GPU Computing

Patterns for massively parallel programming (part 2)

Scan Pattern

Edwin Carlinet, Joseph Chazalon

`firstname.lastname@epita.fr`

Spring 2024

EPITA Research Laboratory (LRE)

Slides generated on March 26, 2024

Scan Pattern

What is a scan?

Scan computes all partial reductions of a collection:

$$B_k = A_0 \oplus \ldots \oplus A_k$$

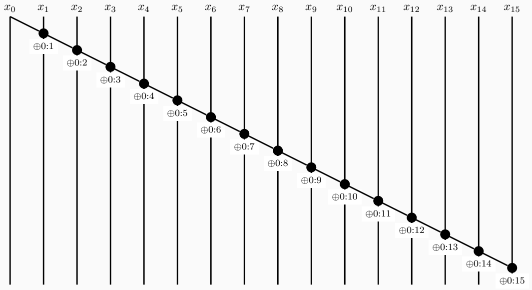
```
tmp = init;
for (i = 0; i < n; ++i)
  B[i] = (tmp += A[i])
```

In	1	5	3	4	2	1
Out	1	6	9	13	15	16

Usage:

- Integration (cumulated histogram)
- Resource allocation (memory to parallel threads, camping spots…)
- Base building block for many algorithms (sorts, strings comparisons…)

Performance baselines



Sequential version

The sequential (linear) version is *work efficient*:

- Number of operations: $N - 1$
- Number of steps: $N - 1$

Naive parallel version

Have every thread to add up all x elements needed for the y element

$$y0 = x0$$

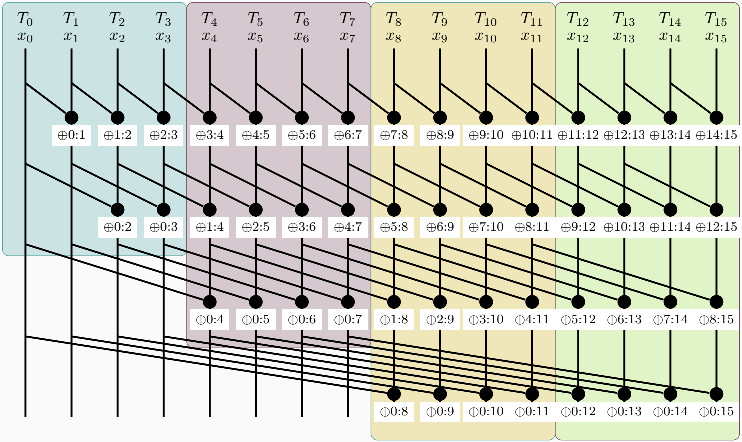
$$y1 = x0 + x1$$

$$y2 = x0 + x1 + x2$$

- Number of operations: $\frac{N*(N-1)}{2} \sim O(N^2)$
- Number of steps: $N - 1$

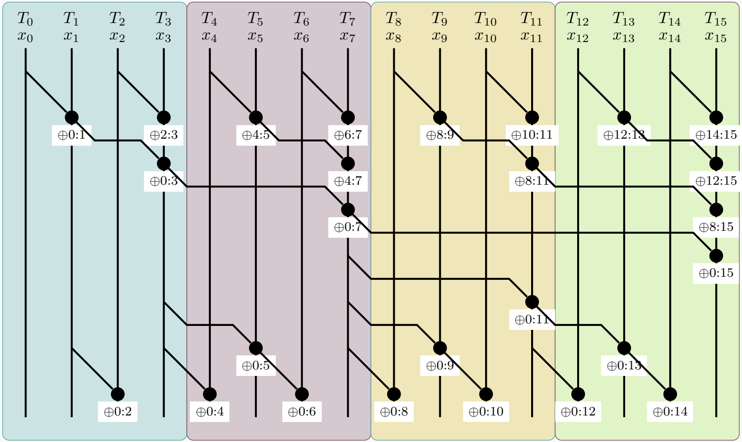
Parallel programming is easy as long as you do not care about performance.

Scan Pattern at the Warp or Block Level : Kogge-Stone

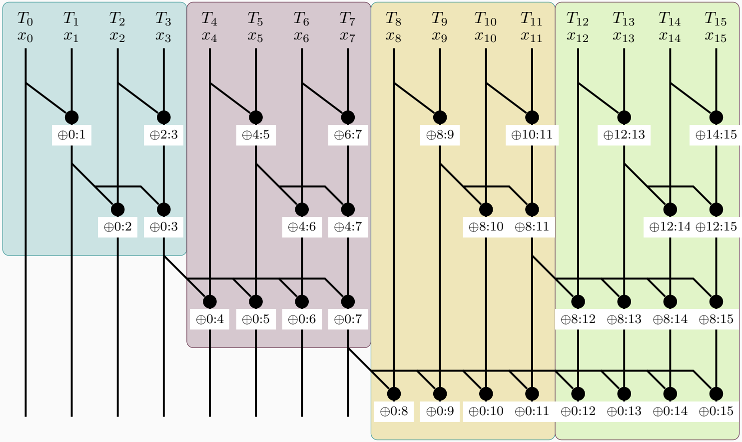


- Number of steps: $\log N$ ✓
- Ressource efficiency: ✓
- Work efficiency: $\sim N \log N$ ✗

Scan Pattern at the Warp or Block Level : Brent-Kung



- Number of steps: $2 \log N$
- Ressource efficiency: ✗(all warps remain active till the end)
- Work efficiency: $2N$ ✓



- Number of steps: $\log N$ ✓
- Ressource efficiency: ✓
- Work efficiency: $\frac{N}{2}\log N$ ✓

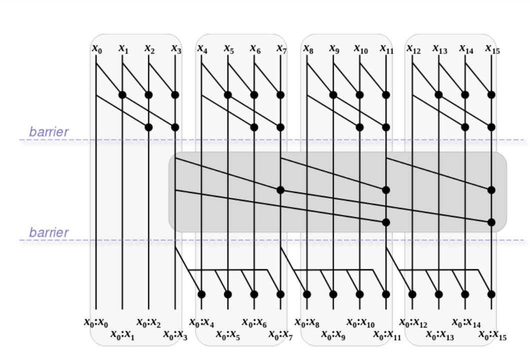
Scan Pattern at the Block or Grid Level

The patterns before can be applied:

- At the warp level (no sync until Volta)
- At the block level (thread sync)

At the global level: multi-level kernel application in global memory

- Scan then propagate
- Reduce then scan



At the *grid* level: 3 kernels

1. *Scan* per *block*.

   Store the sum in global memory

   `tmp[blockIdx.x] = local_sum`.
2. Perform a *scan* on `tmp`
3. Perform an *Add* on each *block* with offset

   `tmp[blockIdx.x - 1]`

At the *block* level: 1 kernel with sync.

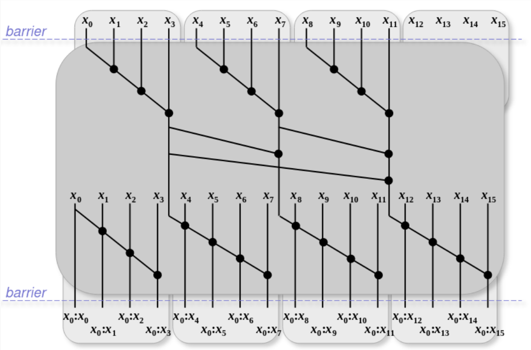
1. *Scan* per *warp*.

   Store the sum in *shared* memory

   `tmp[warpId] = local_sum`.
2. Perform a *scan* on `tmp` (using sync threads)
3. Perform an *Add* on each *warp* with offset

   `tmp[warpId - 1]`

Scan Pattern at the Block or Grid Level : Reduce then scan



At the *grid* level: 3 kernels

1. *Reduce* per *block*.

   Store the sum in *global* mem.

   `tmp[blockIdx.x] = local_sum`
2. Perform a *scan* on `tmp` (recursive call)
3. Perform a *scan* on each *block* with offset

   `tmp[blockIdx.x - 1]`

At the *block* level: 1 kernel with sync.

1. *Reduce* per *warp*.

   Store the sum in *shared* memory

   `tmp[warpId] = local_sum`.
2. Perform a *scan* on `tmp` (using sync threads)
3. Perform a *scan* on each *warp* with offset

   `tmp[warpId - 1]`

Scan summary

Lot more to say about the scan.

Not easy to implement properly at block level:

- a smart implementation would group active threads while minimizing memory accesses
- direct implementation of Kogge-Stone is fast ($log_2(N)$ steps) but requires many operations ($Nlog_2(N) - (N-1)$)
- direct implementation of Brent-Kung requires more steps ($2log_2(N)$) while requiring less operations ($2N$) in theory, but on NVidia architectures most of inactive threads (in active warps) continue to occupy resources

Even more complex at the grid level:

- it is possible to avoid separating the algorithm in three distinct phases, using some synchronization between blocks
- idea: as soon as reduction for block 0 and 1 are complete, propagation for block 1 is possible