cuda-kat: CUDA Kernel Author's Toolkit

An install-less, header-only library which is a loosely-coupled collection of utility functions and classes for writing device-side CUDA code (kernels and non-kernel functions). These let us:

- Write templated device-side without constantly coming up against not-trivially-templatable bits.
- Use standard-library(-like) containers in device-side code (but not have to use them).
- Not repeat ourselves as much (the DRY principle).
- Use less magic numbers.
- Make our device-side code less cryptic and idiosyncratic, with clearer naming and semantics.

... while not committing to any particular framework, paradigm or class hierarchy - and not compromising performance.

Table of contents

- Tools in the box
- Motivating example: The patterns that repeat!
- Questions? Suggestions? Found a bug?

The tools in the box

The library has Doxygen documentation, available here. However - it is far from being complete.

An alternative place to start is the table below. Since cuda-kat’s different facilities mostly correspond to different files (and not many such files), the table The various utility functions (and occasional other constructs) available in the cuda-kat package are accessed through the files in the following table. Each of these may be used independently of the others (although internally there are some dependencies); and no file imposes any particular abstraction or constraint on the rest of your code.

<table>
<thead>
<tr>
<th>Header(s)</th>
<th>Examples and description</th>
</tr>
</thead>
<tbody>
<tr>
<td>grid_info.cuh</td>
<td>lane::id()</td>
</tr>
<tr>
<td></td>
<td>block::num_warps()</td>
</tr>
<tr>
<td></td>
<td>Shorthands/mnemonics for information about positions and sizes of lanes, threads, warps and blocks within warps, blocks and grids (particularly for one-</td>
</tr>
<tr>
<td>Header(s)</td>
<td>Examples and description</td>
</tr>
<tr>
<td>--------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ptx.cuh</td>
<td>C++ bindings for PTX instructions which are not made available by CUDA itself. (Templatized versions of some of these appear in <code>builtins.cuh</code>.)</td>
</tr>
<tr>
<td>shared_memory.cuh</td>
<td>Access and size-determination gadgets for templated dynamic shared memory (which is not directly possible in CUDA!).</td>
</tr>
<tr>
<td>atomics.cuh</td>
<td>Uniform, templated, zero-overhead invocation of all CUDA-supported atomic operations; and compare-and-swap-based implementation for operations without atomic primitives on your GPU.</td>
</tr>
<tr>
<td>builtins.cuh</td>
<td>Uniform, templated, zero-overhead, non-cryptically-named invocation of all (non-atomic) operations involving a single PTX instructions (which aren't C++ language builtins like + or -).</td>
</tr>
<tr>
<td>shuffle.cuh</td>
<td>Exchange arbitrary-type (and arbitrarily large) values using intra-warp shuffles.</td>
</tr>
<tr>
<td>math.cuh</td>
<td>Templated mathematical functions - some <code>constexpr</code>, some using special GPU capabilities which cannot be executed at compile time even when data is available, and more efficient runtime implementation of compile-time-executable functions.</td>
</tr>
<tr>
<td>constexpr_math.cuh</td>
<td>The part of <code>math.cuh</code> which can be <code>constexpr</code>-evaluated, i.e. executed at compile-time (but with <code>__host__ __device__</code> for good measure).</td>
</tr>
<tr>
<td>streams/*.cuh</td>
<td>Example code: <code>stringstream ss; ss &lt;&lt; &quot;easy as &quot; &lt;&lt; 123 &lt;&lt; &quot;; use_a_string(ss.c_str()); printfing_ostream cout; cout &lt;&lt; &quot;This text and number (&quot; &lt;&lt; 123.45 &lt;&lt; &quot;) got printfed on</code></td>
</tr>
<tr>
<td>Header(s)</td>
<td>Examples and description</td>
</tr>
<tr>
<td>------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>endl/flush</code> &lt;&lt; flush;</td>
<td>Stream functionality using dynamic memory allocation and/or CUDA's device-side <code>printf()</code>. Also supports automatic self-identification of the printer thread.</td>
</tr>
<tr>
<td><code>read_unaligned_safe(const uint32_t* ptr)</code></td>
<td>Read access to multi-byte values at non-naturally-aligned addresses (the GPU hardware cannot directly read these). Caveat: UNTESTED.</td>
</tr>
<tr>
<td><code>block::collaborative::fill(data_start, data_end, 0xDEADBEEF)</code></td>
<td>Grid, block and warp-level collaborative primitives of acti. Examples: Coalescing-friendly Visitation/traversal of a large index sequence; balloting, complex balloting operations, block-level broadcast, multisearch and more.</td>
</tr>
<tr>
<td><code>warp::collaborative&gt;(plus, result_ptr, data, )</code></td>
<td>Other grid, block and warp-level collaborative primitives (i.e. ones in which the threads in a grid, block, or warp have a common generic task to perform rather than per-thread task). Examples: Coalescing-friendly Visitation/traversal of a large index sequence; balloting, complex balloting operations, block-level broadcast, multisearch and more.</td>
</tr>
<tr>
<td><code>swap(T&amp; x, T&amp;y)</code></td>
<td>Implementations of (almost) all functions in the C standard library's <code>&lt;string.h&gt;</code> header, to be executed at the single-thread level. This is neither quick nor efficient, but, you know, just in case you happen to need it.</td>
</tr>
<tr>
<td><code>swap(T&amp; x, T&amp;y)</code></td>
<td>Functionality not otherwise categorized, but of sufficient worth to have available.</td>
</tr>
</tbody>
</table>

For further details you can either explore the files themselves or read the Doxygen-generated [official documentation](#). Note The toolbox is not complete (especially the bottom items in the listing above), so expect some additions of files, and within files.

**Motivating example: The patterns that repeat!**

To illustrate the motivation for `cuda-kat` and the spirit in which it's written, let's focus on one kind of functionality: Referring to information about blocks, threads, warps, lanes and grids.

**You know you've written this before...**
As you write more CUDA kernels - for an application or a library - you likely begin to notice yourself repeating many small patterns of code. A popular one would be computing the global id of a thread in a linear grid:

```cpp
auto global_thread_id = threadIdx.x + blockIdx.x * blockDim.x;
```

we've all been using that again and again, right? Many of us find ourselves writing a small utility function for this pattern. Perhaps it's something like:

```cpp
auto global_thread_id() { return threadIdx.x + blockIdx.x * blockDim.x; }
```

In fact, some of those end up having written this same function several times, in several projects we've worked on. This is irksome, especially when you know that about every other CUDA library probably has this function, or uses this pattern, somewhere in its bowels. thrust does. cub does. cudf does. (moderngpu, weirdly, doesn't) and so on.

But here's the rub: When you do implement these convenience/utility functions, they're a part of something larger, which comes with its own assumptions, and dependencies, and abstractions. If you could only just share such small, low-level pieces of code among multiple projects without there being any "strings attached"...

Well, this is what cuda-kat is for.

**Say it with natural-language-like mnemonics**

The example of `global_thread_id()` is quite obvious in that it involves multiple operations and writing the function name is more terse than spelling out the computation. But naming this pattern is attractive just as much due to our giving of a **meaningful name** to the computation, regardless of its length. It's also an application of the **DRY principle**; and it removes numerous repetitive calculations, which divert attention from the more meaningful computational work you're actually carrying out in your kernel.

But let's make that more concrete. Consider the following statements:

```cpp
if (threadIdx.x == blockDim.x - 1) { do_stuff(); }
auto foo = threadIdx.x % warpSize;
auto bar = blockDim.x / warpSize;
```

when you write them, what you really mean to be saying is:

```cpp
if (this_thread_is_the_last_in_its_block()) { do_stuff(); }
auto foo = lane_id_within_warp();
auto bar = num_full_warps_in_grid_block();
```

and these, while longer to type, are clearer to the reader and less prone to typos.

... and put them all in order.

Instead of a flat collection of rather long-named functions:

```cpp
global_thread_id()
```
the library groups these (and many other related) functions into relevant namespaces. We thus have:

```cpp
lane_id_within_warp()
this_thread_is_the_last_in_its_block()
num_full_warps_in_grid_block()
```

which is easier to browse through if you use auto-complete. The order comes at the expense of brevity... but we can alleviate this with an appropriate namespace. The above can then become simply:

```cpp
namespace gi = kat::linear_grid::grid_info;
gi::thread::global_id()
gi::lane::id()
gi::thread::is_last_in_block()
gi::block::num_full_warps()
```

in your code. Now this is how I want to write my kernels!

You will note, that most similar phrases you could come up with about positions and sizes within the grid - already have implementations. For example: "I can get the number of full warps, but now I want the number of warps, period"; well, just replace `num_full_warps()` with `num_warps()` and it's there: `linear_grid::grid_info::block::num_warps()` is available.

And as a final bonus - if you write a non-linear kernel, with blocks and grids having y and z dimensions other than 1 - you will only need to change your namespace = or using statements, to be able to write the same code and use 3-D implementations of these functions instead.

Questions? Suggestions? Found a bug?

- Have a question? There's a FAQ on the library's Wiki; please check it out first.
- Found a bug? A function/feature that's missing? A poor choice of design or of wording?- Please file an issue.
- Have a question that's not in the FAQ? If you believe it's generally relevant, also file an issue, and clearly state that it's a question.
- Want to suggest significant additional functionality, which you believe would be of general interest? Either file an issue or write me about it.
- Pull Requests are welcome, but it's better to work things out by talking to me before you invest a lot of work in preparing a PR.