads an element and uses `atomicAdd()` to update the same location in memory.
- **Why is this slow for large inputs?**

Implementation	$n = 1000$	$n = 1000000$
Tree reduction	$77\mu s$	$363\mu s$
Block reduction	$17\mu s$	$179\mu s$
Chunked reduction	$70\mu s$	$103\mu s$
Atomics	$8\mu s$	$1278\mu s$

Coalesced Access to Global Memory

- On NVIDIA GPUs, the hardware threads are split into WARPS, where a WARP consists of 32 threads.
- The threads in a WARP execute in lockstep—in SIMD fashion—meaning at each given cycle they all execute the same instruction.

- Coalesced access to global memory is obtained when the threads in a WARP access in their common load/store instruction consecutive words in global memory.
- A coalesced load/store instruction is serviced by one (possibly two) memory transactions, hence fast.

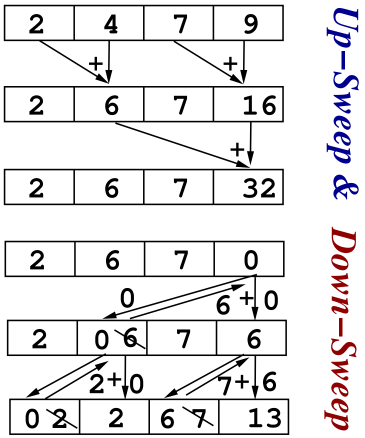
- Uncoalesced access may requires as many as 32 memory transactions, which are sequentially issued by the memory controller, hence may generate significant slowdown.
- Multi-threading somehow alleviates the “uncoalesced” overhead but only to some extent.

For Non-Commutative Operators (MSSP)

We denote the CUDA block size with B.

- We can also fuse the mapped function in the reduction itself.
- Have each thread reducing sequentially some CHUNK consecutive elements (statically known):
  - then the B per-thread results are reduced cooperatively by the threads in a block.
  - sequential (virtualization) loop on top so that the number of blocks is kept to under 1024, i.e., two-stage reduce.
- However, if done naively, this will lead to uncoalesced access. For coalesced, use shared memory as a staging buffer:
  - the threads in a block read consecutive elements from global memory and write them in shared memory;
  - then each thread processes sequentially its CHUNK consecutive elements from shared memory, which is not subject to the "coalescing" overhead.

Parallel Exclusive Scan with Associative Operator $\oplus$

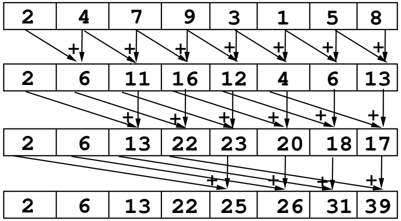


Two Steps:

- Up-Sweep: similar with reduction
- Root is replaced with neutral element.
- Down-Sweep:
  - the left child sends its value to parent and updates its value to that of parent.
  - the right-child value is given by $\oplus$ applied to the left-child value and the (old) value of parent.
  - note that the right child is in fact the parent, i.e., in-place algorithm.

Scan's Work and Depth: $D(n) = \Theta(lg\ n), W(n) = \Theta(n)$

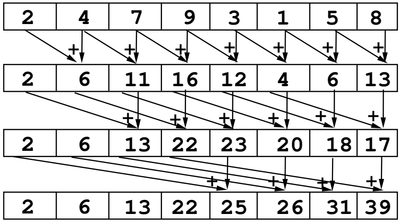
Warp-Level Inclusive Scan for GPUs



```
Input:  array A of $n=2^k$ elements
                         of type $\alpha$
        $\oplus : (\alpha,\alpha) \to \alpha$ associative
Output: B = [$a_1$, $a_1 \oplus a_2$, ... ,$\oplus_{j=0}^{n-1} a_j$]
1.   forall i = 0 : n-1 do
2.     B[i] $\leftarrow$ A[i]
3.   endfor
4.   for d = 0 to k-1 do
5.     h = $2^d$
6.     forall i = h to n-1 do
7.       B[i] $\leftarrow$ B[i-h] $\oplus$ B[i]
8.     endfor
9.   endfor
```

Offers better performance because it operates in one sweep rather than two!

Warp-Level Inclusive Scan Implementation

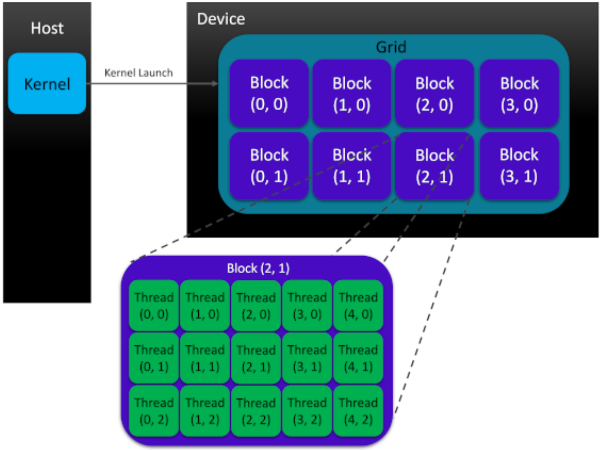


```
Input:  array A of n=$2^k$ elements
                         of type $\alpha$
        $\oplus$ : $(\alpha,\alpha) \rightarrow \alpha$ associative
Output: B = [$a_1$, $a_1 \oplus a_2$, ... ,$\oplus_{j=0}^{n-1}$ $a_j$]
1.  forall i = 0 : n-1 do
2.    B[i] $\leftarrow$ A[i]
3.  endfor
4.  for d = 0 to k-1 do
5.    h = $2^d$
6.    forall i = h to n-1 do
7.      B[i] $\leftarrow$ B[i-h] $\oplus$ B[i]
8.    endfor
9.  endfor
```

- Open file ”pbbKernels.cu.h”, and implement function named “scanIncWarp” (follow the instructions)
- Your `n = WARP` and `k = lgWARP`; Ignore the init loop;
- Unroll the `for d` loop (`#pragma unroll`);
- loop `forall i = h to n-1` is implicit (parallel threads),
- it should be replaced by a condition `if (i>=h) { ... }`,
- except that `i` in condition `if (i>=h)` **is not the thread id**.
- Remember, you want to `scan` each warp, independently!

Cuda: Grid-Block Structure of Threads

Credit: pictures taken from `http://education.molssi.org/gpu_programming_beginner/03-cuda-`

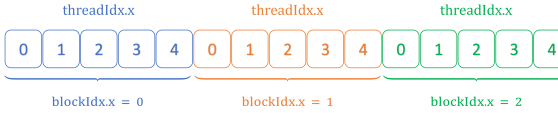


Blocks and Grids have at most three dimensions—denoted x, y, z, with x innermost and z outermost. Their sizes are specified at kernel launch. Inside the **kernel** you may use:

- `blockDim.x`: block size in dim x
- `blockIdx.x`: current block index (in x)
- `threadIdx.x`: local index of the current thread inside its block (in dim x)
- `gridDim.x` number of blocks on dim x
- Ditto for dimensions y and z.

The global thread index in dim q$\in \{x,y,z\}$:

`threadIdx.q`$+$**`blockIdx.q`**$\cdot$**`blockDim.q`**



CUDA Scan Implementation

BLACKBOARD!

- Generic CPU skeleton in `hostSkel.cu.h`.
  1. reduce each block and publish the per-block results in a buffer `buff` (number of blocks ≤ 1024)
  2. scan in place the buffer `buff` using one CUDA block.
  3. now you can implement the core scan kernel:
     - 3.1 each thread reads and scans its CHUNK of consecutive elements held in register memory; (The elements are copied in a coalesced way by using shared memory as a staging buffer.)
     - 3.2 the last element (reduction) of each thread's chunk (held in registers) is published in shared memory.
     - 3.3 then the threads in the block cooperatively scan the per thread reductions.
     - 3.4 then each thread `tid` updates the elements of its `chunk` array with the element at position `tid-1` from step 3.3, and with the previous-block element of `buff` from step 2.
     - 3.5 the CHUNK per-thread results are copied from register to shared to global memory (to ensure coalesced writes)
- If N is the length of the input, this requires $2N$ coalesced reads and N coalesced writes from/to global memory.