






CUB :

Kernel-level software reuse and library design



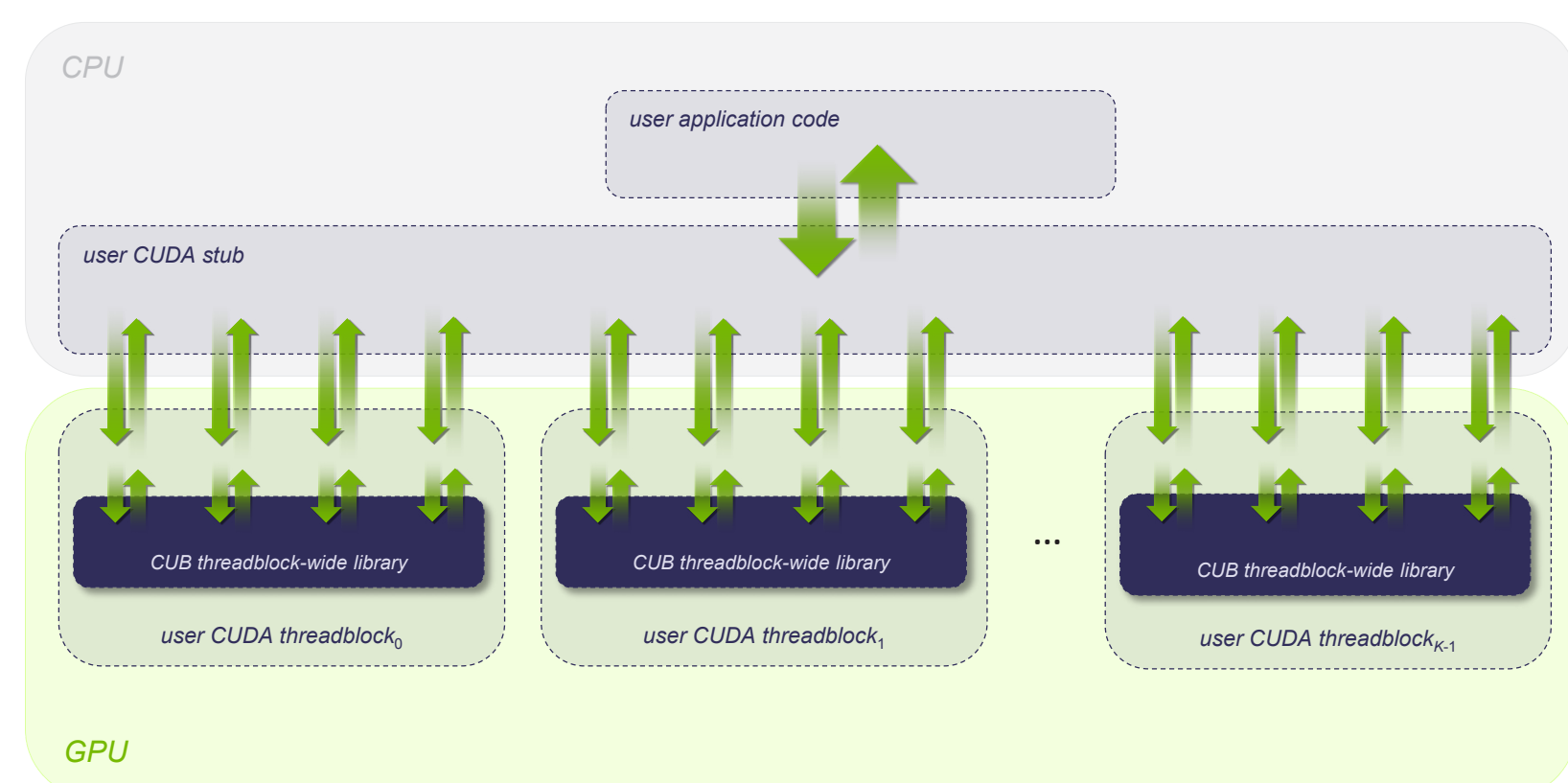
 <http://nvlabs.github.com/cub>
 <https://github.com/NVlabs/cub>
 <http://groups.google.com/group/cub-users>



Duane Merrill , NVIDIA Research
<dumerrill@nvidia.com>

What is CUB?

CUB is a library of high-performance parallel primitives and other utilities for constructing CUDA kernel software.



Building and fine-tuning kernel code is perhaps the most challenging, time-consuming aspect of CUDA programming. The goal of CUB is to provide abstractions for complex block-level, warp-level, and thread-level operations.

CUB's primitives are not bound to any particular width of parallelism or to any particular data type. This allows them to be flexible and tunable to fit your kernels' needs. Thus CUB is *CUDA Unbound*.

CUB includes collections of:

- **Cooperative primitives**, including:
 - Thread block operations (e.g., radix sort, prefix scan, reduction, etc.)
 - Warp operations (e.g., prefix scan)
- **SIMT utilities**, including:
 - Tile-based I/O utilities (e.g., for performing coalesced and/or vectorized data movement of data tiles)
 - Low-level thread I/O using cache-modifiers
 - Abstractions for thread block work distribution (e.g., work-stealing, even-share, etc.)
- **Host utilities**, including:
 - Caching allocator for quick management of device temporaries
 - Device reflection

A simple example

The following code snippet illustrates a simple CUDA kernel for sorting a thread block's data:

```
#include <cub.cuh>

// An tile-sorting CUDA kernel
template <
    int BLOCK_THREADS, // Threads per block
    int ITEMS_PER_THREAD, // Items per thread
    typename T> // Numeric data type
__global__ void TileSortKernel(T *d_in, T *d_out)
{
    using namespace cub;
    const int TILE_SIZE = BLOCK_THREADS * ITEMS_PER_THREAD;

    // Parameterize cub::BlockRadixSort for this context
    typedef BlockRadixSort<T, BLOCK_THREADS> BlockRadixSort;

    // Declare the shared memory needed by BlockRadixSort
    __shared__ typename BlockRadixSort::SmemStorage smem_storage;

    // A segment of data items per thread
    T data[ITEMS_PER_THREAD];

    // Load a tile of data using vector-load instructions
    BlockLoadVectorized(data, d_in + (blockIdx.x * TILE_SIZE));

    // Sort data in ascending order
    BlockRadixSort::SortBlocked(smem_storage, data);

    // Store the sorted tile using vector-store instructions
    BlockStoreVectorized(data, d_out + (blockIdx.x * TILE_SIZE));
}
```

The `cub::BlockRadixSort` type performs a cooperative radix sort across the thread block's data items. Its implementation is parameterized by `BLOCK_THREADS` and the key type `T`, and is opaquely specialized for the compilation architecture.

Once instantiated, the `cub::BlockRadixSort` type exposes an opaque `cub::BlockRadixSort::SmemStorage` member type. The thread block uses this storage type to allocate the shared memory needed by the primitive. This storage type can be aliased or union'd with other types so that the shared memory can be reused for other purposes.

Furthermore, the kernel uses CUB's primitives for vectorizing global loads and stores. For example, lower-level `ld.global.v4.s32` PTX instructions will be generated when `T = int` and `ITEMS_PER_THREAD` is a multiple of 4.

How does CUB work?

C++ Templates. A cooperative SIMT library must accommodate a wide spectrum of parallel contexts, i.e., specific:

- Data types
- Widths of parallelism (threads per block)
- Grain sizes (data items per thread)
- Underlying architectures (special instructions, warp size, rules for bank conflicts, etc.)
- Tuning requirements (e.g., latency vs. throughput)

To provide this flexibility, CUB is implemented as a C++ template library. There is no need to build CUB separately. You simply `#include` the `cub.cuh` header file into your `.cu` CUDA C++ sources and compile with NVIDIA's `nvcc` compiler.

Reflective Type Structure. Cooperation within a thread block requires shared memory. However, the specific size and layout of shared memory needed will be specific to how many threads are calling into it, how many items are processed per thread, etc. Furthermore, this shared memory must be allocated outside of the component if it is to be reused elsewhere by the thread block.

```
// Parameterize a BlockScan type for use with 128 threads
// and 4 items per thread
typedef cub::BlockScan<unsigned int, 128, 4> BlockScan;

// Declare shared memory for BlockScan
__shared__ typename BlockScan::SmemStorage smem_storage;

// A segment of consecutive input items per thread
int data[4];

// Obtain data in blocked order
...

// Perform an exclusive prefix sum across the tile of data
BlockScan::ExclusiveSum(smem_storage, data, data);
```

To address these issues, we encapsulate cooperative procedures within reflective type structure (C++ classes). As illustrated in the `cub::BlockScan` example above, these primitives are C++ classes with interfaces that expose both:

- Procedural entrypoints for a block of threads to invoke
- An opaque shared memory type needed by those entrypoints

Why do you need CUB?

Constructing and fine-tuning kernel code is perhaps the most challenging, time-consuming aspect of CUDA programming. Programmers must reason about deadlock, livelock, synchronization, race conditions, shared memory layout, plurality of state, granularity, throughput, latency, memory bottlenecks, etc.

However, with the exception of CUB, there are few (if any) software libraries of reusable kernel primitives. In the CUDA ecosystem, CUB is unique in this regard. As a SIMT library and software abstraction layer, CUB provides:

- **Simplicity of composition.** Parallel CUB primitives can be simply sequenced together in kernel code (similarly to programming with Thrust primitives on the host)
- **High performance.** We're taking care to provide CUB with the fastest available algorithms, strategies, techniques, etc.
- **Performance portability.** CUB primitives are specialized to match the target hardware. Furthermore, CUB serves a central point of evolution for accommodating new algorithmic developments, hardware instructions, etc.

- **Simplicity of performance tuning.** CUB primitives provide parallel abstractions whose behavior can be statically tuned. For example, most CUB primitives support alternative algorithmic strategies and variable grain sizes (threads per block, items per thread, etc.).
- **Robustness and durability.** We've designed CUB primitives to function properly for arbitrary data types and widths of parallelism (not just for the built-in C++ types or for powers-of-two threads per block).