GPU TECHNOLOGY CONFERENCE

CUB :

A pattern of "collective" software design, abstraction, and reuse for kernel-level programming

DUANE MERRILL, PH.D.

NVIDIA RESEARCH

What is CUB?

1. A design model for *collective* kernel-level primitives
   - How to make reusable software components for SIMT groups (warps, blocks, etc.)
2. A library of collective primitives
   - Block-reduce, block-sort, block-histogram, warp-scan, warp-reduce, etc.
3. A library of global primitives (built from collectives)
   - Device-reduce, device-sort, device-scan, etc.
   - Demonstrate collective composition, performance, *performance-portability*

Outline

1. Software reuse
2. SIMT collectives: the “missing” CUDA abstraction layer
3. The soul of collective component design
4. Using CUB’s collective primitives
5. Making your own collective primitives
6. Other Very Useful Things in CUB
7. Final thoughts

Software reuse

Abstraction & *composability* are fundamental design principles

- **Reduce redundant programmer effort**
  - Save time, energy, money
  - Reduce buggy software
- **Encapsulate complexity**
  - Empower productivity-oriented programmers
  - Insulation from underlying hardware

Software reuse empowers a **durable** programming model

Software reuse

Abstraction & *composability* are fundamental design principles

- **Reduce redundant programmer effort**
  - Save time, energy, money
  - Reduce buggy software
- **Encapsulate complexity**
  - Empower productivity-oriented programmers
  - **Insulation from changing capabilities of the underlying hardware**
    - *NVIDIA has produced nine different CUDA GPU architectures since 2008!*

Software reuse empowers a <u>durable</u> programming model

Outline

1. Software reuse
2. SIMT collectives: the “missing” CUDA abstraction layer
3. The soul of collective component design
4. Using CUB’s collective primitives
5. Making your own collective primitives
6. Other Very Useful Things in CUB
7. Final thoughts

Parallel programming is **hard**…

No, <u>cooperative</u> parallel programming is hard…

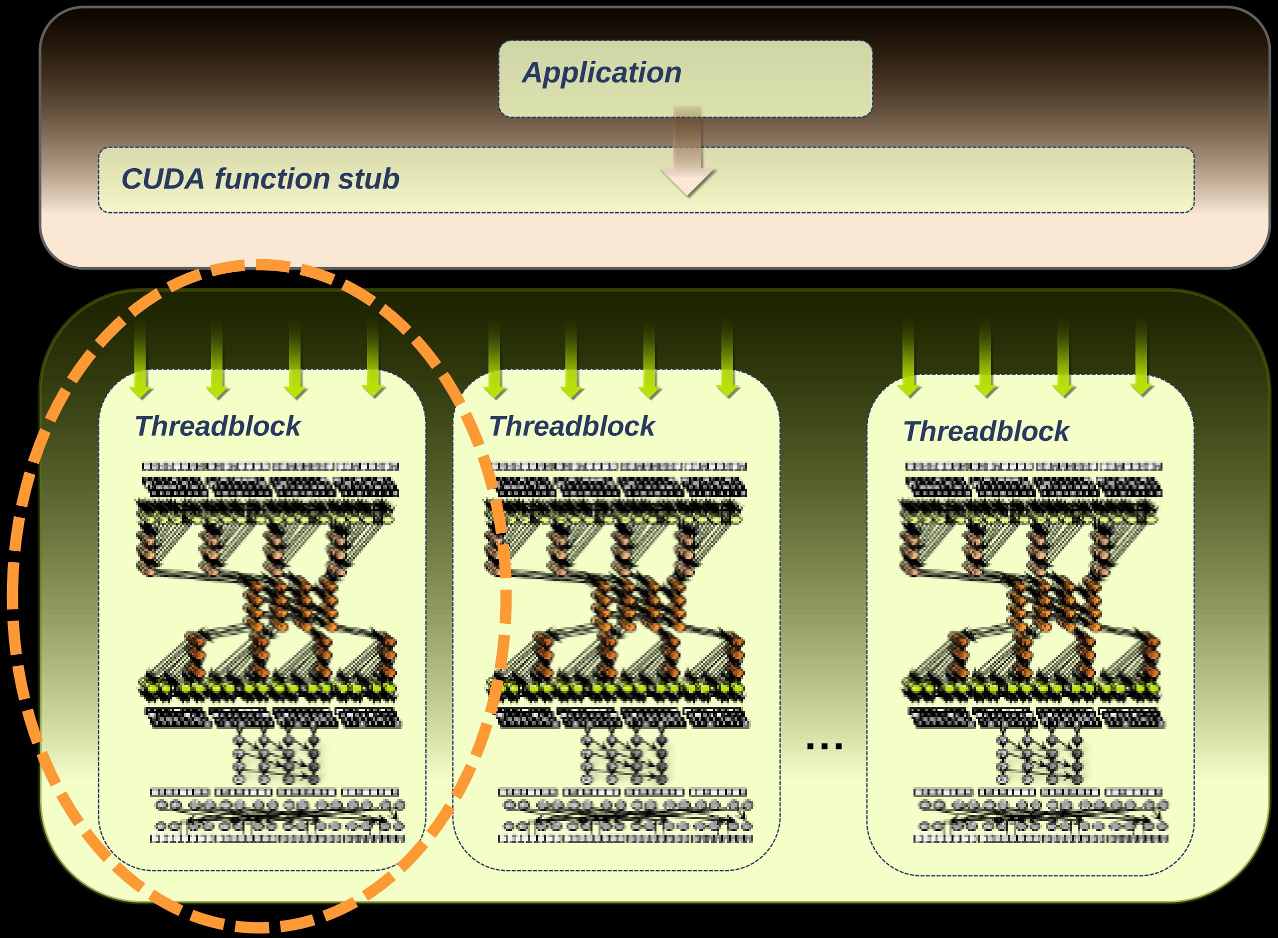
- Parallel decomposition and grain sizing
- Synchronization
- Deadlock, livelock, and data races
- Plurality of state
- Plurality of flow control (divergence, etc.)
- Bookkeeping control structures
- Memory access conflicts, coalescing, etc.
- Occupancy constraints from SMEM, RF, etc
- Algorithm selection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

No, cooperative parallel programming is hard…

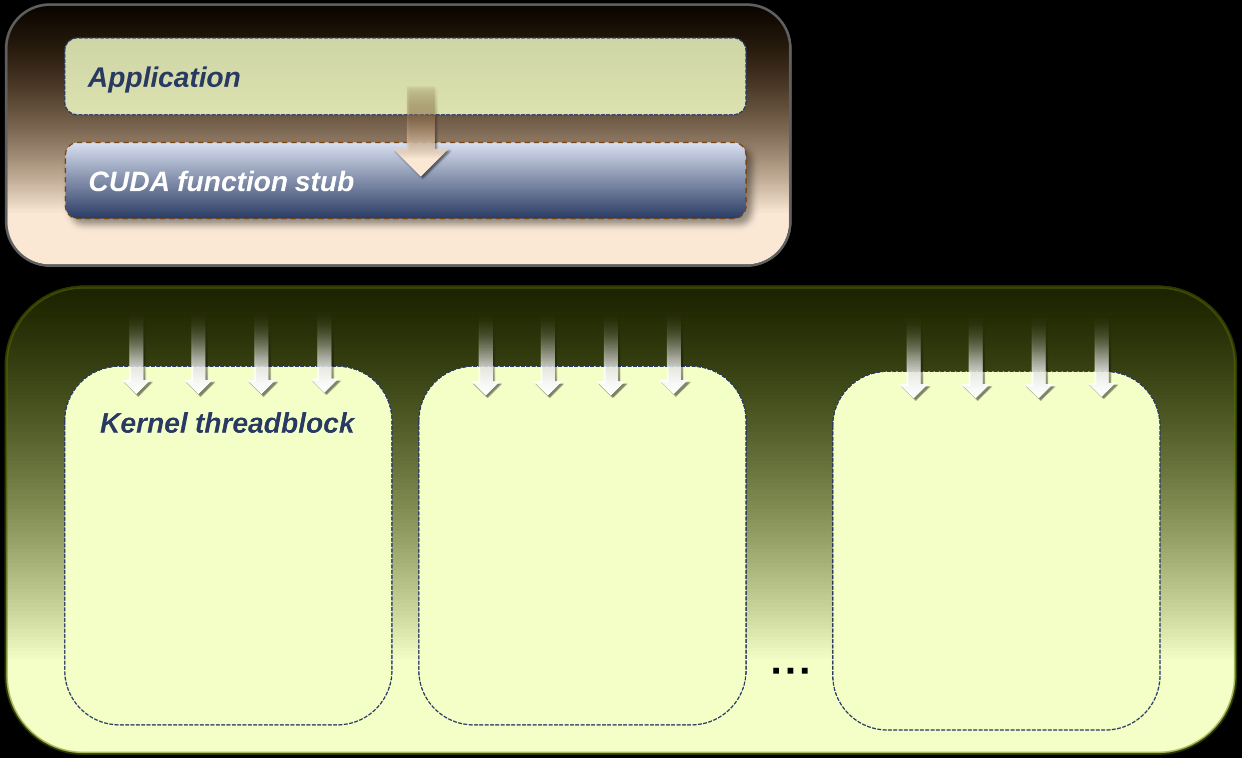
- Parallel decomposition and grain sizing
- Synchronization
- Deadlock, livelock, and data races
- Plurality of state
- Plurality of flow control (divergence, etc.)
- Bookkeeping control structures
- Memory access conflicts, coalescing, etc.
- Occupancy constraints from SMEM, RF, etc
- Algorithm selection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

…

CUDA today

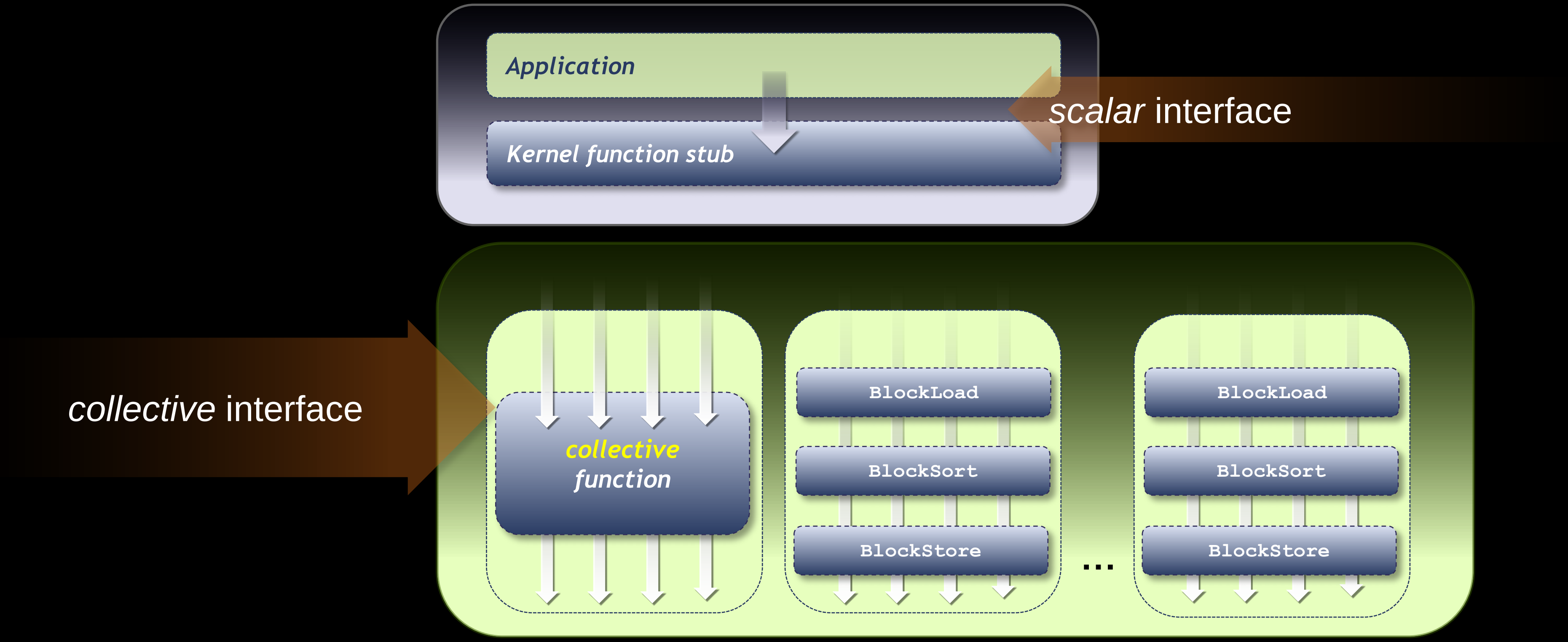

Application
CUDA function stub
Threadblock
Threadblock
...
Threadblock

Software abstraction in CUDA



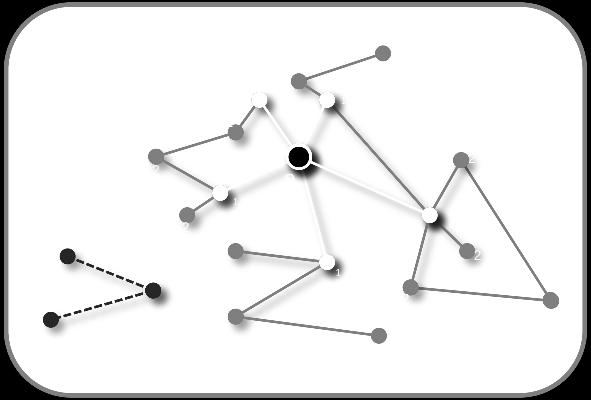
- **PROBLEM:** virtually every CUDA kernel written today is cobbled from scratch
  - A tunability, portability, and maintenance concern

Software abstraction in CUDA



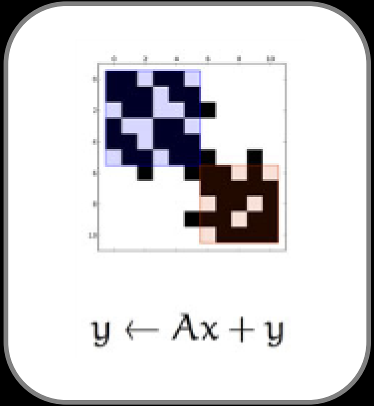
- Collective software components
  - reduce development cost, hide complexity, bugs, etc.

What do these applications have in common?

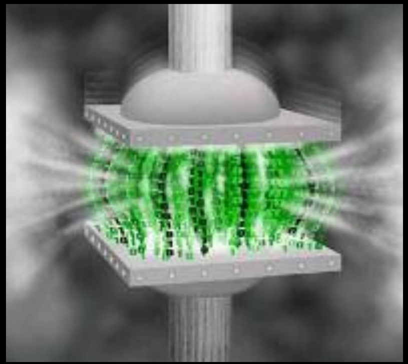


Parallel sparse graph traversal

Parallel radix sort



Parallel SpMV

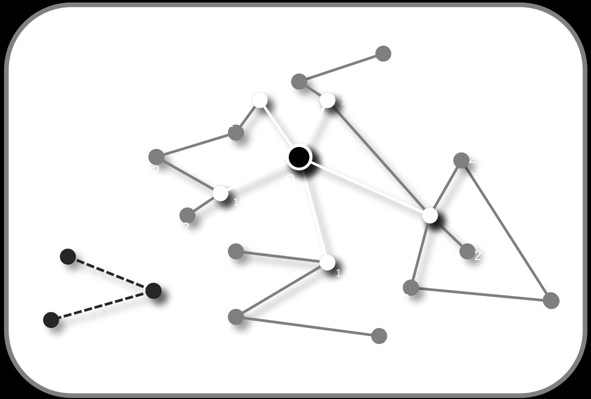


Parallel BWT compression

What do these applications have in common?

Block-wide prefix-scan

Scan for enqueueing

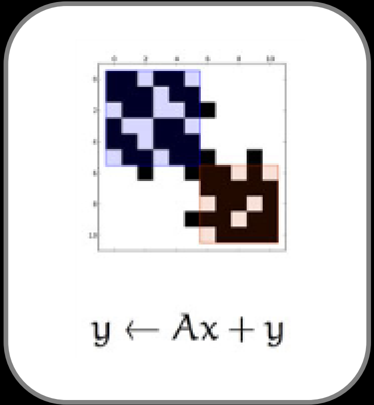


Parallel sparse graph traversal

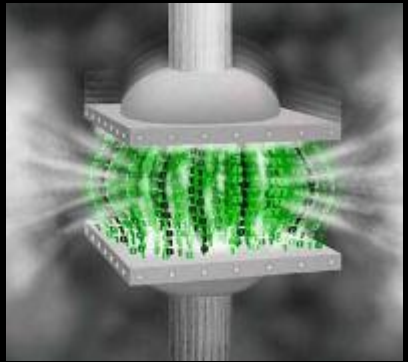
Parallel radix sort

Scan for partitioning

Scan for segmented reduction



Parallel SpMV

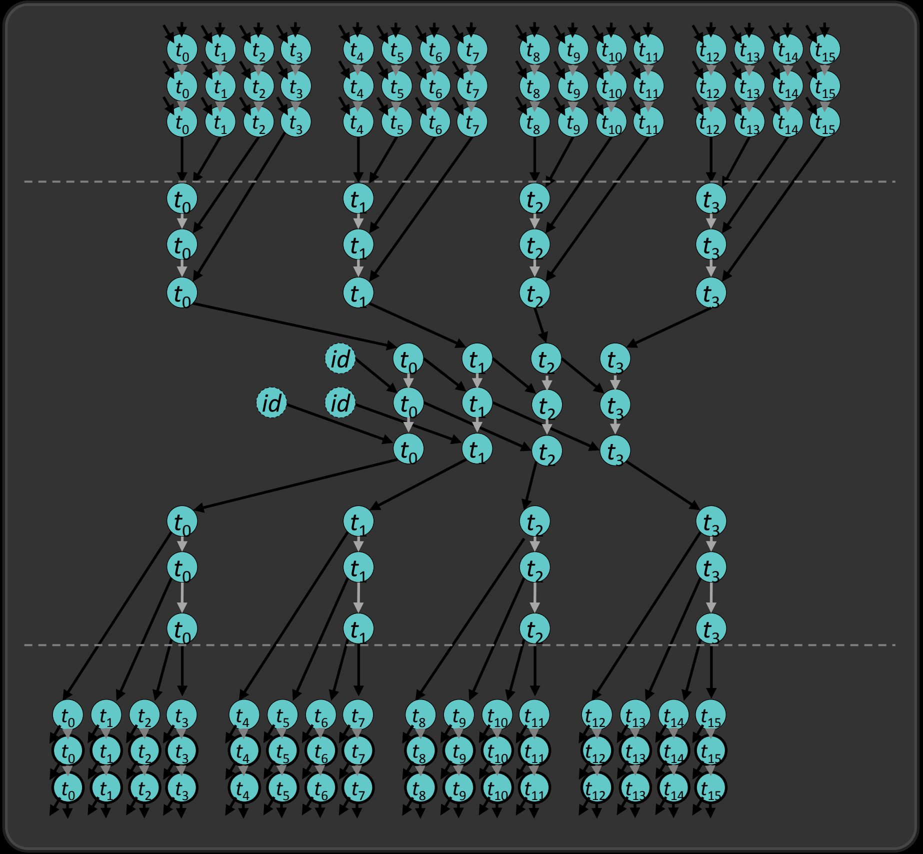


Parallel BWT compression

Scan for solving recurrences (move-to-front)

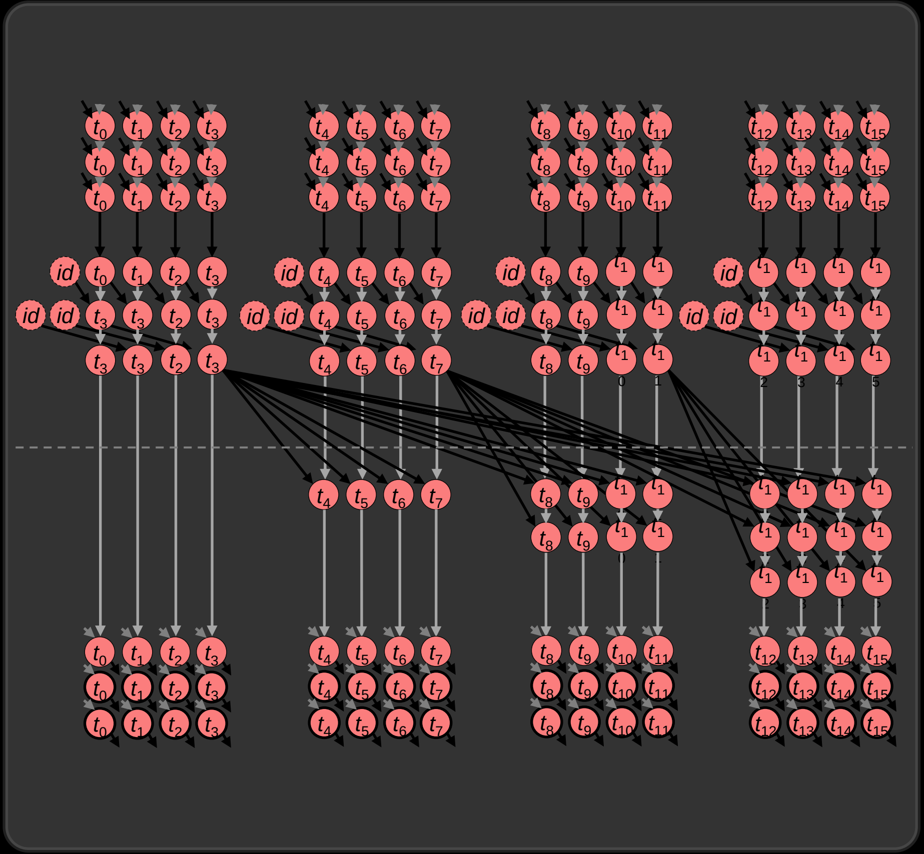
Examples of parallel scan data flow

16 threads contributing 4 items each



Brent-Kung hybrid

(Work-efficient ~130 binary ops, depth 15)

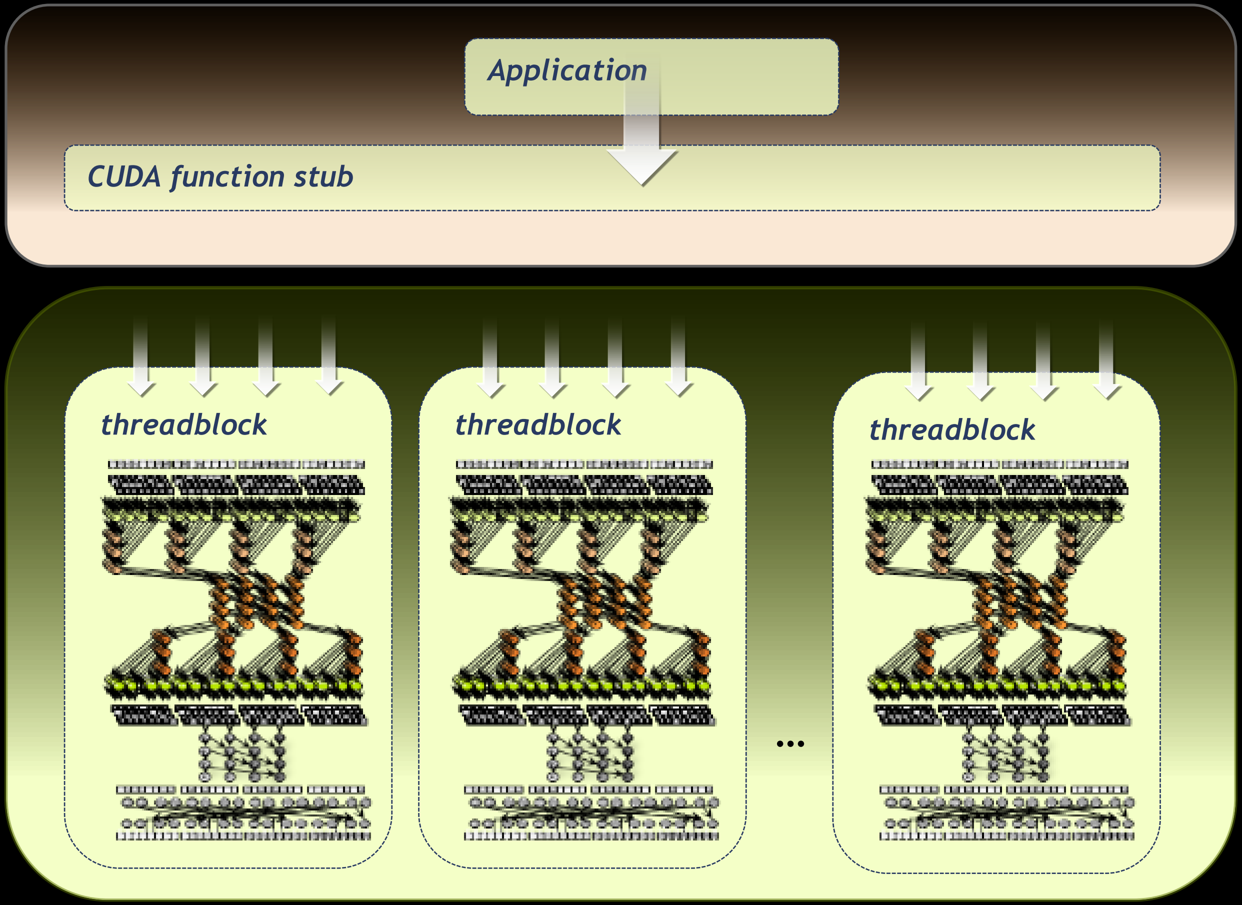


Kogge-Stone hybrid

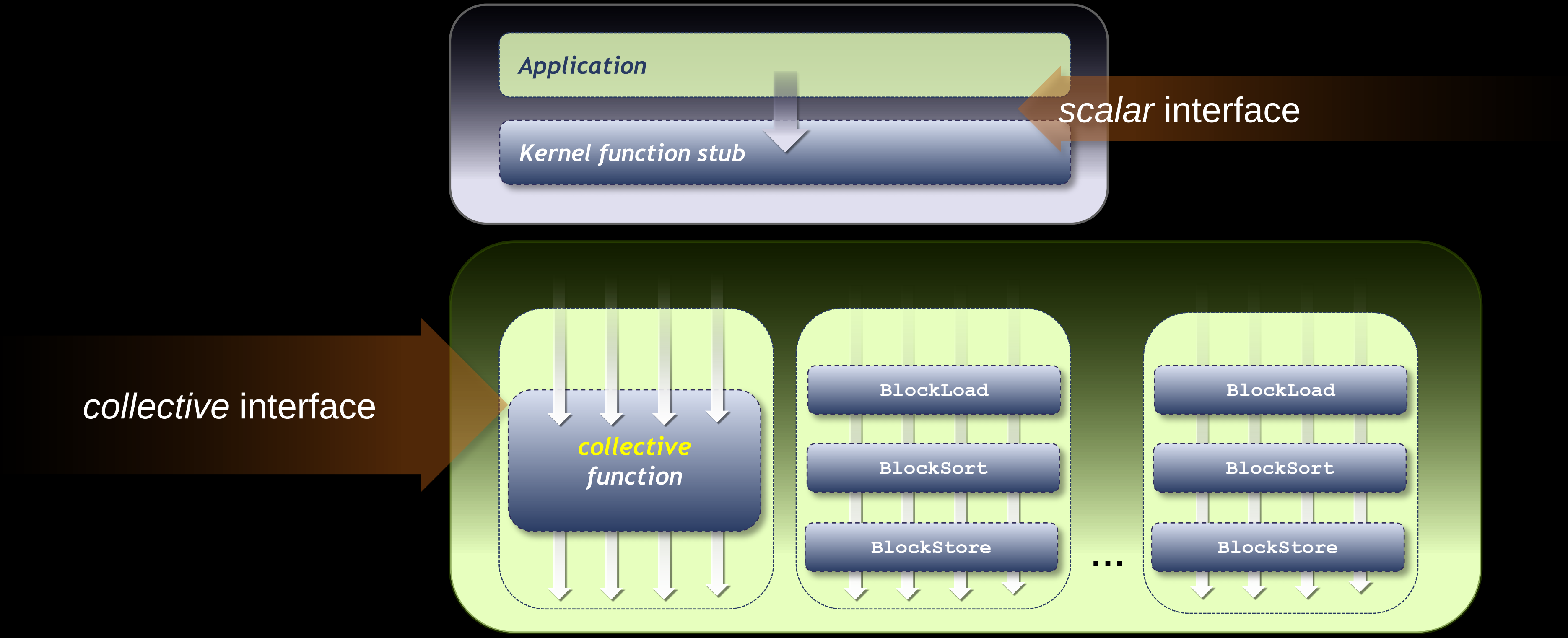
(Depth-efficient ~170 binary ops, depth 12)

CUDA today

Kernel programming is complicating



Software abstraction in CUDA



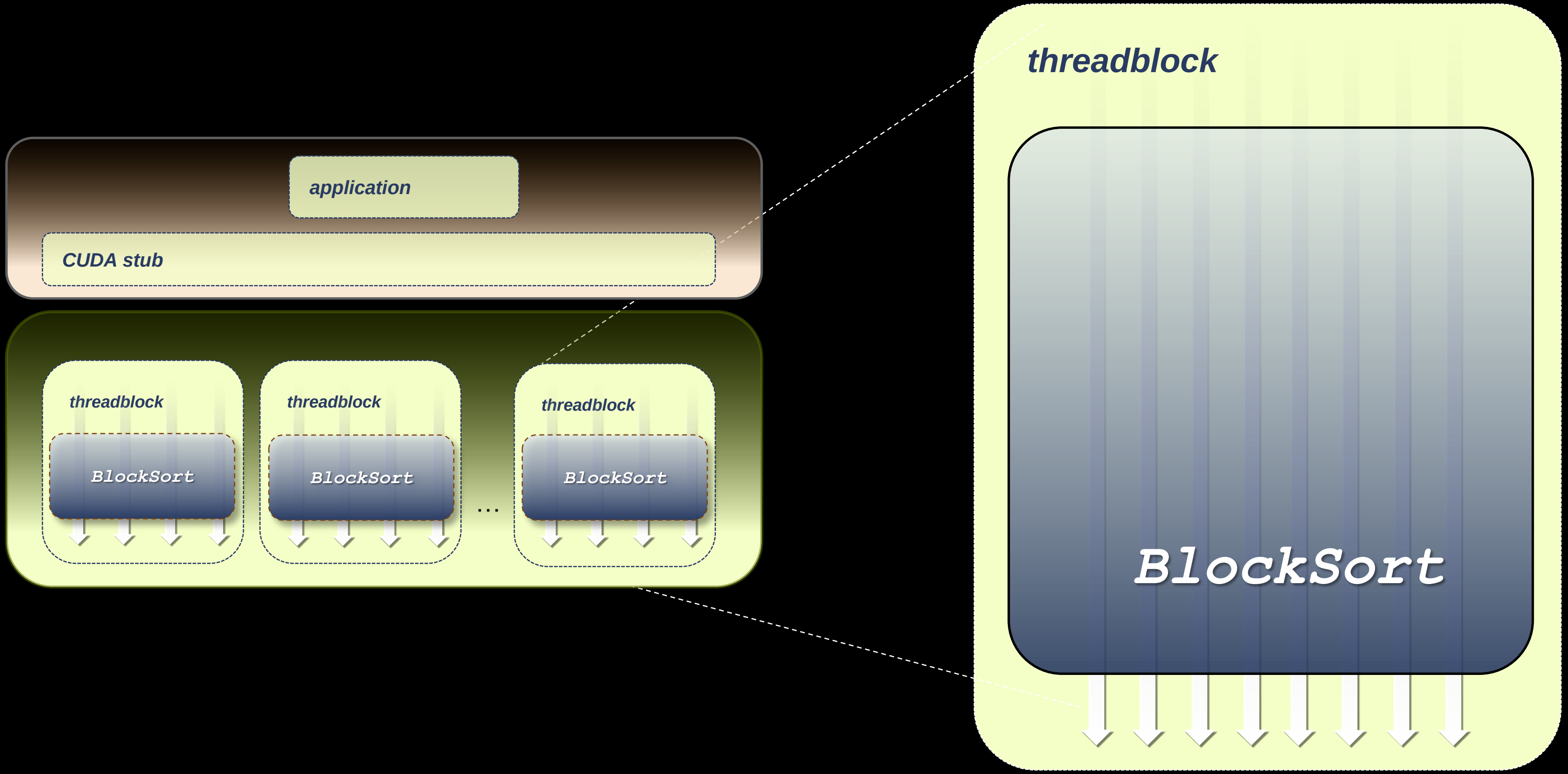
- Collective software components
  - reduce development cost, hide complexity, bugs, etc.

Outline

1. Software reuse
2. SIMT collectives: the “missing” CUDA abstraction layer
3. **The soul of collective component design**
4. Using CUB’s collective primitives
5. Making your own collective primitives
6. Other Very Useful Things in CUB
7. Final thoughts

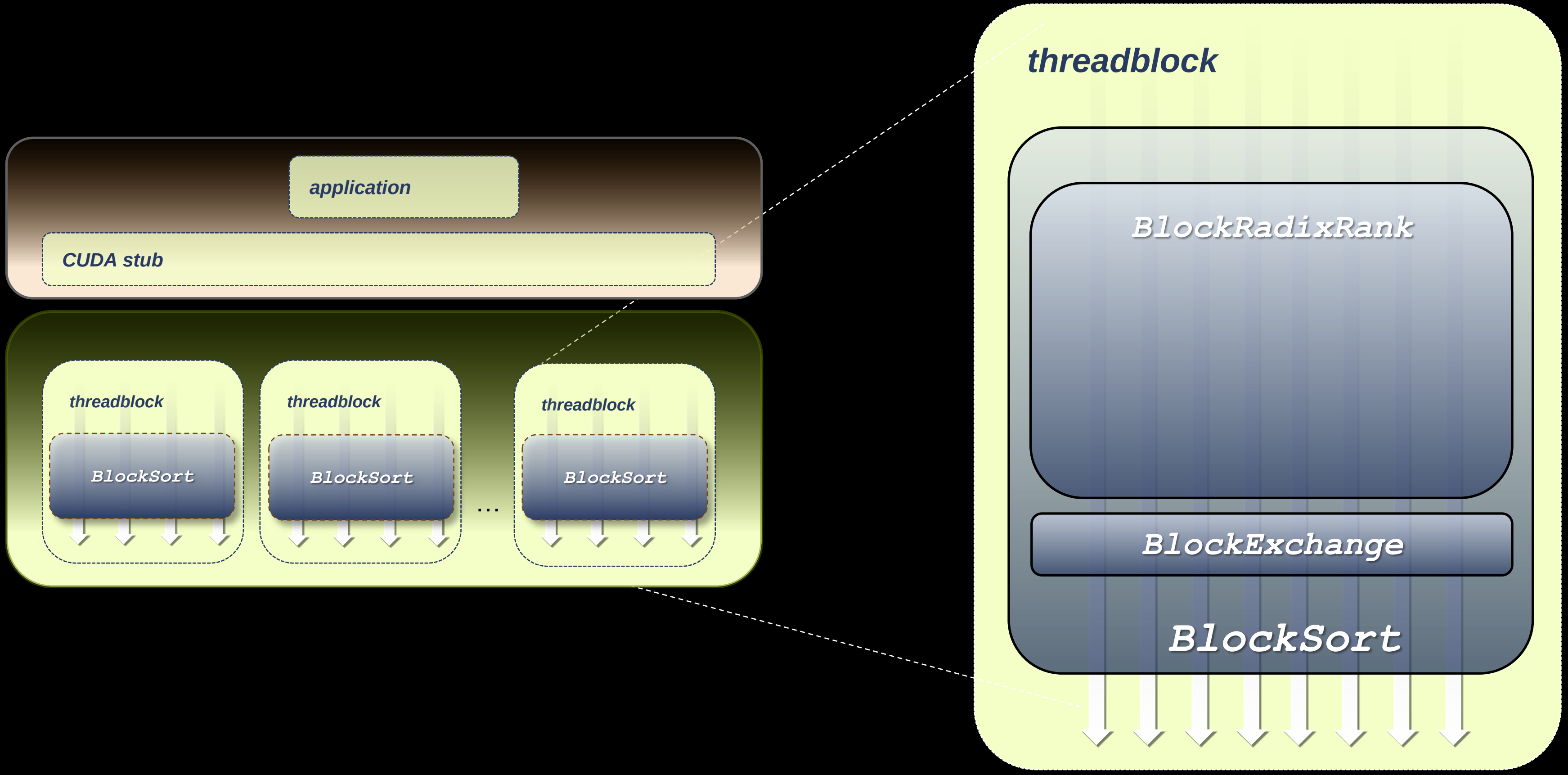
Collective composition

CUB primitives are easily nested & sequenced



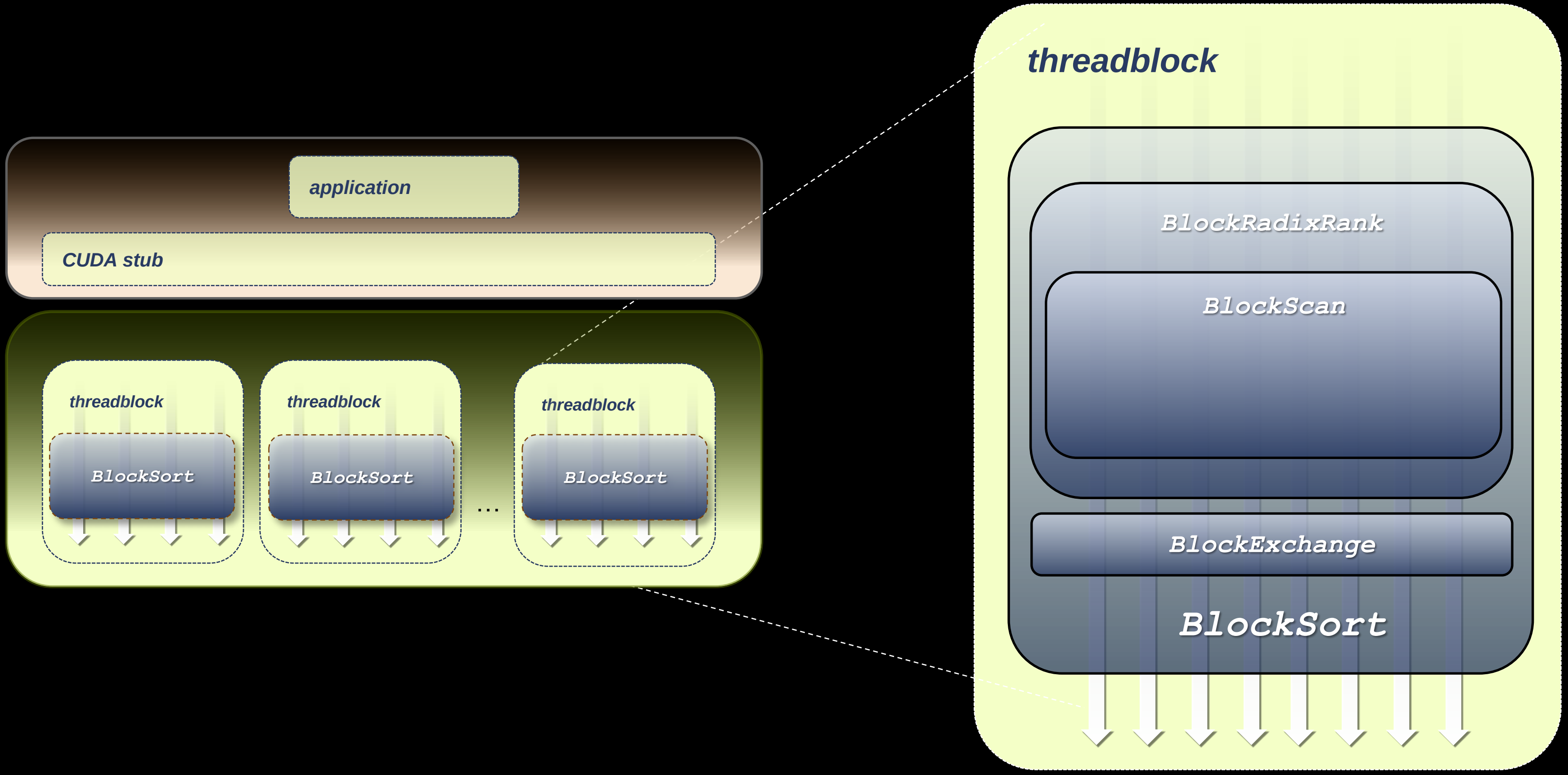
Collective composition

CUB primitives are easily nested & sequenced



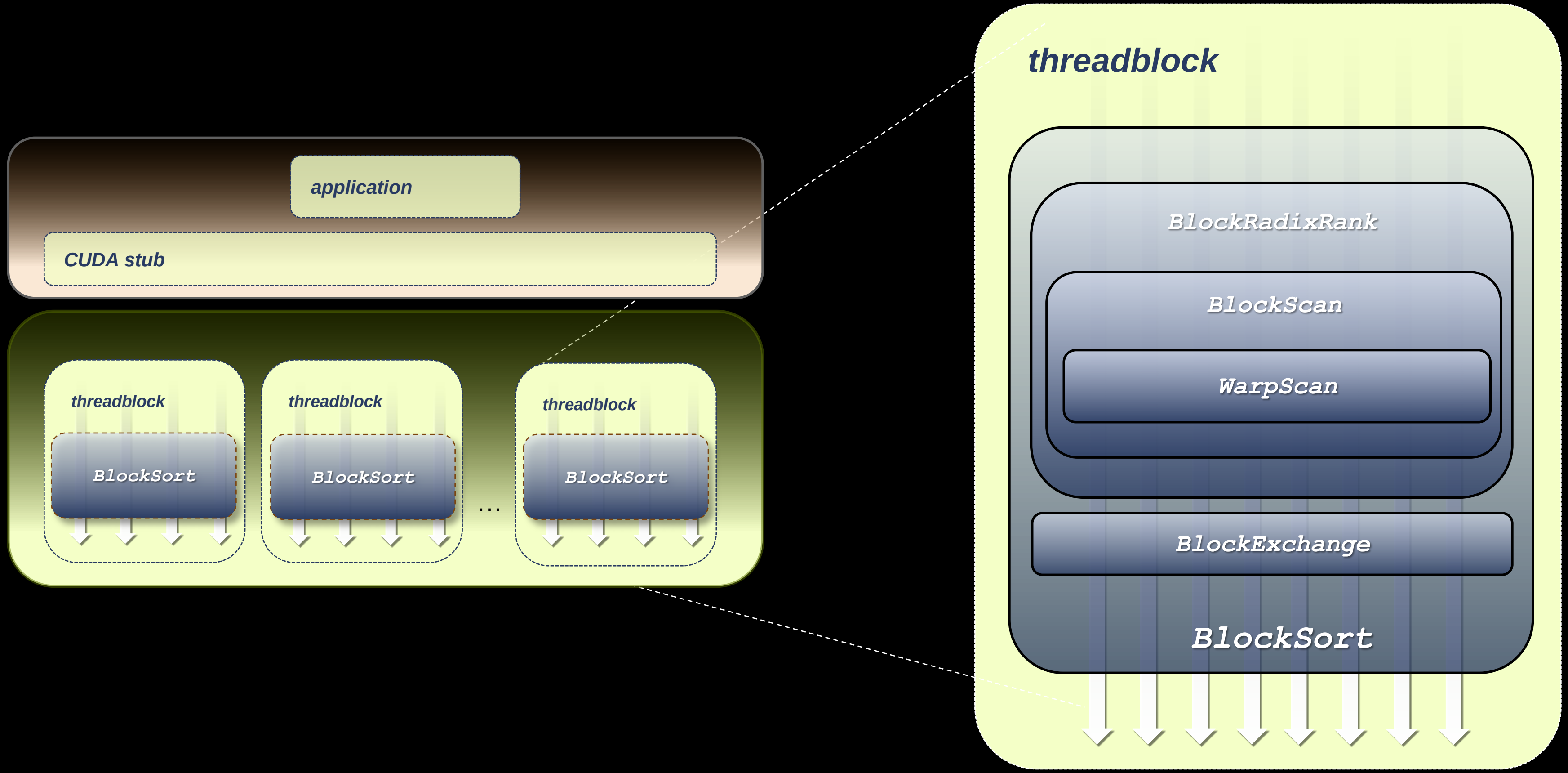
Collective composition

CUB primitives are easily nested & sequenced



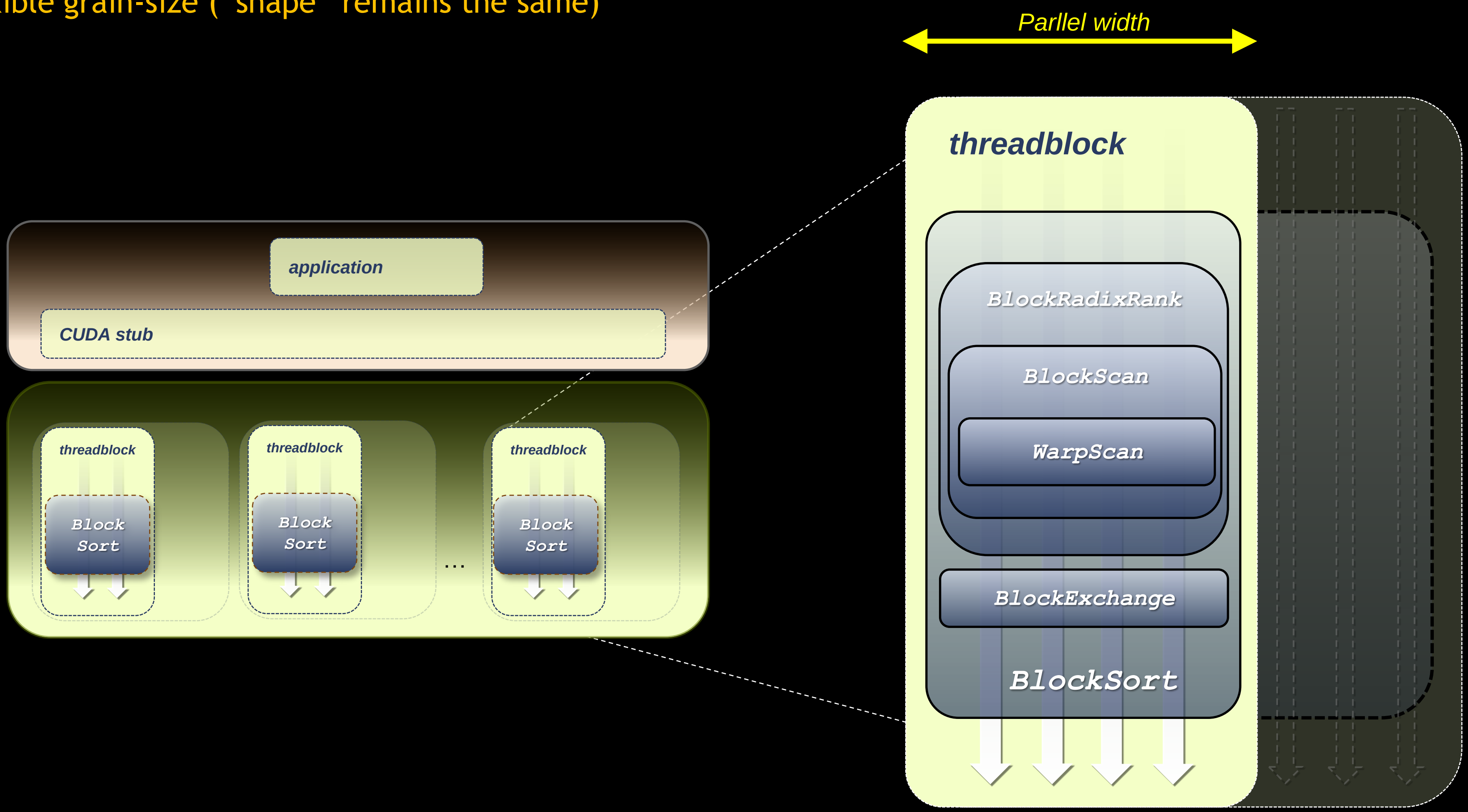
Collective composition

CUB primitives are easily nested & sequenced



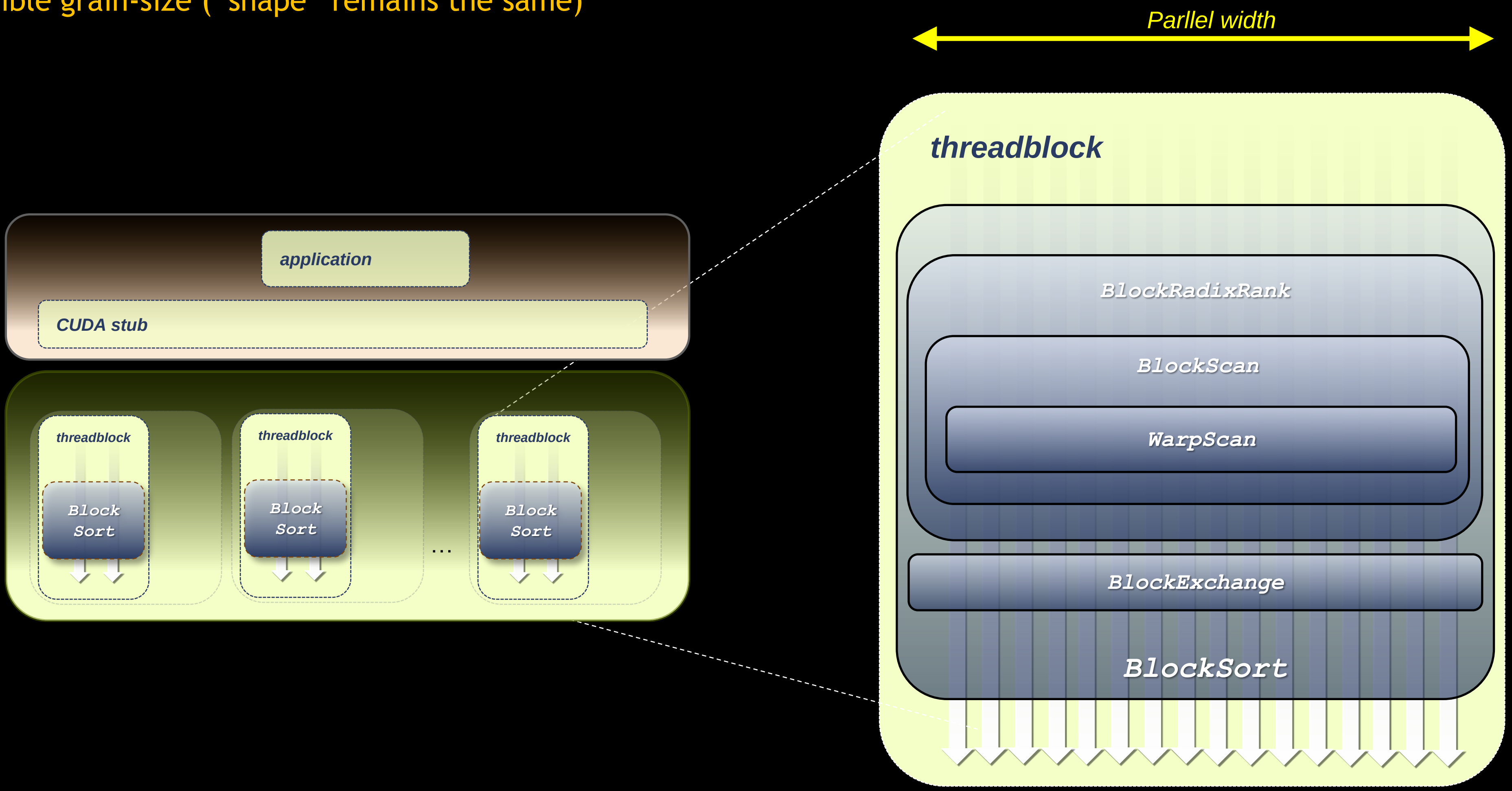
Tunable composition

Flexible grain-size ("shape" remains the same)



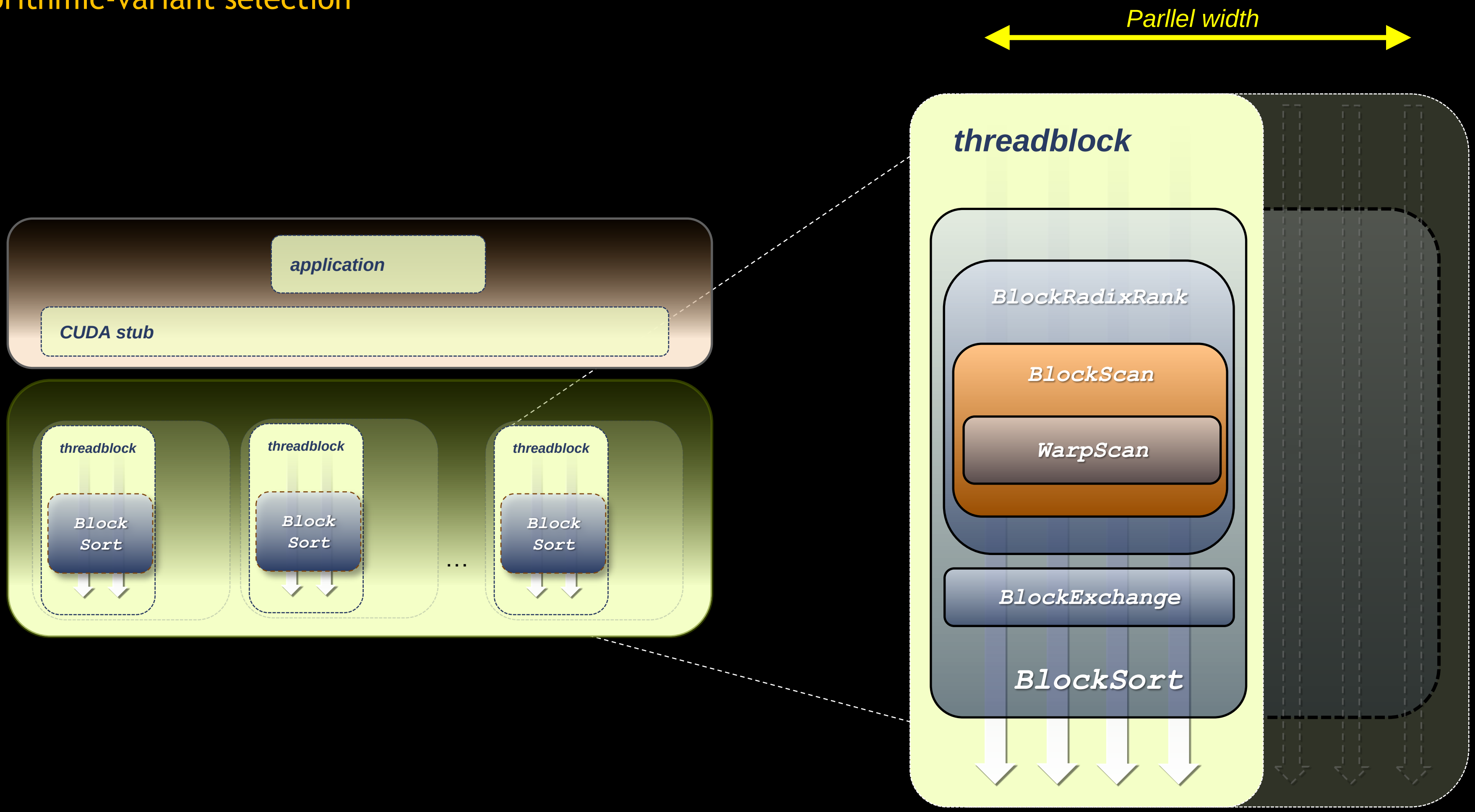
Tunable composition

Flexible grain-size ("shape" remains the same)



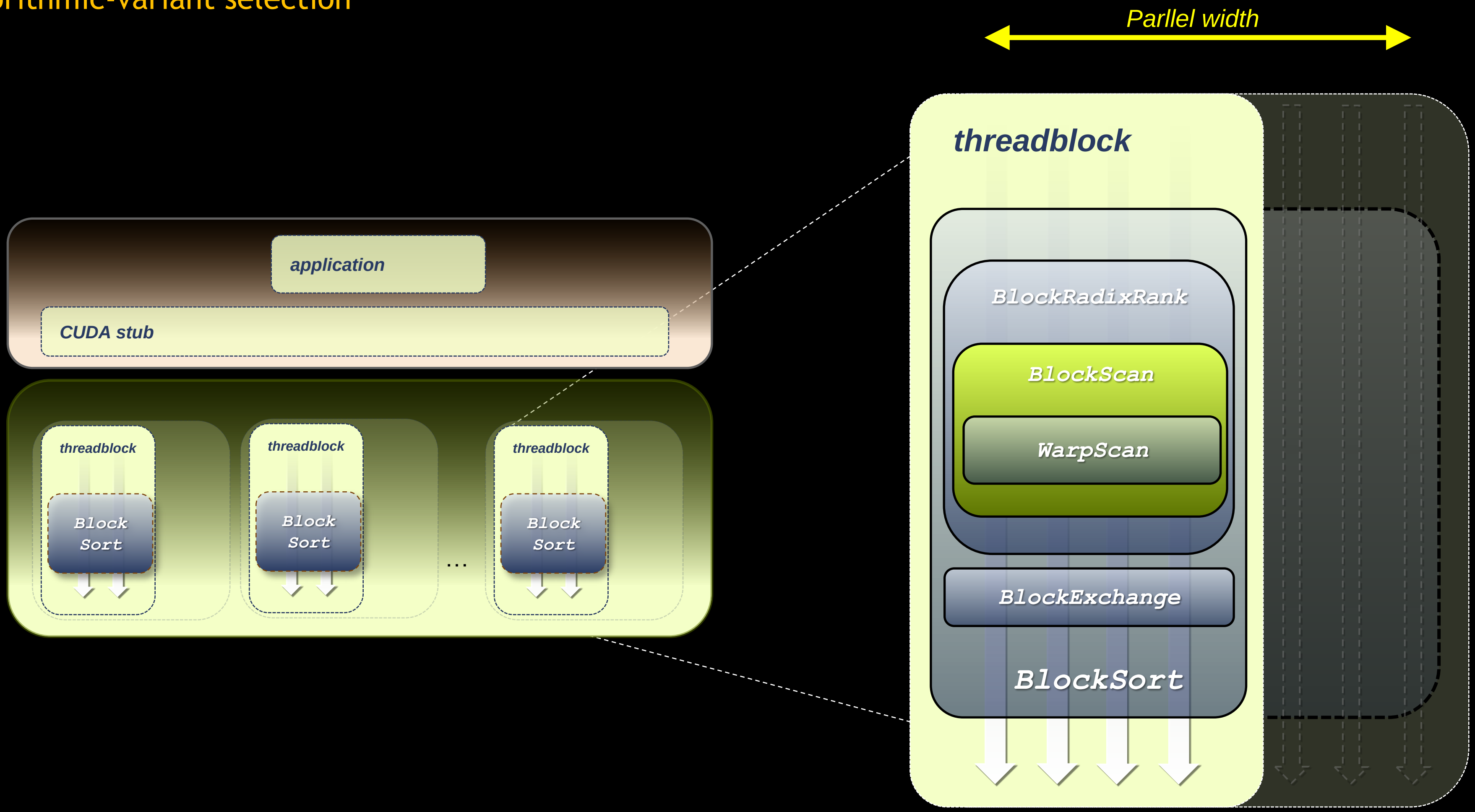
Tunable composition

Algorithmic-variant selection



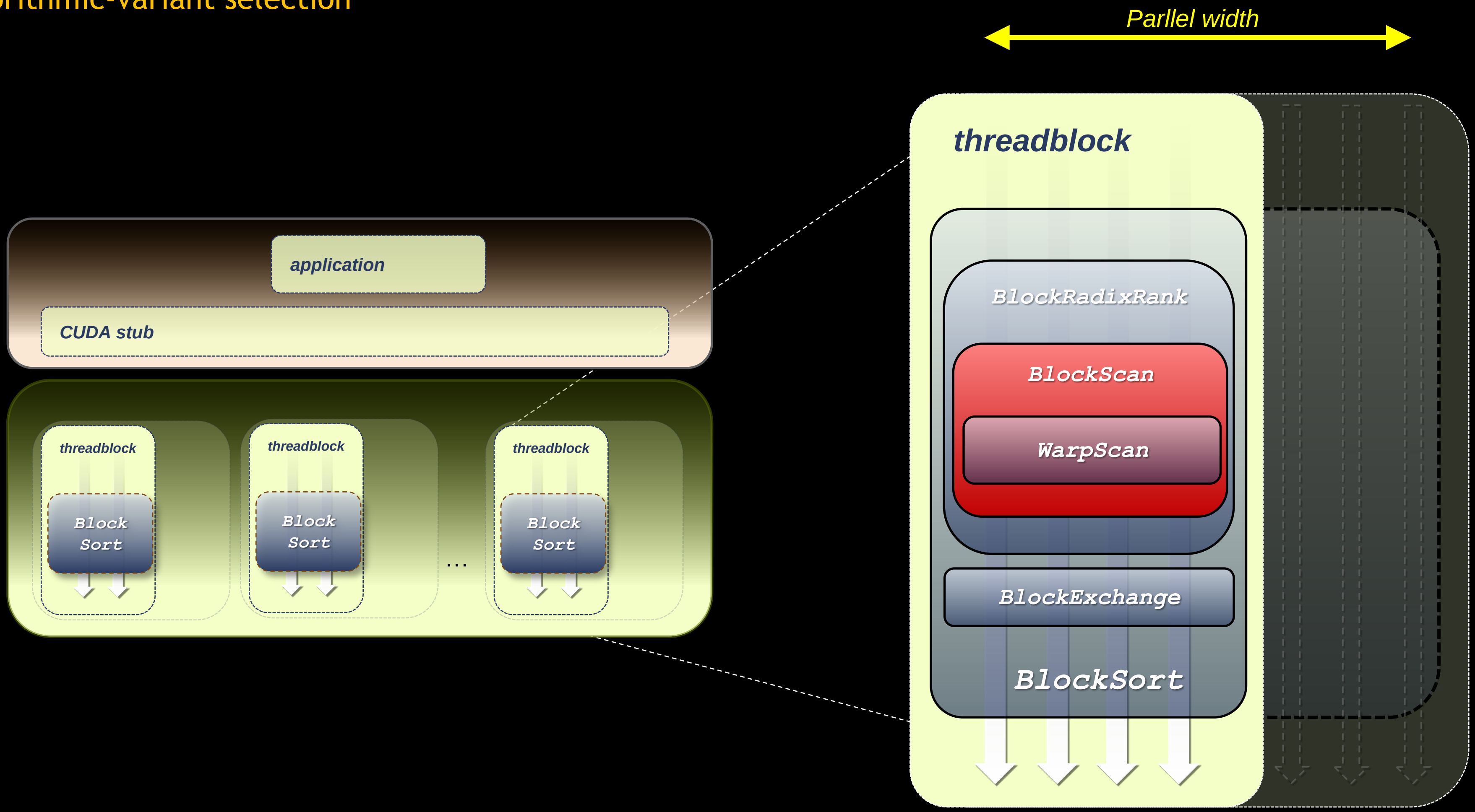
Tunable composition

Algorithmic-variant selection



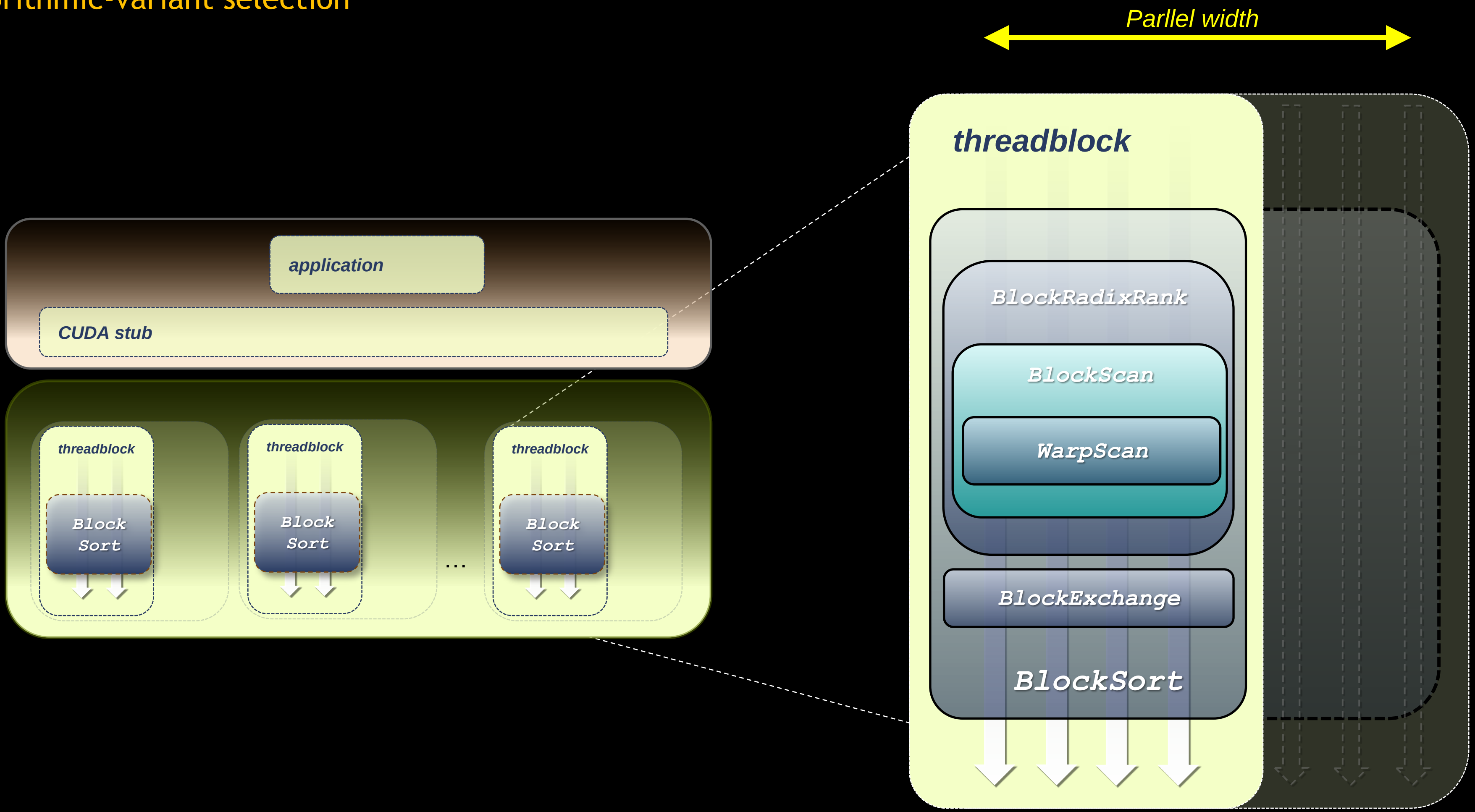
Tunable composition

Algorithmic-variant selection



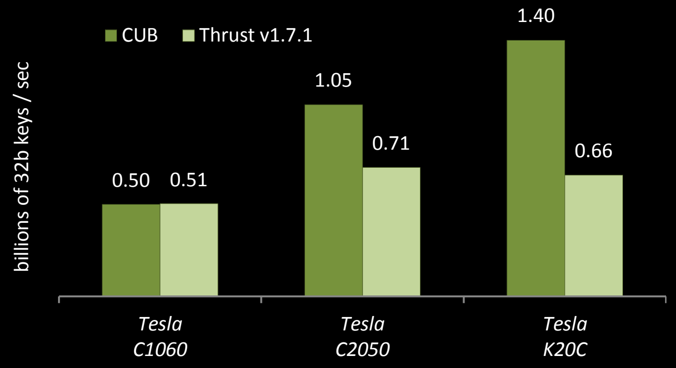
Tunable composition

Algorithmic-variant selection

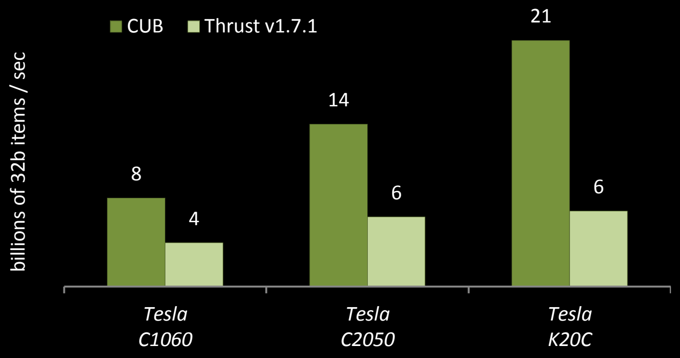


CUB: device-wide performance-portability

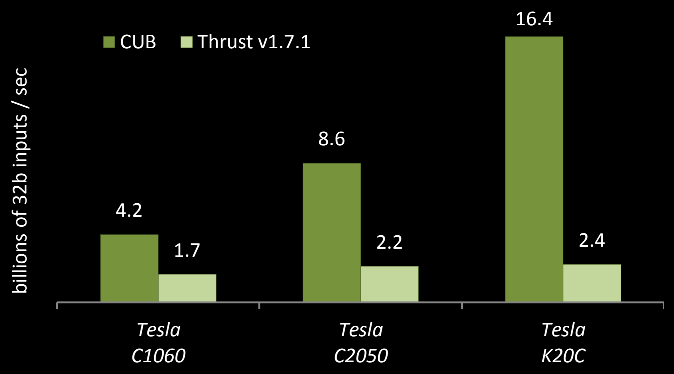
vs. *Thrust* and *NPP* across the last 4 major NVIDIA arch families (Telsa, Fermi, Kepler, Maxwell)



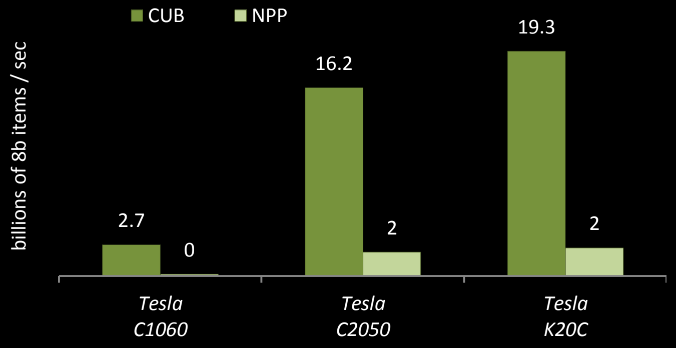
Global radix sort



Global prefix scan



Global partition-if



Global Histogram

Outline

1. Software reuse
2. SIMT collectives: the "missing" CUDA abstraction layer
3. The soul of collective component design
4. Using CUB's collective primitives
5. Making your own collective primitives
6. Other Very Useful Things in CUB
7. Final thoughts

CUB collective usage

```
__global__ void ExampleKernel(...)
{






}
```

CUB collective usage

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











}
```

1. Collective specialization

CUB collective usage

3 parameter fields (specialization, construction, function call) + resource reflection

```
__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;

}
```

1. Collective specialization
2. Reflected shared resource type

CUB collective usage

```
__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 tile of 512 items blocked across 128 threads
    int items[4];
    ...



}
```

1. Collective specialization
2. Reflected shared resource type

CUB collective usage

```
__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 tile of 512 items blocked across 128 threads
        int items[4];
        ...

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

1. Collective specialization
2. Reflected shared resource type
3. Collective construction
4. Collective function call

CUB collective usage

3 parameter fields (specialization, construction, function call) + resource reflection

```
__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 tile of 512 items blocked across 128 threads
        int items[4];
        ...

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

1. Collective specialization
2. Reflected shared resource type
3. Collective construction
4. Collective function call

Sequencing CUB primitives

```
// A kernel for computing tiled prefix sums
__global__ void ExampleKernel(int* d_in, int* d_out)
{















}
```

Sequencing CUB primitives

```
// A kernel for computing tiled prefix sums
__global__ void ExampleKernel(int* d_in, int* d_out)
{
    // Specialize for 128 threads owning 4 integers each
    typedef cub::BlockLoad<int*, 128, 4>     BlockLoadT;
    typedef cub::BlockScan<int, 128>         BlockScanT;
    typedef cub::BlockStore<int*, 128, 4>    BlockStoreT;















}
```

1. Specialize the collective primitive types

Sequencing CUB primitives

```
// A kernel for computing tiled prefix sums
__global__ void ExampleKernel(int* d_in, int* d_out)
{
    // Specialize for 128 threads owning 4 integers each
    typedef cub::BlockLoad<int*, 128, 4>    BlockLoadT;
    typedef cub::BlockScan<int, 128>        BlockScanT;
    typedef cub::BlockStore<int*, 128, 4>   BlockStoreT;

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

}
```

1. Specialize the collective primitive types
2. Allocate shared memory with a union of TempStorage structured-layout types

Sequencing CUB primitives

```
// A kernel for computing tiled prefix sums
__global__ void ExampleKernel(int* d_in, int* d_out)
{
    // Specialize for 128 threads owning 4 integers each
    typedef cub::BlockLoad<int*, 128, 4>    BlockLoadT;
    typedef cub::BlockScan<int, 128>        BlockScanT;
    typedef cub::BlockStore<int*, 128, 4>   BlockStoreT;

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

    // Cooperatively load a tile of 512 items across 128 threads
    int items[4];
    BlockLoadT(temp_storage.load).Load(d_in, items);









}
```

1. Specialize the collective primitive types
2. Allocate shared memory with a union of TempStorage structured-layout types
3. Block-wide load

Sequencing CUB primitives

```
// A kernel for computing tiled prefix sums
__global__ void ExampleKernel(int* d_in, int* d_out)
{
    // Specialize for 128 threads owning 4 integers each
    typedef cub::BlockLoad<int*, 128, 4>    BlockLoadT;
    typedef cub::BlockScan<int, 128>        BlockScanT;
    typedef cub::BlockStore<int*, 128, 4>   BlockStoreT;

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

    // Cooperatively load a tile of 512 items across 128 threads
    int items[4];
    BlockLoadT(temp_storage.load).Load(d_in, items);

    __syncthreads();  // Barrier for smem reuse









}
```

1. Specialize the collective primitive types
2. Allocate shared memory with a union of TempStorage structured-layout types
3. Block-wide load,
4. barrier

Sequencing CUB primitives

```
// A kernel for computing tiled prefix sums
__global__ void ExampleKernel(int* d_in, int* d_out)
{
    // Specialize for 128 threads owning 4 integers each
    typedef cub::BlockLoad<int*, 128, 4>    BlockLoadT;
    typedef cub::BlockScan<int, 128>        BlockScanT;
    typedef cub::BlockStore<int*, 128, 4>   BlockStoreT;

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

    // Cooperatively load a tile of 512 items across 128 threads
    int items[4];
    BlockLoadT(temp_storage.load).Load(d_in, items);

    __syncthreads();  // Barrier for smem reuse

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

1. Specialize the collective primitive types
2. Allocate shared memory with a union of TempStorage structured-layout types
3. Block-wide load,
4. barrier,
5. block-wide scan

Sequencing CUB primitives

```
// A kernel for computing tiled prefix sums
__global__ void ExampleKernel(int* d_in, int* d_out)
{
    // Specialize for 128 threads owning 4 integers each
    typedef cub::BlockLoad<int*, 128, 4>    BlockLoadT;
    typedef cub::BlockScan<int, 128>        BlockScanT;
    typedef cub::BlockStore<int*, 128, 4>   BlockStoreT;

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

    // Cooperatively load a tile of 512 items across 128 threads
    int items[4];
    BlockLoadT(temp_storage.load).Load(d_in, items);

    __syncthreads();  // Barrier for smem reuse

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

    __syncthreads();  // Barrier for smem reuse



}
```

1. Specialize the collective primitive types

2. Allocate shared memory with a union of TempStorage structured-layout types

3. Block-wide load,
4. barrier,
5. block-wide scan,
6. barrier

Sequencing CUB primitives

```
// A kernel for computing tiled prefix sums
__global__ void ExampleKernel(int* d_in, int* d_out)
{
    // Specialize for 128 threads owning 4 integers each
    typedef cub::BlockLoad<int*, 128, 4>    BlockLoadT;
    typedef cub::BlockScan<int, 128>        BlockScanT;
    typedef cub::BlockStore<int*, 128, 4>   BlockStoreT;

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

    // Cooperatively load a tile of 512 items across 128 threads
    int items[4];
    BlockLoadT(temp_storage.load).Load(d_in, items);

    __syncthreads();  // Barrier for smem reuse

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

    __syncthreads();  // Barrier for smem reuse

    // Cooperatively store a tile of 512 items across 128 threads
    BlockStoreT(temp_storage.load).Store(d_in, items);
}
```

1. Specialize the collective primitive types

2. Allocate shared memory with a union of TempStorage structured-layout types

3. Block-wide load,
4. barrier,
5. block-wide scan
6. barrier,
7. block-wide store

Tuning with CUB primitives

```
// A kernel for computing tiled prefix sums
__global__ void ExampleKernel(int* d_in, int* d_out)
{
    // Specialize for 128 threads owning 4 integers each
    typedef cub::BlockLoad<int*, 128, 4>    BlockLoadT;
    typedef cub::BlockScan<int, 128>        BlockScanT;
    typedef cub::BlockStore<int*, 128, 4>   BlockStoreT;

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

    // Cooperatively load a tile of 512 items across 128 threads
    int items[4];
    BlockLoadT(temp_storage.load).Load(d_in, items);

    __syncthreads();  // Barrier for smem reuse

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

    __syncthreads();  // Barrier for smem reuse

    // Cooperatively store a tile of 512 items across 128 threads
    BlockStoreT(temp_storage.load).Store(d_in, items);
}
```

```
int* d_in;  // = ...
int* d_out; // = ...

// Invoke kernel (GF110 Fermi)
ExampleKernel <<<1, 128>>>(
    d_in,
    d_out);
```

Tuning with CUB primitives

```
template <typename T>
__global__ void ExampleKernel(T* d_in, T* d_out)
{
    // Specialize for 128 threads owning 4 Ts each
    typedef cub::BlockLoad<T*, 128, 4>    BlockLoadT;
    typedef cub::BlockScan<T, 128>        BlockScanT;
    typedef cub::BlockStore<T*, 128, 4>   BlockStoreT;

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

    // Cooperatively load a tile of 512 items across 128 threads
    T items[4];
    BlockLoadT(temp_storage.load).Load(d_in, items);

    __syncthreads();  // Barrier for smem reuse

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

    __syncthreads();  // Barrier for smem reuse

    // Cooperatively store a tile of 512 items across 128 threads
    BlockStoreT(temp_storage.load).Store(d_in, items);
}
```

```
int* d_in;  // = ...
int* d_out; // = ...

// Invoke kernel (GF110 Fermi)
ExampleKernel <<<1, 128>>>(
    d_in,
    d_out);
```

Tuning with CUB primitives

```
template <int BLOCK_THREADS, typename T>
__global__ void ExampleKernel(T* d_in, T* d_out)
{
    // Specialize for BLOCK_THREADS threads owning 4 integers each
    typedef cub::BlockLoad<T*, BLOCK_THREADS, 4>    BlockLoadT;
    typedef cub::BlockScan<T, BLOCK_THREADS>        BlockScanT;
    typedef cub::BlockStore<T*, BLOCK_THREADS, 4>   BlockStoreT;

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

    // Cooperatively load a tile of items
    T items[4];
    BlockLoadT(temp_storage.load).Load(d_in, items);

    __syncthreads();  // Barrier for smem reuse

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

    __syncthreads();  // Barrier for smem reuse

    // Cooperatively store a tile of items
    BlockStoreT(temp_storage.load).Store(d_in, items);
}
```

```
int* d_in;  // = ...
int* d_out; // = ...

// Invoke kernel (GF110 Fermi)
ExampleKernel <128> <<<1, 128>>>(
    d_in,
    d_out);
```

Tuning with CUB primitives

```
template <int BLOCK_THREADS, int ITEMS_PER_THREAD, typename T>
__global__ void ExampleKernel(T* d_in, T* d_out)
{
    // Specialize for BLOCK_THREADS threads owning ITEMS_PER_THREAD integers each
    typedef cub::BlockLoad<T*, BLOCK_THREADS, ITEMS_PER_THREAD>     BlockLoadT;
    typedef cub::BlockScan<T, BLOCK_THREADS>                        BlockScanT;
    typedef cub::BlockStore<T*, BLOCK_THREADS, ITEMS_PER_THREAD>    BlockStoreT;

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

    // Cooperatively load a tile of items
    T items[ITEMS_PER_THREAD];
    BlockLoadT(temp_storage.load).Load(d_in, items);

    __syncthreads();  // Barrier for smem reuse

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

    __syncthreads();  // Barrier for smem reuse

    // Cooperatively store a tile of items
    BlockStoreT(temp_storage.load).Store(d_in, items);
}
```

```
int* d_in;  // = ...
int* d_out; // = ...

// Invoke kernel (GF110 Fermi)
ExampleKernel <128, 4> <<<1, 128>>>(
    d_in,
    d_out);
```

Tuning with CUB primitives

```
template <int BLOCK_THREADS, int ITEMS_PER_THREAD, BlockLoadAlgorithm LOAD_ALGO>
__global__ void ExampleKernel(T* d_in, T* d_out)
{
    // Specialize for BLOCK_THREADS threads owning ITEMS_PER_THREAD integers each
    typedef cub::BlockLoad<T*, BLOCK_THREADS, ITEMS_PER_THREAD, LOAD_ALGO> BlockLoadT;
    typedef cub::BlockScan<T, BLOCK_THREADS> BlockScanT;
    typedef cub::BlockStore<T*, BLOCK_THREADS, ITEMS_PER_THREAD> BlockStoreT;

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

    // Cooperatively load a tile of items
    T items[ITEMS_PER_THREAD];
    BlockLoadT(temp_storage.load).Load(d_in, items);

    __syncthreads();  // Barrier for smem reuse

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

    __syncthreads();  // Barrier for smem reuse

    // Cooperatively store a tile of items
    BlockStoreT(temp_storage.load).Store(d_in, items);
}
```

```
int* d_in;  // = ...
int* d_out; // = ...

// Invoke kernel (GF110 Fermi)
ExampleKernel <128, 4, BLOCK_LOAD_WARP_TRANSPOSE>
         <<<1, 128>>>(
     d_in,
     d_out);
```

Tuning with CUB primitives

```
template <int BLOCK_THREADS, int ITEMS_PER_THREAD, BlockLoadAlgorithm LOAD_ALGO,
    BlockScanAlgorithm SCAN_ALGO, typename T>
__global__ void ExampleKernel(T* d_in, T* d_out)
{
    // Specialize for BLOCK_THREADS threads owning ITEMS_PER_THREAD integers each
    typedef cub::BlockLoad<T*, BLOCK_THREADS, ITEMS_PER_THREAD, LOAD_ALGO> BlockLoadT;
    typedef cub::BlockScan<T, BLOCK_THREADS, SCAN_ALGO> BlockScanT;
    typedef cub::BlockStore<T*, BLOCK_THREADS, ITEMS_PER_THREAD> BlockStoreT;

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

    // Cooperatively load a tile of items
    T items[ITEMS_PER_THREAD];
    BlockLoadT(temp_storage.load).Load(d_in, items);

    __syncthreads();  // Barrier for smem reuse

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

    __syncthreads();  // Barrier for smem reuse

    // Cooperatively store a tile of items
    BlockStoreT(temp_storage.load).Store(d_in, items);
}
```

```
int* d_in;  // = ...
int* d_out; // = ...

// Invoke kernel (GF110 Fermi)
ExampleKernel <128, 4, BLOCK_LOAD_WARP_TRANSPOSE,
        BLOCK_SCAN_RAKING> <<<1, 128>>>(
    d_in,
    d_out);
```

Tuning with CUB primitives

```
template <int BLOCK_THREADS, int ITEMS_PER_THREAD, BlockLoadAlgorithm LOAD_ALGO,
    CacheLoadModifier LOAD_MODIFIER, BlockScanAlgorithm SCAN_ALGO, typename T>
__global__ void ExampleKernel(T* d_in, T* d_out)
{
    // Specialize for BLOCK_THREADS threads owning ITEMS_PER_THREAD integers each
    typedef cub::BlockLoad<T*, BLOCK_THREADS, ITEMS_PER_THREAD, LOAD_ALGO> BlockLoadT;
    typedef cub::BlockScan<T, BLOCK_THREADS> BlockScanT;
    typedef cub::BlockStore<T*, BLOCK_THREADS, ITEMS_PER_THREAD> BlockStoreT;

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

    // Cooperatively load a tile of items
    T items[ITEMS_PER_THREAD];
    typedef cub::CacheModifiedInputIterator<LOAD_MODIFIER, T> InputItr;
    BlockLoadT(temp_storage.load).Load(InputItr(d_in), items);

   __syncthreads();  // Barrier for smem reuse

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

    __syncthreads();  // Barrier for smem reuse

    // Cooperatively store a tile of items
    BlockStoreT(temp_storage.load).Store(d_in, items);
}
```

```
int* d_in;  // = ...
int* d_out; // = ...

// Invoke kernel (GF110 Fermi)
ExampleKernel <128, 4, BLOCK_LOAD_WARP_TRANSPOSE,
        LOAD_DEFAULT, BLOCK_SCAN_RAKING> <<<1, 128>>>(
    d_in,
    d_out);
```

Tuning with CUB primitives

```
template <int BLOCK_THREADS, int ITEMS_PER_THREAD, BlockLoadAlgorithm LOAD_ALGO,
    CacheLoadModifier LOAD_MODIFIER, BlockScanAlgorithm SCAN_ALGO, typename T>
__global__ void ExampleKernel(T* d_in, T* d_out)
{
    // Specialize for BLOCK_THREADS threads owning ITEMS_PER_THREAD integers each
    typedef cub::BlockLoad<T*, BLOCK_THREADS, ITEMS_PER_THREAD, LOAD_ALGO> BlockLoadT;
    typedef cub::BlockScan<T, BLOCK_THREADS> BlockScanT;
    typedef cub::BlockStore<T*, BLOCK_THREADS, ITEMS_PER_THREAD> BlockStoreT;

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

    // Cooperatively load a tile of items
    T items[ITEMS_PER_THREAD];
    typedef cub::CacheModifiedInputIterator<LOAD_MODIFIER, T> InputItr;
    BlockLoadT(temp_storage.load).Load(InputItr(d_in), items);

   __syncthreads();  // Barrier for smem reuse

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

    __syncthreads();  // Barrier for smem reuse

    // Cooperatively store a tile of items
    BlockStoreT(temp_storage.load).Store(d_in, items);
}
```

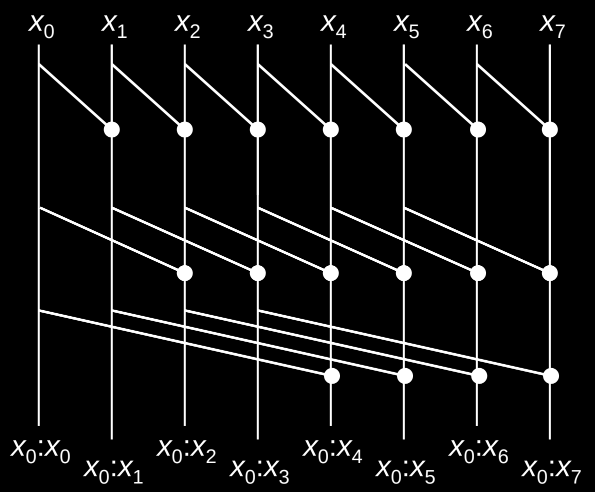
```
int* d_in;  // = ...
int* d_out; // = ...

// Invoke kernel (GK110 Kepler)
ExampleKernel <128, 21, BLOCK_LOAD_DIRECT,
        LOAD_LDG, BLOCK_SCAN_WARP_SCANS> <<<1, 128>>>(
    d_in,
    d_out);
```

Outline

1. Software reuse
2. SIMT collectives: the “missing” CUDA abstraction layer
3. The soul of collective component design
4. Using CUB’s collective primitives
5. Making your own collective primitives
6. Other Very Useful Things in CUB
7. Final thoughts

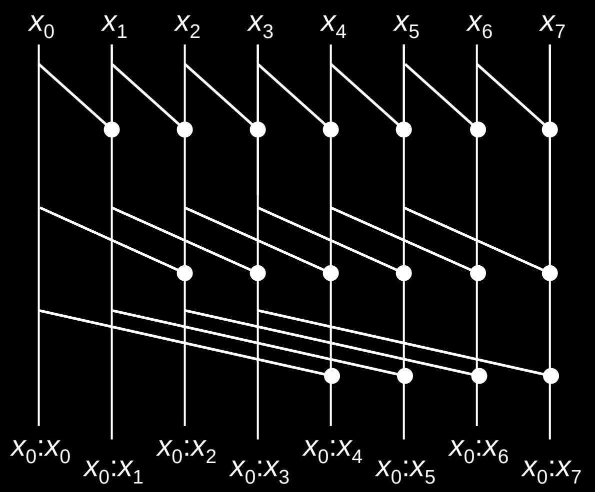
Block-wide prefix sum (simplified)

```
// Simple collective primitive for block-wide prefix sum
template <typename T, int BLOCK_THREADS>
class BlockScan
{









};
```



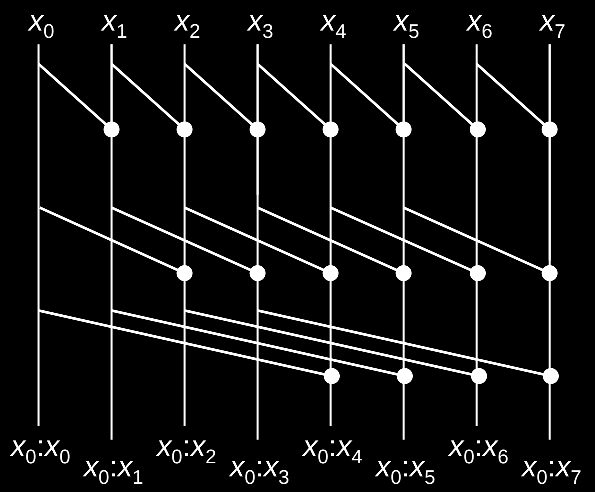
Block-wide prefix sum (simplified)

```
// Simple collective primitive for block-wide prefix sum
template <typename T, int BLOCK_THREADS>
class BlockScan
{
    // Type of shared memory needed by BlockScan
    typedef T TempStorage[BLOCK_THREADS];











};
```



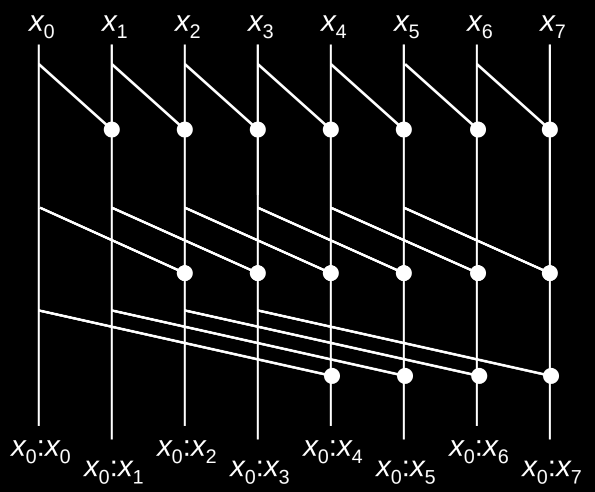
Block-wide prefix sum (simplified)

```
// Simple collective primitive for 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 (reference to shared storage)
    TempStorage &temp_storage;

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










};
```



Block-wide prefix sum (simplified)

```
// Simple collective primitive for 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 (reference to shared storage)
    TempStorage &temp_storage;

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

    // Inclusive prefix sum operation (each thread contributes its own data item)
    T InclusiveSum (T thread_data)
    {
        #pragma unroll
        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;
    }
};
```



Block-wide reduce-by-key (simplified)

```
// Reduce-by-segment scan data type
struct ValueOffsetPair
{
    ValueT  value;
    int     offset;

    // Sum operation
    ValueOffsetPair operator+(ValueOffsetPair &other)
    {
        ValueOffsetPair retval;
        retval.offset = offset + other.offset;
        retval.value = (other.offset) ?
            other.value :
            value + other.value;
        return retval;
    }
};
```

keys	*a*	*a*	*b*	*b*	*c*	*c*	*c*	*c*	
prev-keys	-	*a*	*a*	*b*	*b*	*c*	*c*	*c*	*c*
head-flags	*0*	*0*	*1*	*0*	*1*	*0*	*0*	*0*	*1*
values	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*	
scanned-flags	*0*	*0*	*0*	*1*	*1*	*2*	*2*	*2*	*2*
scanned-values	*0*	*1.0*	*2.0*	*1.0*	*2.0*	*1.0*	*2.0*	*3.0*	*4.0*

Block-wide reduce-by-key (simplified)

```
// Block-wide reduce-by-key
template <typename KeyT, typename ValueT, int BLOCK_THREADS, int ITEMS_PER_THREAD>
struct BlockReduceByKey
{
    // Parameterized BlockDiscontinuity type for keys
    typedef BlockDiscontinuity<KeyT, BLOCK_THREADS> BlockDiscontinuityT;

    // Parameterized BlockScan type
    typedef BlockScan<ValueOffsetPair, BLOCK_THREADS> BlockScanT;

    // Temporary storage type
    union TempStorage
    {
        typename BlockDiscontinuityT::TempStorage discontinuity;
        typename BlockDiscontinuityT::TempStorage scan;
    };

    // Reduce segments using addition operator.
    // Returns the "carry-out" of the last segment
    ValueT Sum(
      TempStorage& temp_storage,                        // shared storage reference
      KeyT         keys[ITEMS_PER_THREAD],              // [in|out] keys
      ValueT       values[ITEMS_PER_THREAD],            // [in|out] values
      int          segment_indices[ITEMS_PER_THREAD])   // [out] segment indices (-1 if invalid)
    {
       ...
    }
};
```

Block-wide reduce-by-key (simplified)

```
// Reduce segments using addition operator.
// Returns the "carry-out" of the last segment
ValueT Sum(
    TempStorage&    temp_storage,
    KeyT            keys[ITEMS_PER_THREAD],
    ValueT          values[ITEMS_PER_THREAD],
    int             segment_indices[ITEMS_PER_THREAD])
{
    KeyT            prev_keys[ITEMS_PER_THREAD];
    ValueOffsetPair scan_items[ITEMS_PER_THREAD];

    // Set head segment_flags.
    BlockDiscontinuityKeysT(temp_storage.discontinuity).FlagHeads(
      segment_indices, keys, prev_keys);

    __syncthreads();

    // Unset the flag for the first item
    if (threadIdx.x == 0)
        segment_indices[0] = 0;

    // Zip values and segment_flags
    for (int ITEM = 0; ITEM < ITEMS_PER_THREAD; ++ITEM)
    {
        scan_items[ITEM].offset = segment_indices[ITEM];
        scan_items[ITEM].value = values[ITEM];
    }

                          ...
```

```
                          ...

    // Exclusive scan of values and segment_flags
    ValueOffsetPair tile_aggregate;
    BlockScanT(temp_storage.scan).ExclusiveSum(
        scan_items, scan_items, tile_aggregate);

    // Unzip values and segment indices
    for (int ITEM = 0; ITEM < ITEMS_PER_THREAD; ++ITEM)
    {
        segment_indices[ITEM] = segment_indices[ITEM] ?
            scan_items[ITEM].offset :
            -1;
        keys[ITEM] = prev_keys[ITEM];
        values[ITEM] = scan_items[ITEM].value;
    }

    // Return "carry-out"
    return tile_aggregate.value;
}
```

Outline

1. Software reuse
2. SIMT collectives: the “missing” CUDA abstraction layer
3. The soul of collective component design
4. Using CUB’s collective primitives
5. Making your own collective primitives
6. Other Very Useful Things in CUB
7. Final thoughts

Cache-modified input iterators

```
#include <cub/cub.cuh>

// Standard layout type
struct Foo
{
    double x;
    char y;
};

__global__ void Kernel(Foo* d_in, Foo* d_out)
{
    // In host or device code: create an LDG wrapper
    cub::CacheModifiedInputIterator<cub::LOAD_LDG, Foo> ldg_itr(d_in);
    cub::CacheModifiedOutputIterator<cub::STORE_WT, Foo> volatile_itr(d_out);

    volatile_itr[threadIdx.x] = ldg_itr[threadIdx.x];
}
```

```
code for sm_35
        Function : _Z6KernelPdS_
MOV R1, c[0x0][0x44];
S2R R0, SR_TID.X;
ISCADD R2, R0, c[0x0][0x140], 0x3;
LDG.64 R4, [R2];
LDG.64 R2, [R6];
ISCADD R0, R0, c[0x0][0x144], 0x4;
TEXDEPBAR 0x1;
ST.WT.64 [R0], R4;
TEXDEPBAR 0x0;
ST.WT.64 [R0+0x8], R2;
EXIT;
```

```
LOAD_DEFAULT,       ///< Default (no modifier)
LOAD_CA,            ///< Cache at all levels
LOAD_CG,            ///< Cache at global level
LOAD_CS,            ///< Cache streaming (likely to be accessed once)
LOAD_CV,            ///< Cache as volatile (including cached system lines)
LOAD_LDG,           ///< Cache as texture
LOAD_VOLATILE,      ///< Volatile (any memory space)
```

Texture obj (and ref) input iterators

```
#include <cub/cub.cuh>

// Standard layout type
struct Foo
{
    int     y;
    double  x;
};

template <typename InputIteratorT, typename OutputIteratorT>
__global__ void Kernel(InputIteratorT d_in, OutputIteratorT d_out)
{
    d_out[threadIdx.x] = d_in[threadIdx.x];
}


// Create a texture object input iterator
Foo* d_foo;
cub::TexObjInputIterator<Foo> d_foo_tex;
d_foo_tex.BindTexture(d_foo);

Kernel<<<1, 32>>>(d_foo_tex, d_foo);

d_foo_tex.UnbindTexture();
```

```
code for sm_35
Function :
_Z6KernelIN3cub19TexObjInputIteratorI3FooiEEPS
2_EvT_T0_

MOV R1, c[0x0][0x44];
S2R R0, SR_TID.X;
IADD R2, R0, c[0x0][0x144];
SHF.L R2, RZ, 0x1, R2;
IADD R3, R2, 0x1;
TLD.LZ.T R2, R2, 0x52, 1D, 0x1;
TLD.LZ.P R4, R3, 0x52, 1D, 0x3;
ISCADD R0, R0, c[0x0][0x150], 0x4;
TEXDEPBAR 0x1;
ST [R0], R2;
TEXDEPBAR 0x0;
ST.64 [R0+0x8], R4;
EXIT;
```

Collective primitives

- `WarpReduce`
  - reduction & segmented reduction
- `WarpScan`

- `BlockDiscontinuity`
- `BlockExchange`
- `BlockHistogram`
- `BlockLoad & BlockStore`
- `BlockRadixSort`
- `BlockReduce`
- `BlockScan`

Device-wide (global) primitives

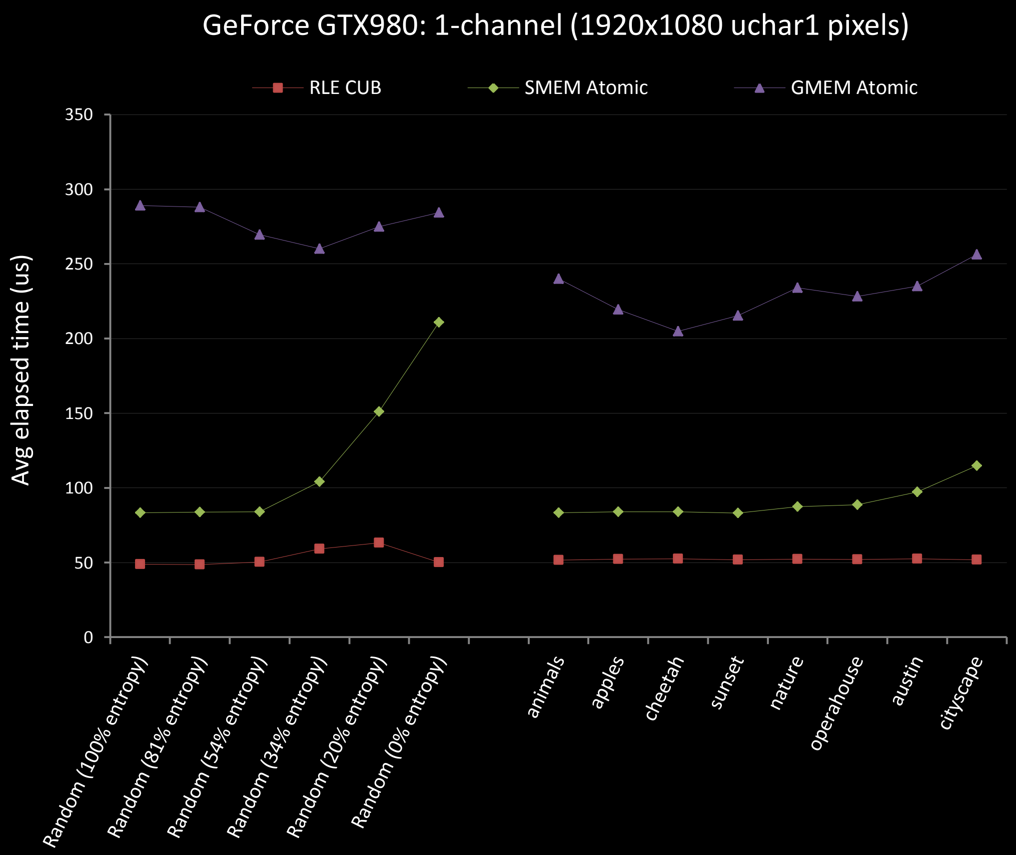
(Usable with CDP, streams, and your own memory allocator)

- `DeviceHistogram`
  - histogram-even
  - histogram-range
- `DevicePartition`
  - partition-if
  - partition-flagged
- `DeviceRadixSort`
  - ascending / descending
- `DeviceReduce`
  - reduction
  - arg-min, arg-max
  - reduce-by-key
- `DeviceRunLengthEncode`
  - RLE
  - Non-trivial segments
- `DeviceScan`
  - inclusive / exclusive
- `DeviceSelect`
  - select-flagged
  - select-if
  - keep-unique
- `DeviceSpmv`

NEW: performance-resilient histogram

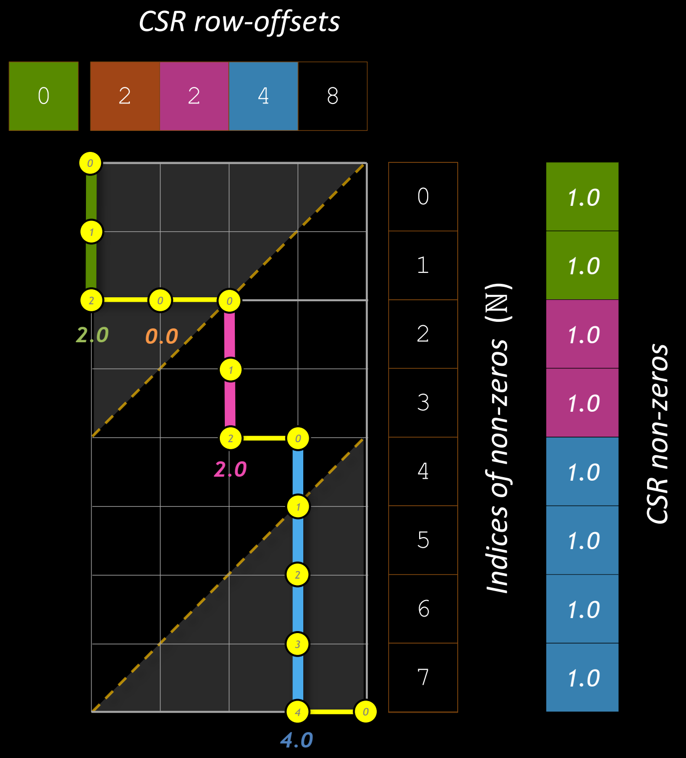
Simple intra-thread RLE provides a uniform performance response regardless of input sample distribution

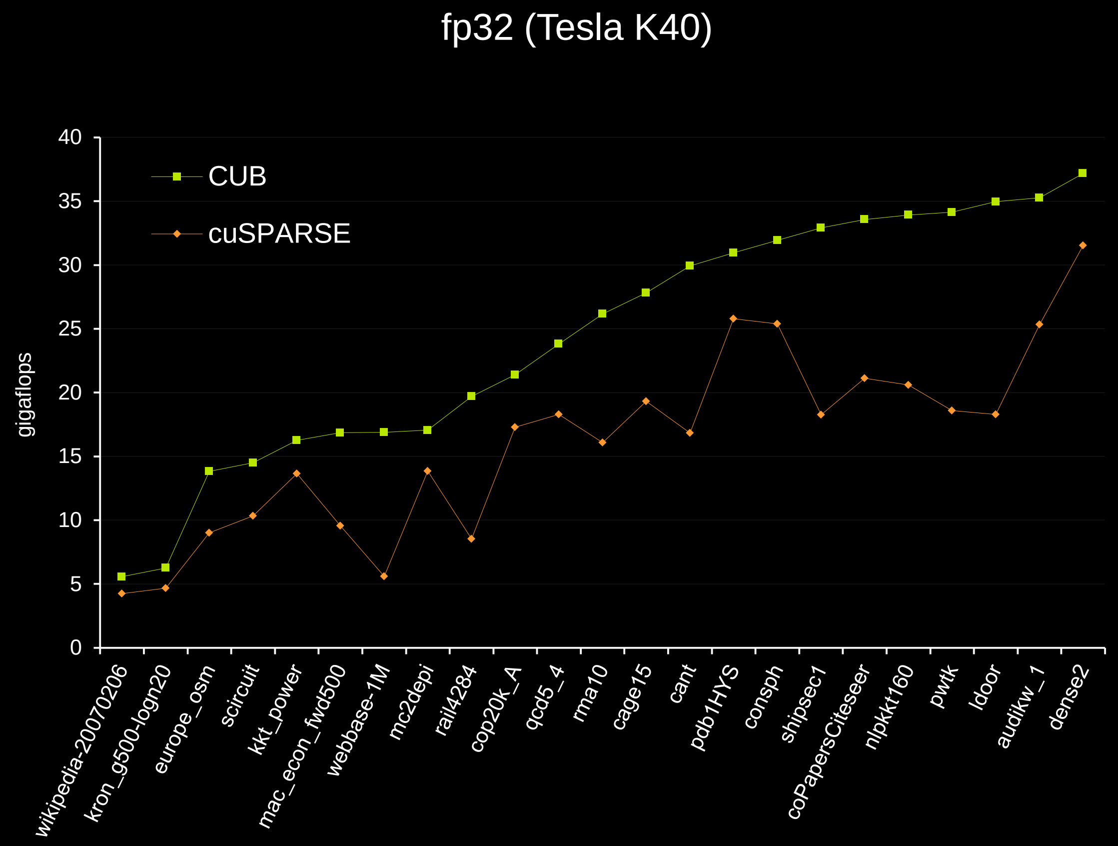
```
// RLE pixel counts within the thread's pixels
int accumulator = 1;
for (int PIXEL = 0;
     PIXEL < PIXELS_PER_THREAD - 1;
     ++PIXEL)
{
    if (bins[PIXEL] == bins[PIXEL + 1])
    {
         accumulator++;
    }
    else
    {
         atomicAdd(
             privatized_histogram + bins[PIXEL],
             accumulator);
         accumulator = 1;
    }
}
```



NEW: CSR SpMV

Merge-based parallel decomposition for load balance





Outline

1. Software reuse
2. SIMT collectives: the “missing” CUDA abstraction layer
3. The soul of collective component design
4. Using CUB’s collective primitives
5. Making your own collective primitives
6. Other Very Useful Things in CUB
7. Final thoughts

Benefits of using CUB primitives

- **Simplicity of composition**
  - Kernels are simply sequences of primitives
- **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?

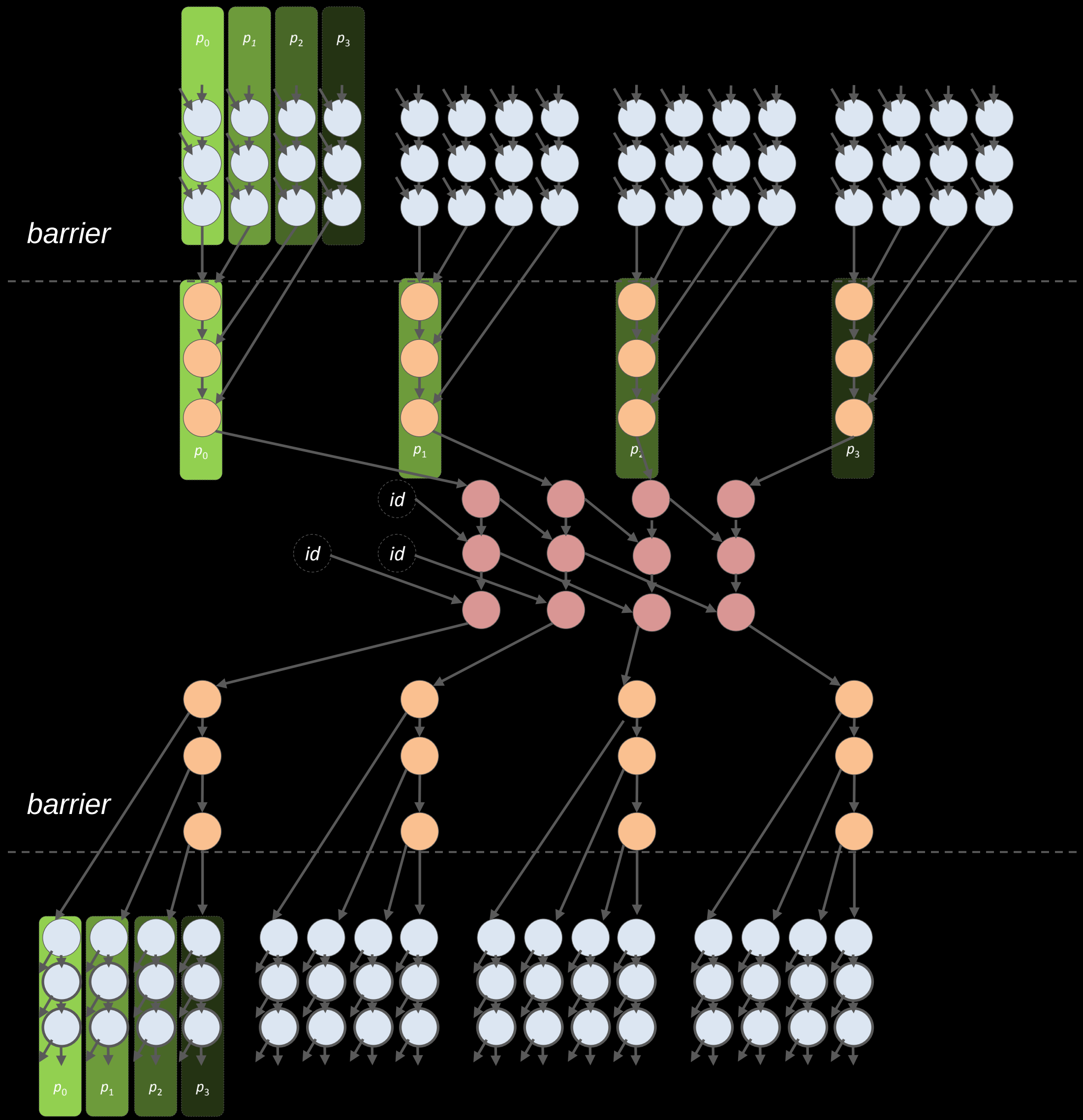
Please visit the **CUB** project on GitHub
http://nvlabs.github.com/cub

Duane Merrill (dumerrill@nvidia.com)

THANK YOU

JOIN THE CONVERSATION

#GTC15


p_0 p_1 p_2 p_3
barrier
p_0 p_1 p_2 p_3
id
id
id
barrier
p_0 p_1 p_2 p_3