GPU TECHNOLOGY CONFERENCE

CUB

"collective" software primitives

Duane Merrill

NVIDIA Research

What is CUB?

1. **A design model for “collective” primitives**
   - How to make reusable SIMT software constructs
2. **A library of collective primitives**
   - Block-reduce, block-sort, block-histogram, warp-scan, warp-reduce, etc.
3. **A library of global primitives**
   - Device-reduce, device-sort, device-scan, etc.
   - Constructed from collective primitives
   - Demonstrate performance, performance-portability

Software reuse

Software reuse

Abstraction & composability are fundamental

- **Reducing redundant programmer effort…**
  - Saves time, energy, money
  - Reduces buggy software
- **Encapsulating complexity…**
  - Empowers ordinary programmers
  - Insulates applications from underlying hardware

- Software reuse empowers a ***durable*** programming model

Software reuse

Abstraction & composability are fundamental

- **Reducing redundant programmer effort…**
  - Saves time, energy, money
  - Reduces buggy software
- **Encapsulating complexity…**
  - Empowers ordinary programmers
  - Insulates applications from underlying hardware

- Software reuse empowers a ***durable*** programming model

“Collective” primitives

Parallel programming is hard…

Cooperative parallel programming is hard…

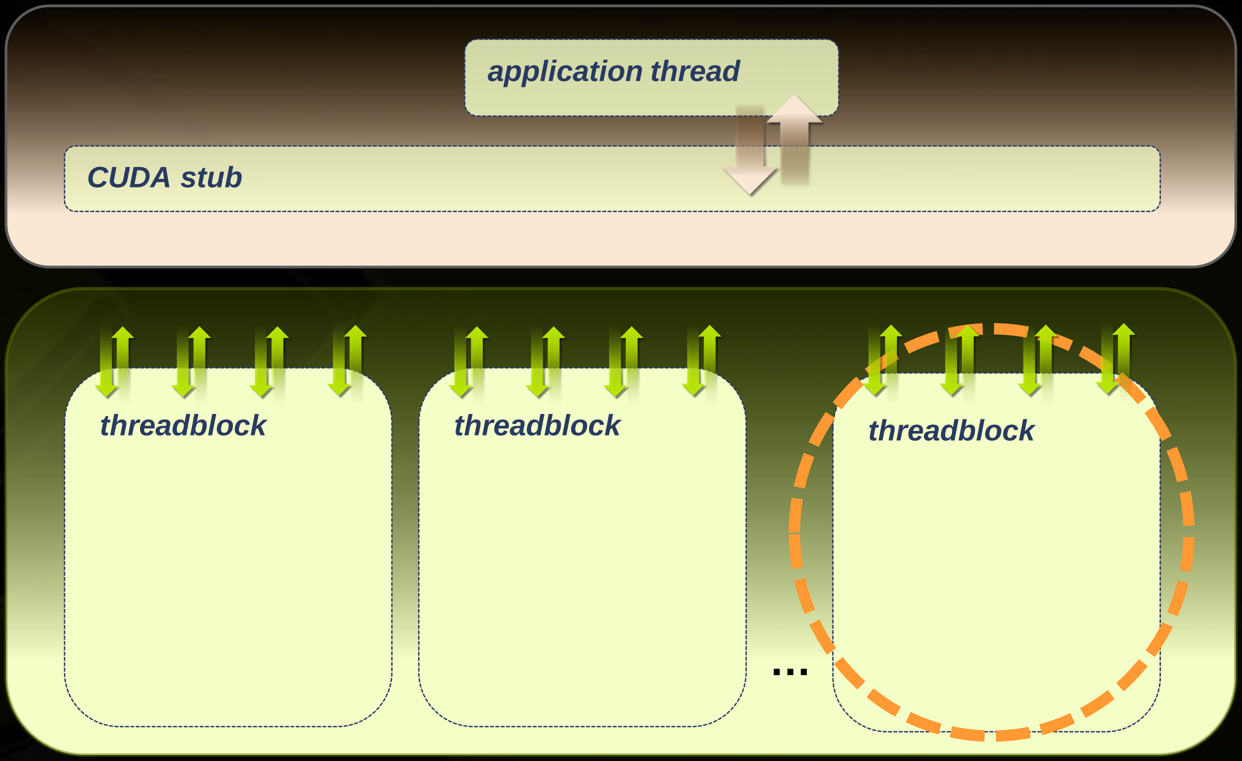
- Parallel decomposition and grain sizing
- Synchronization
- Deadlock, livelock, and data races
- Plurality of state
- Plurality of flow control (divergence, etc.)
- Bookkeeping control structures
- Memory access conflicts, coalescing, etc.
- Occupancy constraints from SMEM, RF, etc
- Algorithm selection and instruction scheduling
- Special hardware functionality, instructions, etc.

Parallel programming is hard…

- Parallel decomposition and grain sizing
- Synchronization
- Deadlock, livelock, and data races
- Plurality of state
- Plurality of flow control (divergence, etc.)
- Bookkeeping control structures
- Memory access conflicts, coalescing, etc.
- Occupancy constraints from SMEM, RF, etc
- Algorithm selection and instruction scheduling
- Special hardware functionality, instructions, etc.

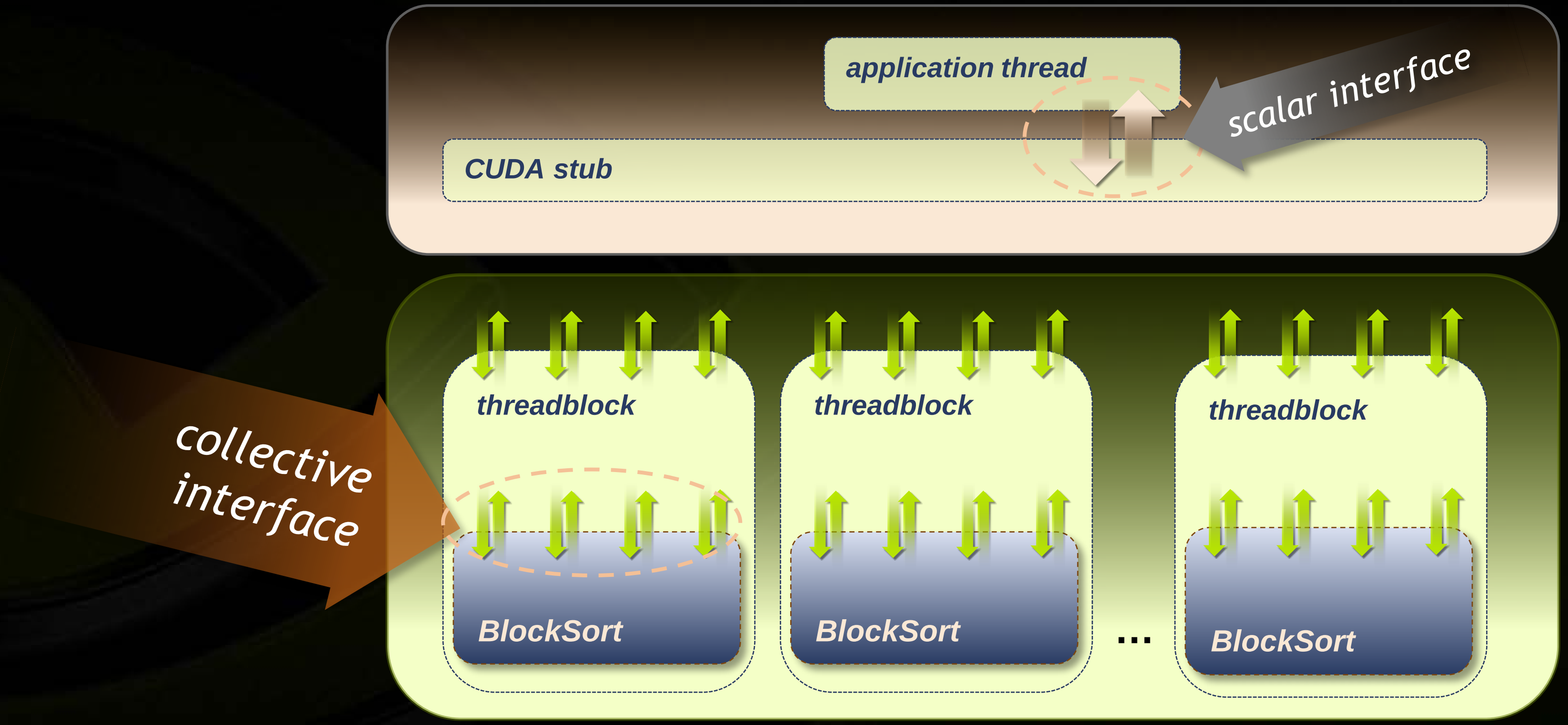
…

CUDA today

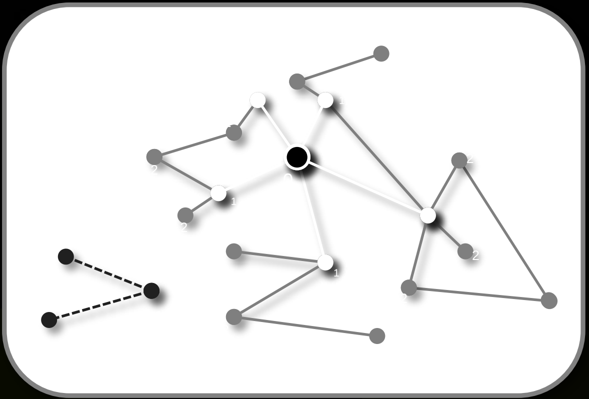

application thread
CUDA stub
threadblock
threadblock
...
threadblock

CUDA today

"Collective primitives" are the missing layer in today's CUDA software stack

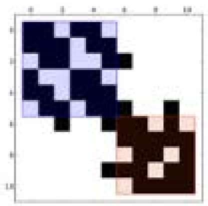


What do these have in common?



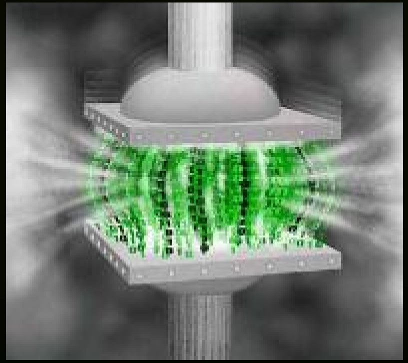
Parallel sparse graph traversal

Parallel radix sort



$$y \leftarrow Ax + y$$

Parallel SpMV

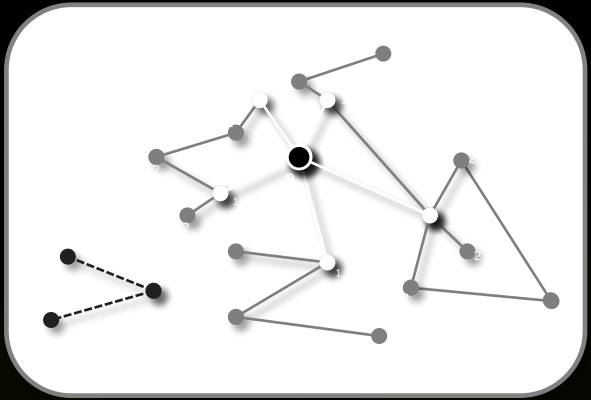


Parallel BWT compression

What do these have in common?

Block-wide prefix-scan

Queue management

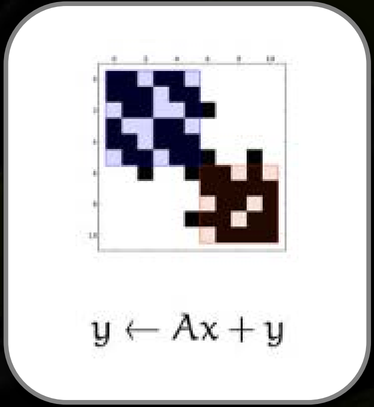


Parallel sparse graph traversal

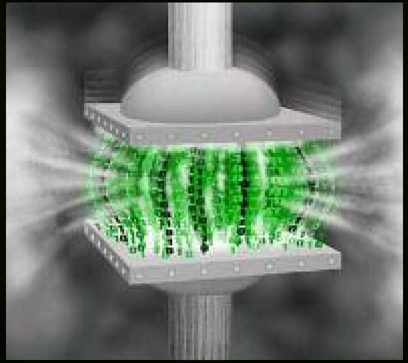
Parallel radix sort

Partitioning

Segmented reduction



Parallel SpMV

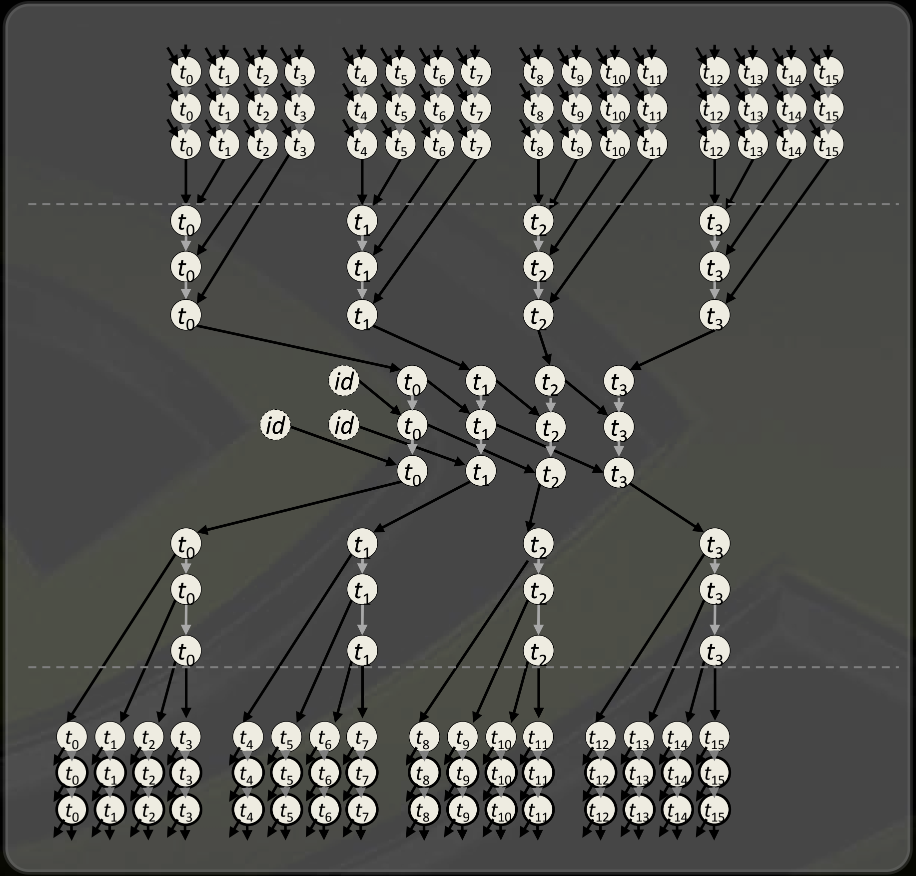


Parallel BWT compression

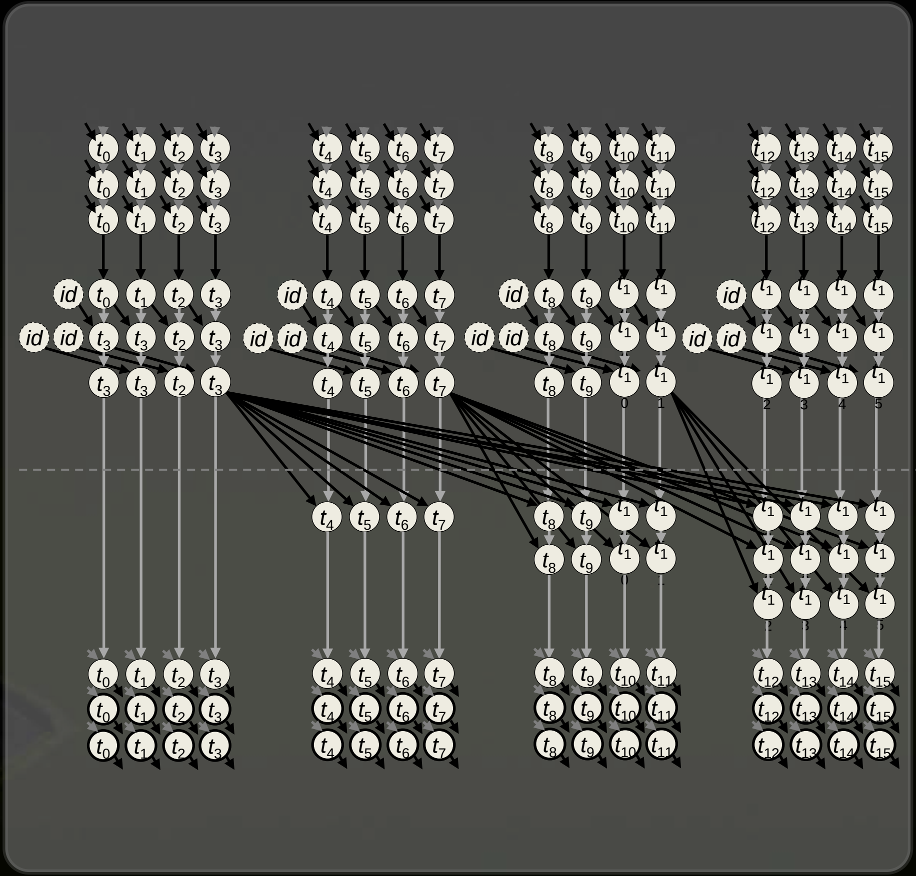
Recurrence solver

Examples of parallel scan data flow

16 threads contributing 4 items each



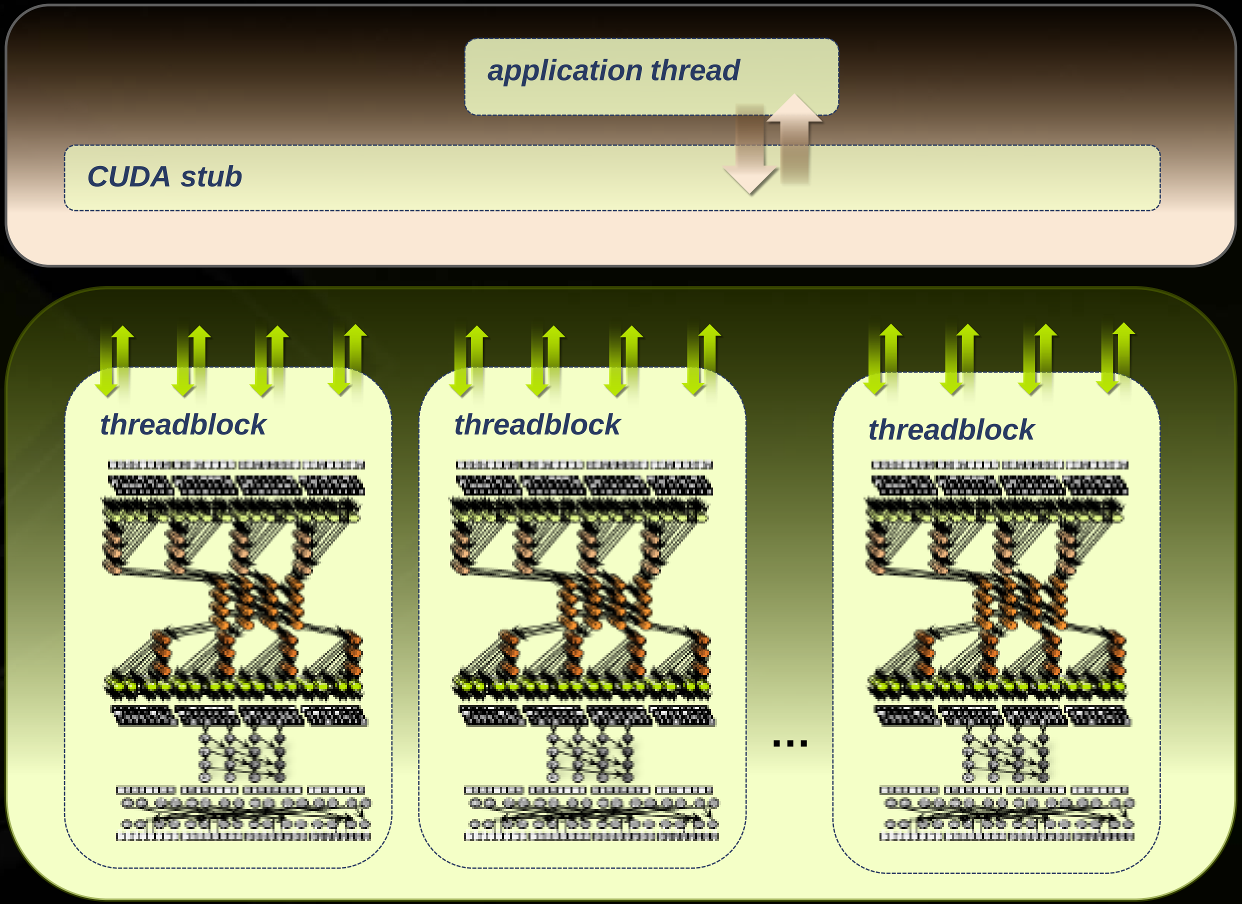
Work-efficient Brent-Kung hybrid
(~130 binary ops)



Depth-efficient Kogge-Stone hybrid
(~170 binary ops)

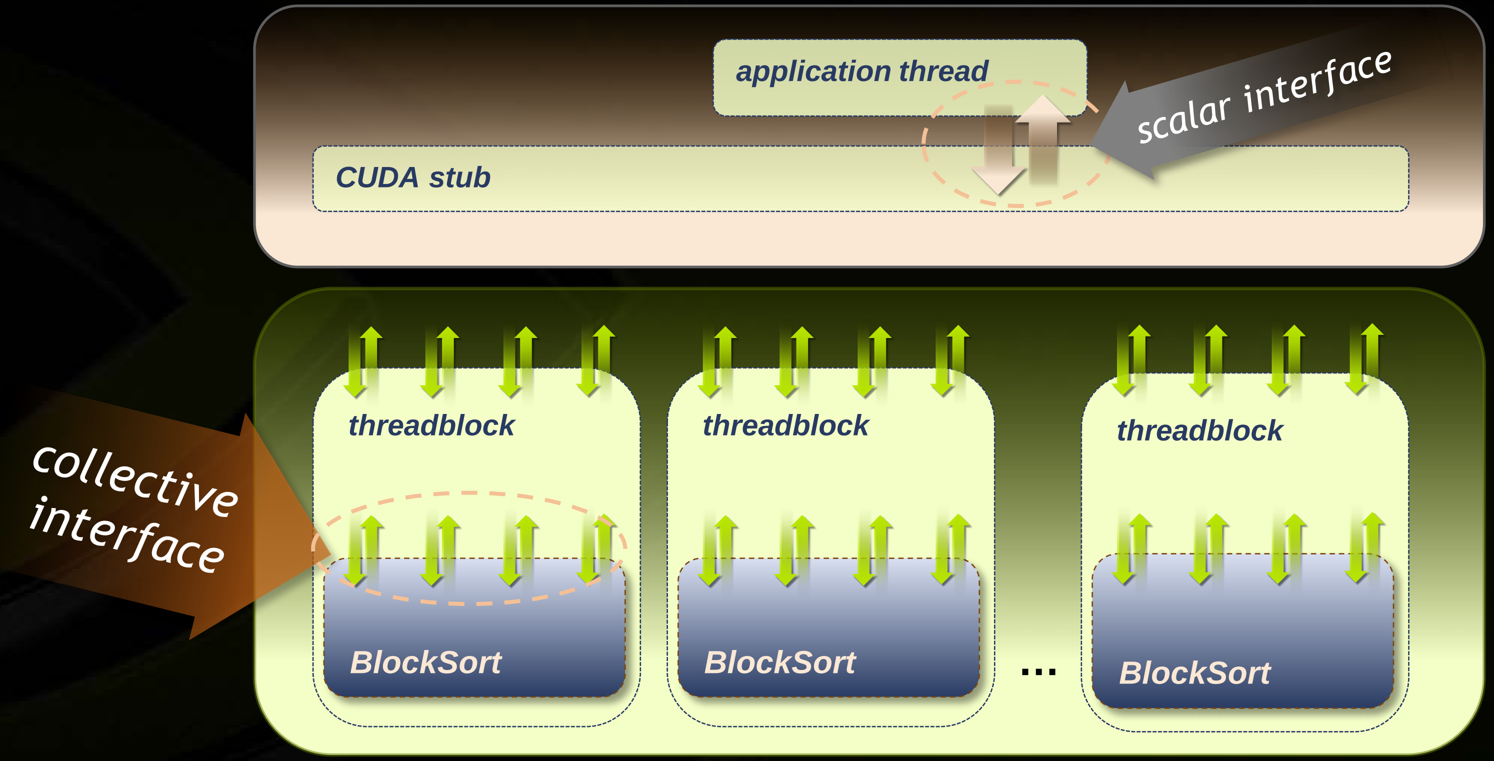
CUDA today

Kernel programming is complicating



CUDA today

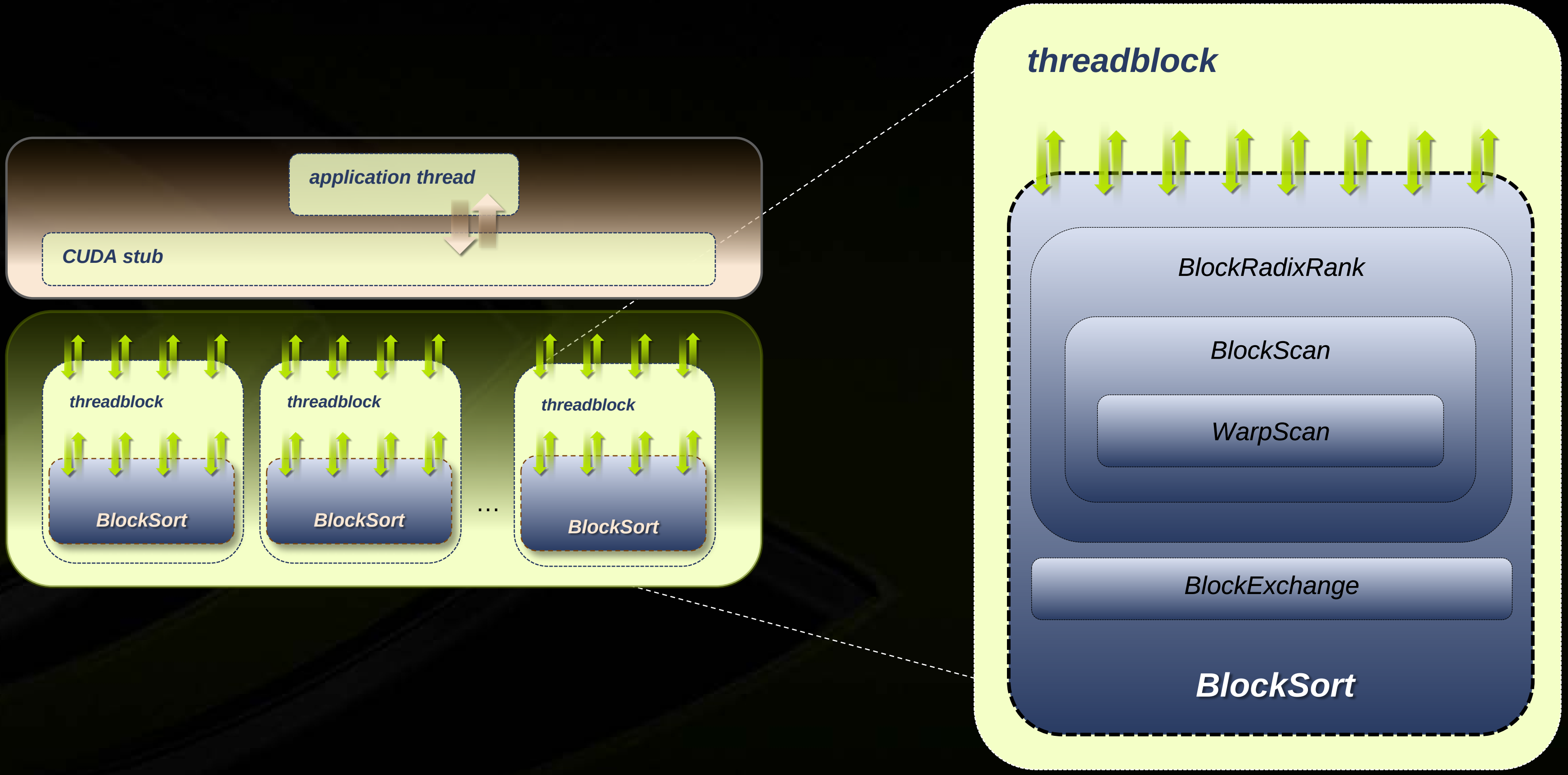
“Collective primitives” are the missing layer in today’s CUDA software stack



Collective design & usage

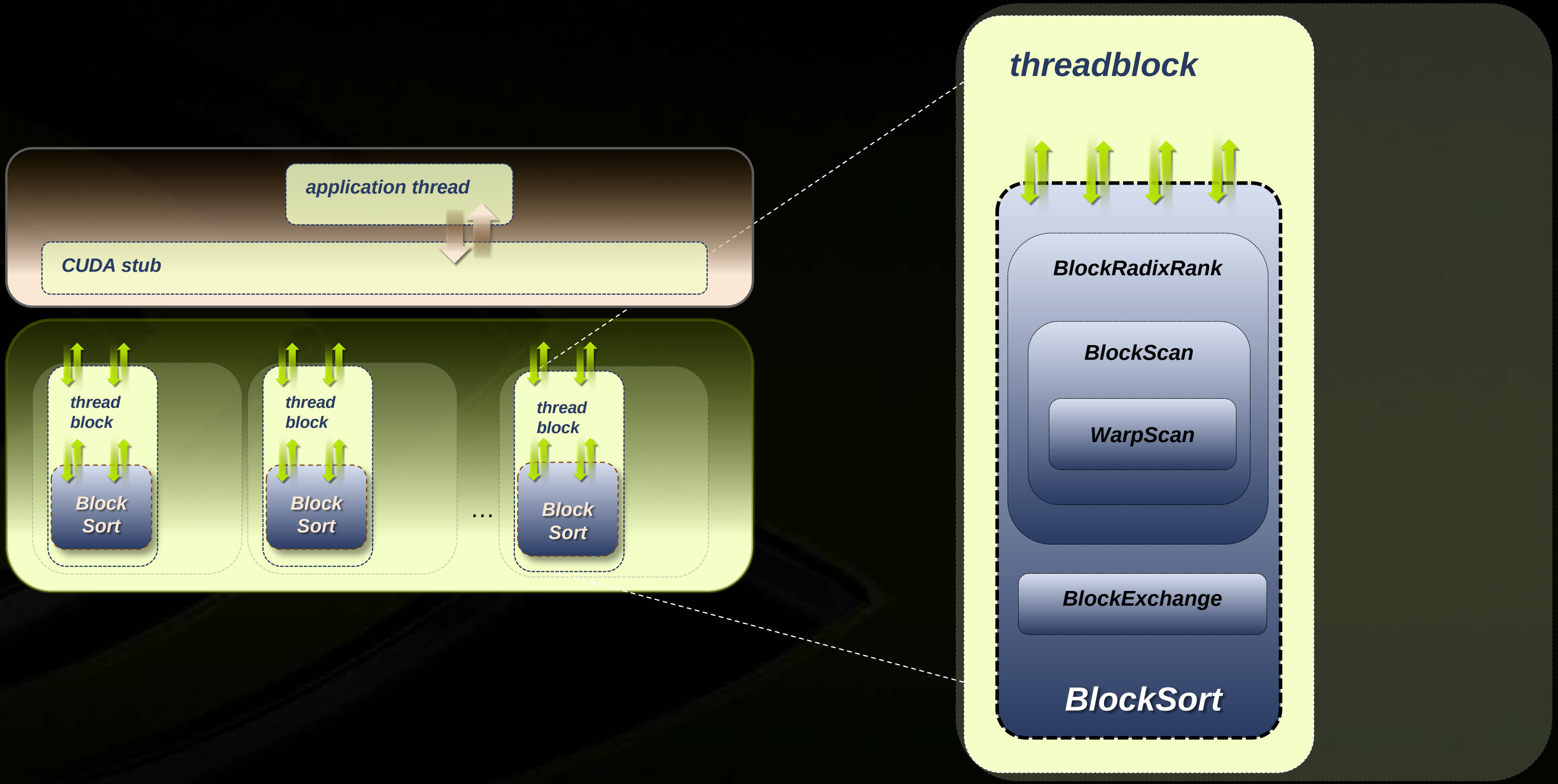
Collective design criteria

Components are easily nested & sequenced



Collective design criteria

Flexible interfaces that scale (& tune) to different block sizes, thread-granularities, etc.



Collective interface design

- 3 parameter fields separated by concerns
- Reflected shared resource types

```
__global__ void ExampleKernel()
{
    // Specialize cub::BlockScan for 128 threads
1   typedef cub::BlockScan<int, 128> BlockScanT;

    // Allocate temporary storage in shared memory
2   __shared__ typename BlockScanT::TempStorage scan_storage;

    // Obtain a 512 items blocked across 128 threads
    int items[4];
    ...

    // Compute block-wide prefix sum
3 4 BlockScanT(scan_storage).ExclusiveSum(items, items);
```

1. **Static specialization interface**
   - Params dictate storage layout and unrolling of algorithmic steps
   - Allows data placement in fast registers
2. **Reflected shared resource types**
   - Reflection enables compile-time allocation and tuning
3. **Collective construction interface**
   - Optional params concerning inter-thread communication
   - Orthogonal to function behavior
4. **Operational function interface**
   - Method-specific inputs/outputs

Collective primitive design

Simplified block-wide prefix sum

```
template <typename T, int BLOCK_THREADS>
class BlockScan
{
    // Type of shared memory needed by BlockScan
    typedef T TempStorage[BLOCK_THREADS];

    // Per-thread data (shared storage reference)
    TempStorage &temp_storage;

    // Constructor
    BlockScan (TempStorage &storage) : temp_storage(storage) {}

    // Prefix sum operation (each thread contributes its own data item)
    T Sum (T thread_data)
    {
        for (int i = 1; i < BLOCK_THREADS; i *= 2)
        {
            temp_storage[tid] = thread_data;
            __syncthreads();
            if (tid – i >= 0) thread_data += temp_storage[tid];
            __syncthreads();
        }

        return thread_data;
    }
};
```

Sequencing CUB primitives

Using `cub::BlockLoad` and `cub::BlockScan`

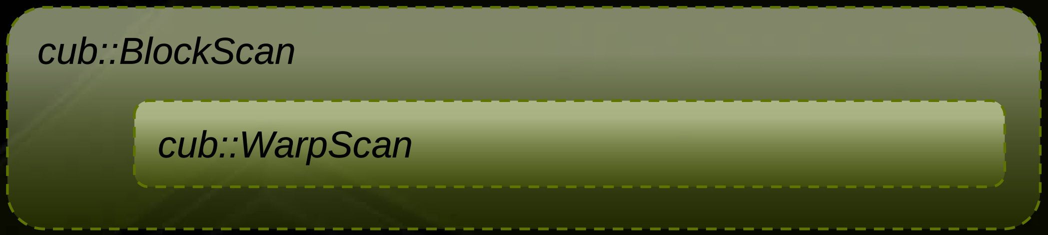
```
__global__ void ExampleKernel(int *d_in)
{
    // Specialize for 128 threads owning 4 integers each
    typedef cub::BlockLoad<int*, 128, 4>  BlockLoadT;
    typedef cub::BlockScan<int, 128>      BlockScanT;

    // Allocate temporary storage in shared memory
    __shared__ union {
        typename BlockLoadT::TempStorage load;
        typename BlockScanT::TempStorage scan;
    } temp_storage;

    // Use coalesced (thread-striped) loads and a subsequent local exchange to
    // block a global segment of 512 items across 128 threads
    int items[4];
    BlockLoadT(temp_storage.load).Load(d_in, items)

    __syncthreads()

    // Compute block-wide prefix sum
    BlockScanT(temp_storage.scan).ExclusiveSum(items, items);
    ...
```

Specialize, Allocate

Load, Scan

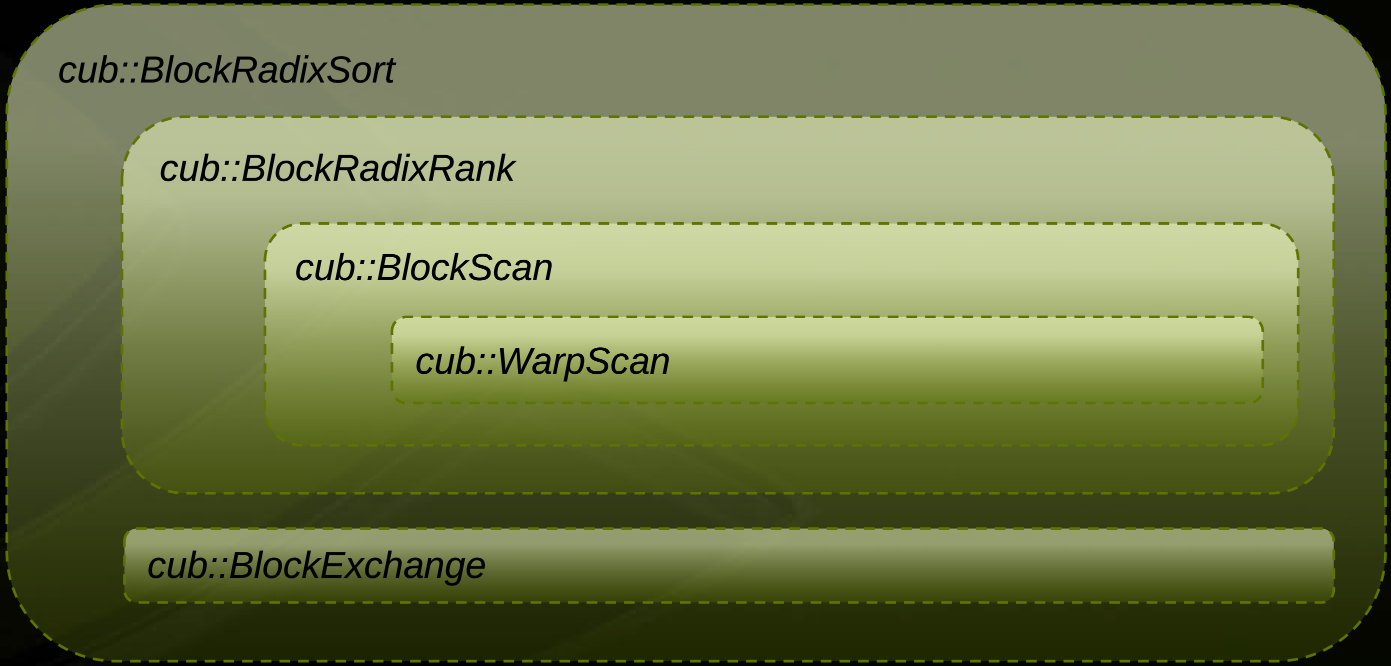
Nested composition of CUB primitives

`cub::BlockScan`



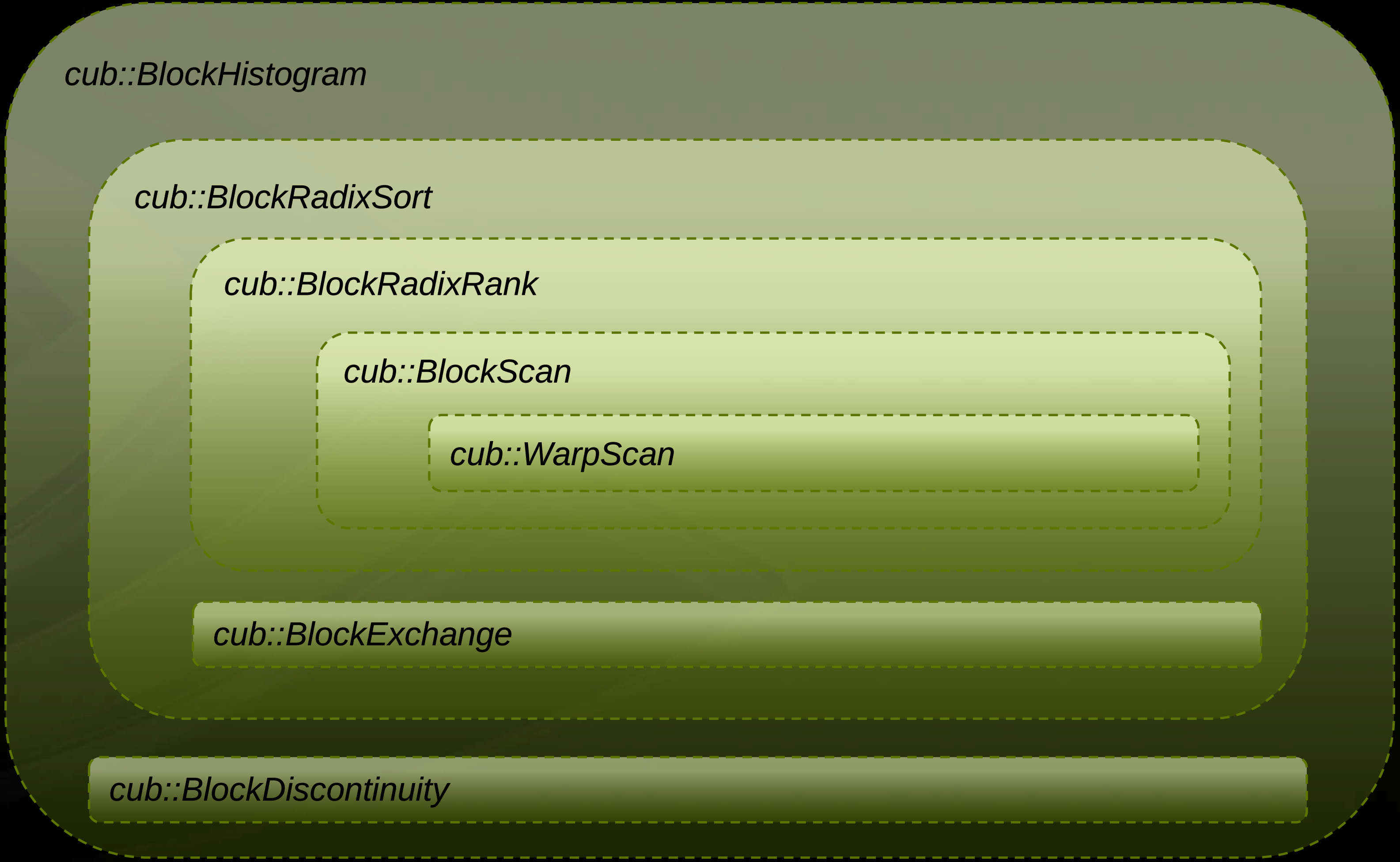
Nested composition of CUB primitives

`cub::BlockRadixSort`



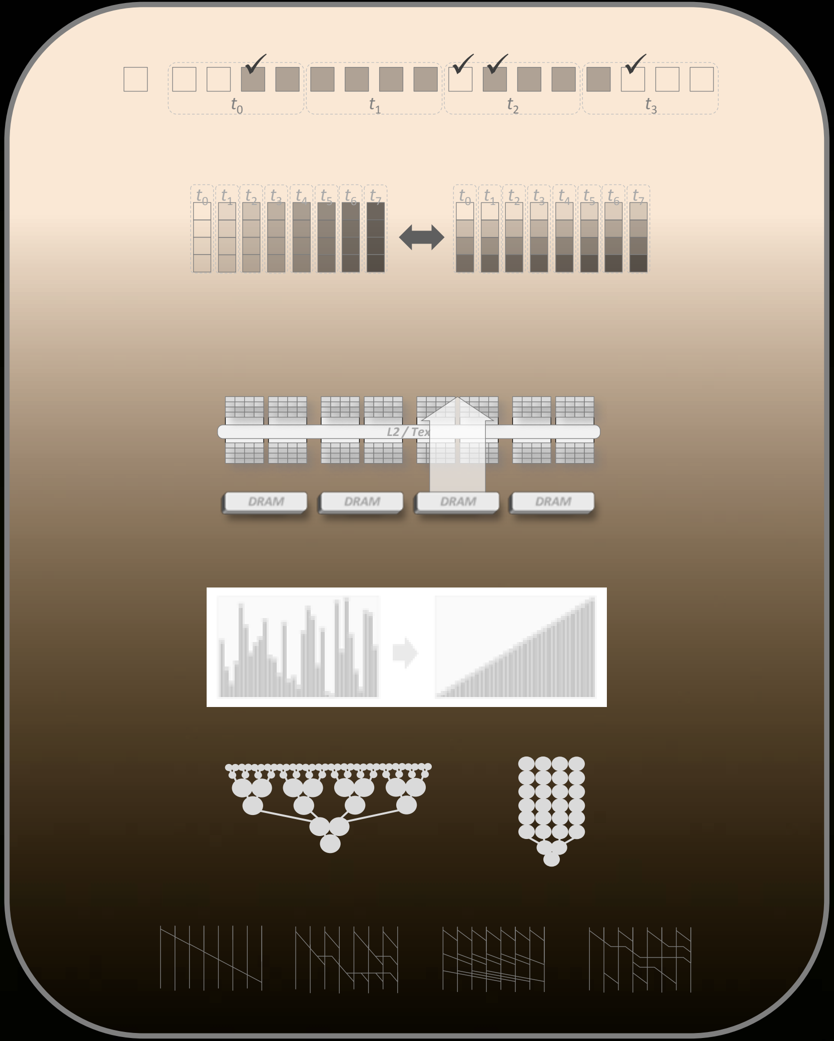
Nested composition of CUB primitives

`cub::BlockHistogram` (specialized for `BLOCK_HISTO_SORT` algorithm)



Block-wide and warp-wide CUB primitives

- `cub::BlockDiscontinuity`
- `cub::BlockExchange`
- `cub::BlockLoad & cub::BlockStore`
- `cub::BlockRadixSort`
- `cub::WarpReduce & cub::BlockReduce`
- `cub::WarpScan & cub::BlockScan`
- `cub::BlockHistogram`

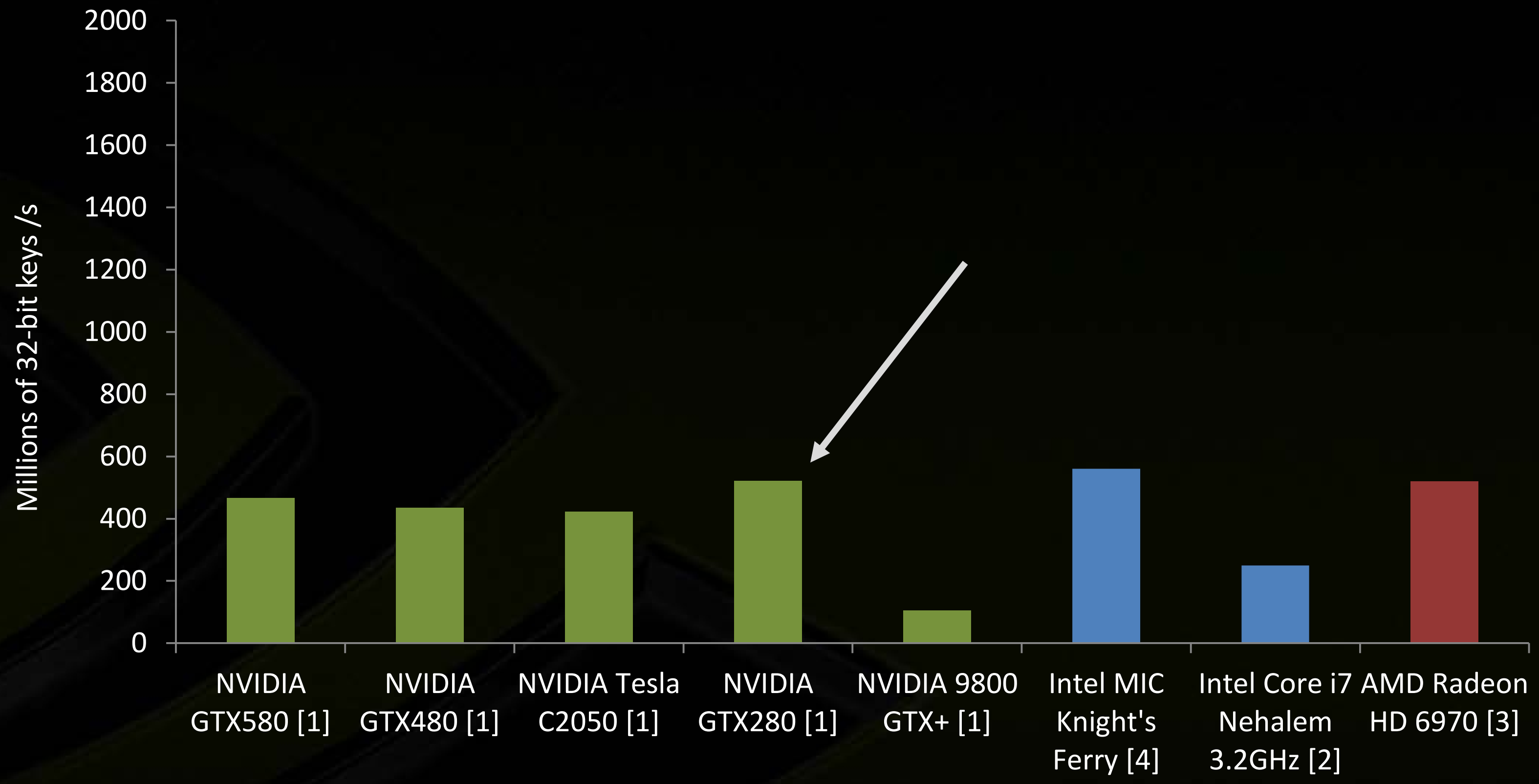


… and more at the **CUB** project on GitHub

http://nvlabs.github.com/cub

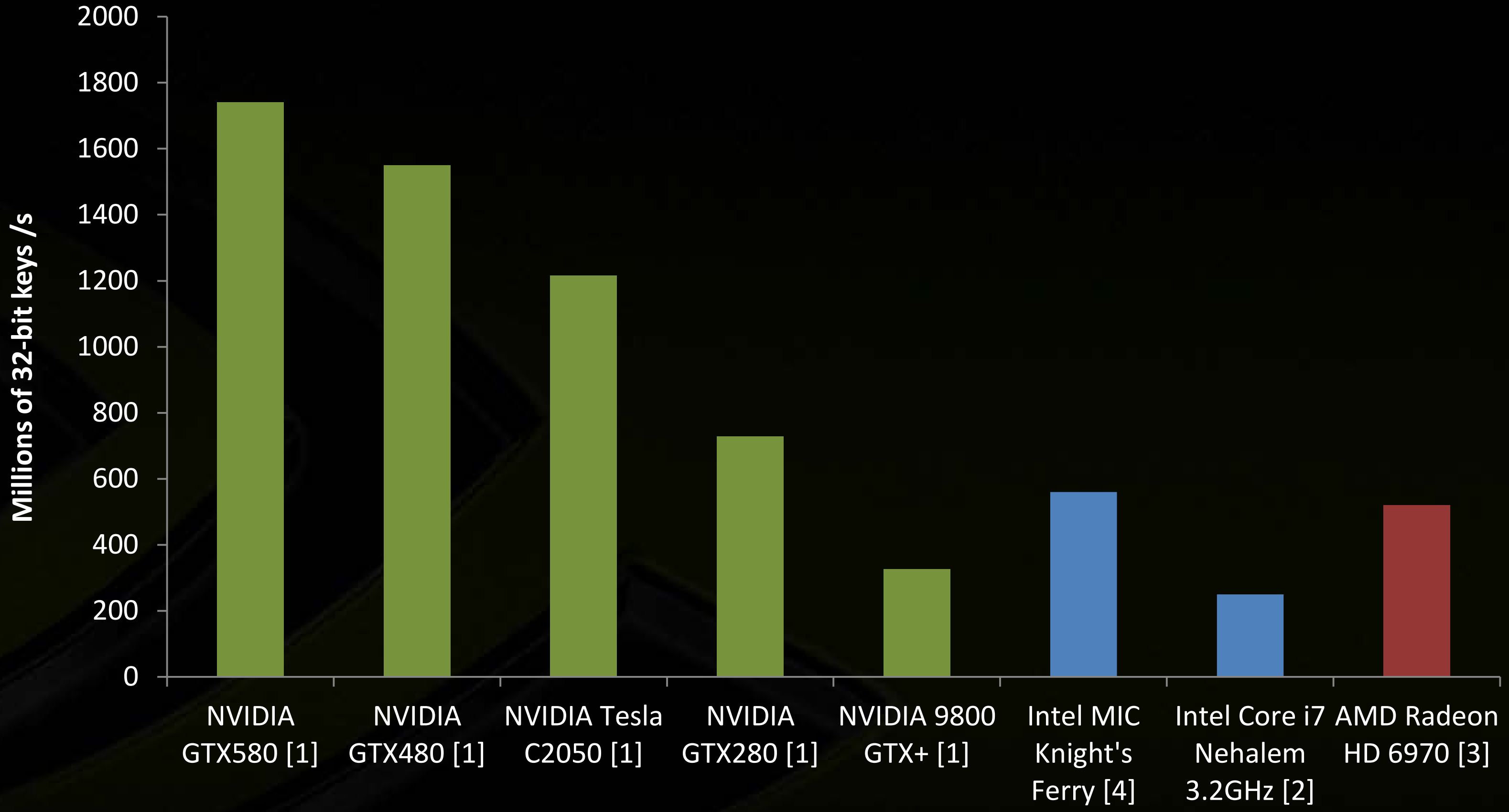
Tuning with flexible collectives

Example: radix sorting throughput
(initial GT200 effort ~2011)



[1] Merrill. Back40 GPU Primitives (2012)
[2] Satish et al. Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort (2010)
[3] T. Harada and L. Howes. Introduction to GPU Radix Sort (2011)
[4] Satish et al. Fast Sort on CPUs, GPUs, and Intel MIC Architectures. Intel Labs, 2010.

Radix sorting throughput (current)



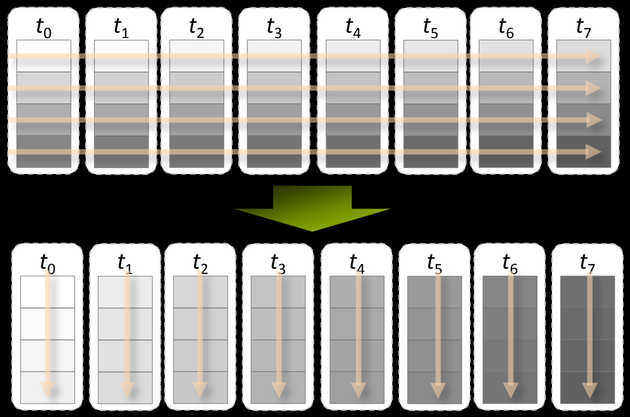
[1] Merrill. Back40, CUB GPU Primitives (2013)
[2] Satish et al. Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort (2010)
[3] T. Harada and L. Howes. Introduction to GPU Radix Sort (2011)
[4] Satish et al. Fast Sort on CPUs, GPUs, and Intel MIC Architectures. Intel Labs, 2010.

Fine-tuning primitives

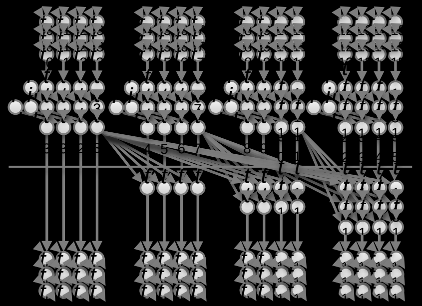
Tiled prefix sum

```
/**
 * Simple CUDA kernel for computing tiled partial sums
 */
template <int BLOCK_THREADS, int ITEMS_PER_THREAD, LoadAlgorithm LOAD_ALGO, ScanAlgorithm SCAN_ALGO>
__global__ void ScanTilesKernel(int *d_in, int *d_out)
{
      // Specialize collective types for problem context
      typedef cub::BlockLoad<int*, BLOCK_THREADS, ITEMS_PER_THREAD, LOAD_ALGO> BlockLoadT;
      typedef cub::BlockScan<int, BLOCK_THREADS, SCAN_ALGO> BlockScanT;

      // Allocate on-chip temporary storage
      __shared__ union {
          typename BlockLoadT::TempStorage load;
          typename BlockScanT::TempStorage reduce;
      } temp_storage;

      // Load data per thread
      int thread_data[ITEMS_PER_THREAD];
      int offset = blockIdx.x * (BLOCK_THREADS * ITEMS_PER_THREAD);
      BlockLoadT(temp_storage.load).Load(d_in + offset, offset);

      __syncthreads();

      // Compute the block-wide prefix sum
      BlockScanT(temp_storage).Sum(thread_data);
      …
}
```

Data is striped across threads for memory accesses



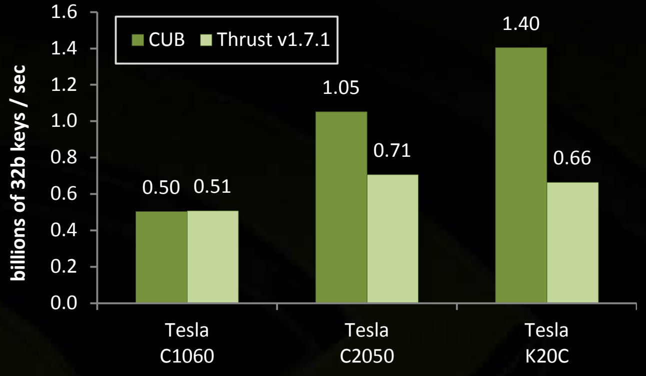
Data is blocked across threads for scanning



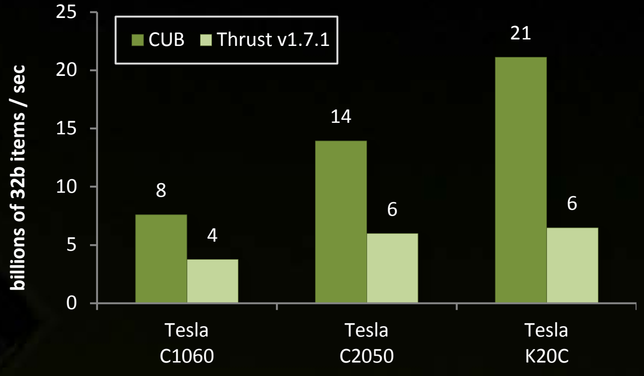
Scan data flow tiled from warpscans

CUB: device-wide performance-portability

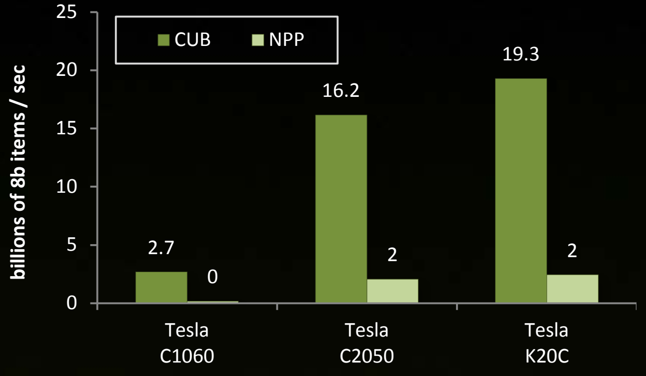
vs. *Thrust* and *NPP* across the last three major NVIDIA arch families (Telsa, Fermi, Kepler)



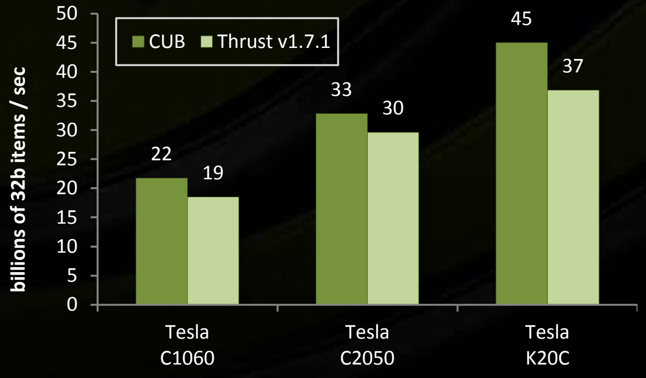
Global radix sort



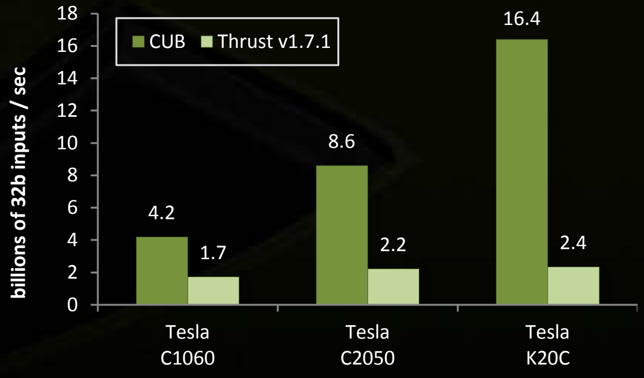
Global prefix scan



Global Histogram



Global reduction



Global partition-if

Summary

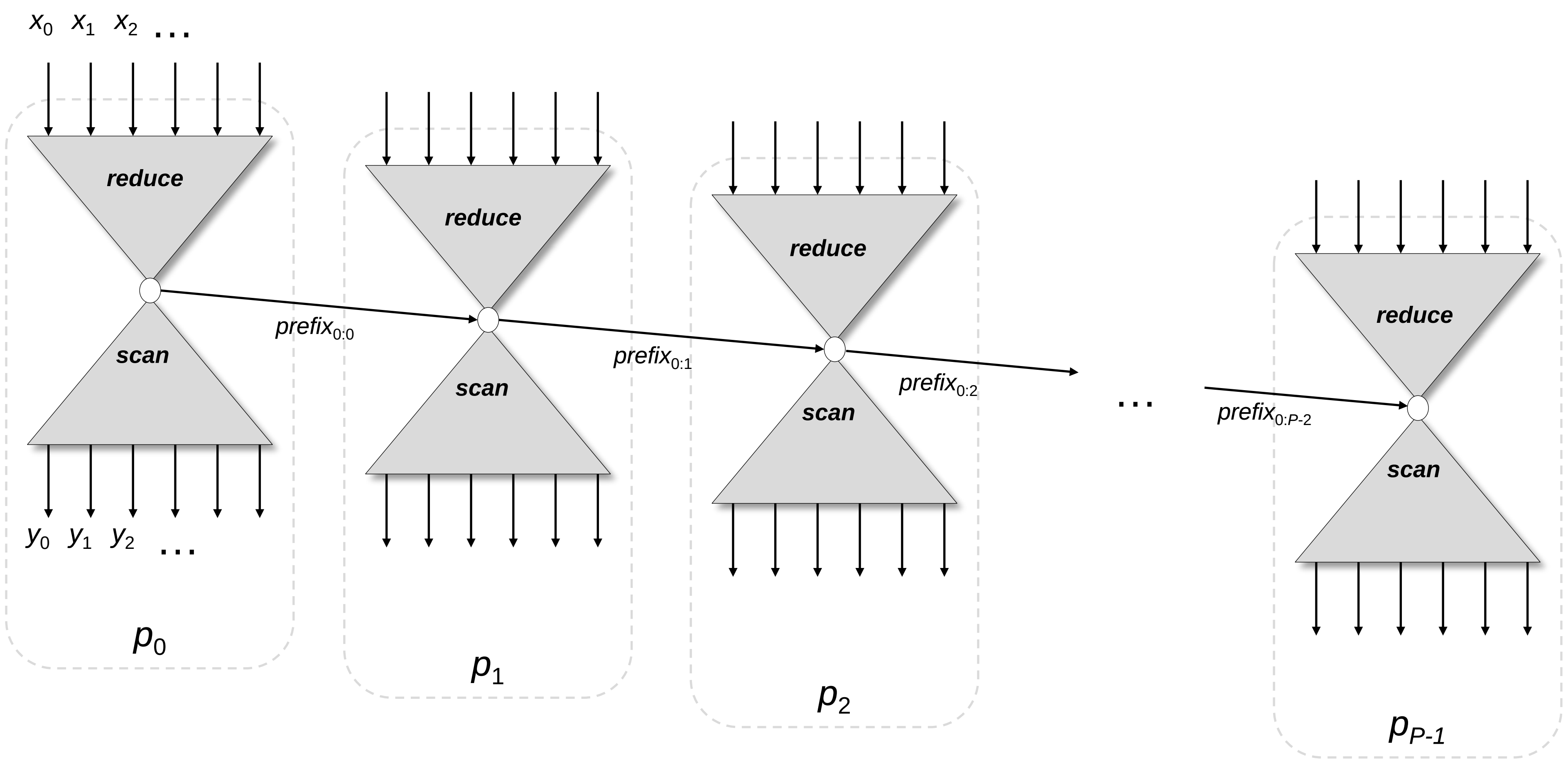
Summary: benefits of using CUB primitives

- ***Simplicity of composition***
  - Kernels are simply sequences of primitives (e.g., BlockLoad -> BlockSort -> BlockReduceByKey)
- ***High performance***
  - CUB uses the best known algorithms, abstractions, and strategies, and techniques
- ***Performance portability***
  - CUB is specialized for the target hardware (e.g., memory conflict rules, special instructions, etc.)
- ***Simplicity of tuning***
  - CUB adapts to various grain sizes (threads per block, items per thread, etc.)
  - CUB provides alterative algorithms
- ***Robustness and durability***
  - CUB supports arbitrary data types and block sizes

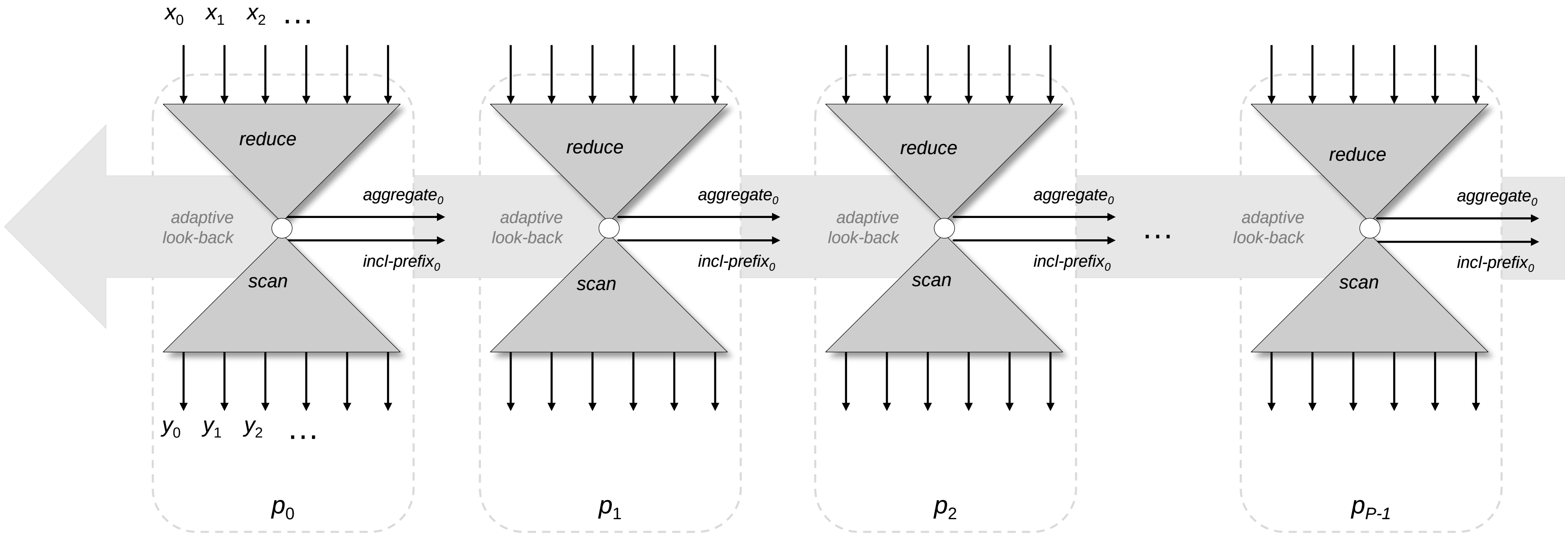
Questions?

Please visit the **CUB** project on GitHub
http://nvlabs.github.com/cub

Duane Merrill (dumerrill@nvidia.com)



x_0 x_1 x_2 …
reduce
scan
y_0 y_1 y_2 …
p_0
$prefix_{0:0}$
reduce
scan
p_1
$prefix_{0:1}$
reduce
scan
p_2
$prefix_{0:2}$
…
$prefix_{0:P-2}$
reduce
scan
p_{P-1}



Status flag	*P*	*A*	*P*	...	*X*
Aggregate	256	256	256	...	-
Inclusive prefix	256	-	768	...	-
	0	1	2		*P*-1