ProgramDescriptor & ProgramCache Proposal

Motivation

Currently **Program Cache** presents a major problem for TTNN OP writers. More specifically, it adds a lot of complexity and causes a lot of hard-to-catch bugs.
Because of existence of Program Cache, each OP must:

- Manually implement \`compute_program_hash\`, defining when the same program can be reused.
- Create a \`shared_variables_t\` struct within each program factory, and populate it upon creation of the program with data which must be preserved between op invocations.
- Manually implement \`override_runtime_arguments\` for each program factory, selectively updating runtime arguments when the program is being reused with different arguments.

In practice, this is one of the most problematic parts in writing any operation, as confirmed by numerous internal and external feedback. To give some examples, we're receiving a steady stream of issues related to the Program Cache in the OPs, just to list a few:
Sliding window hash calculation bug
Low PCC on reshape with program cache enabled
Fixing PCC and Program Cache issues for Repeat and Expand
The bugs in this area are so common that **we recommend disabling program cache as the first step of troubleshooting for external consumers** who observe a weird PCC issue.

Moreover, manually calculating program hash and updating runtime args correctly in all cases is quite hard and sometimes should be updated for changes in the OP code itself, which makes it even harder for the code to stay correct.

Additionally, this proposal coincidentally solves another problem - having a clean / simple / user-friendly API for Program and Kernel.

Proposed Solution

The main idea behind the proposal is that we should completely automate the Program Cache, so a user doesn't have to think about it, and just make it what it sounds like - a cache for programs, no more, no less.

The first step is to define a **ProgramDescriptor** - a full description of the Program from the user perspective. It is a transparent struct, containing information about kernels, compile-time / run-time arguments, defines, circular buffers, semaphores, etc. More specifically, you can take a look at exact definitions of the proposed structures here.
Those structures are fast to create, they don't perform any actions behind the scenes, and just store the required description of the Program while minimizing the extra allocations.

I suggest making a ProgramFactory to do exactly what it's supposed to do - describe the Program which it wants to create. This way ProgramFactory would have a single method `create` returning a ProgramDescriptor. No `shared_variables_t`, `cached_program_t`, `override_runtime_arguments`, etc.

I propose changing the TTNN program caching/creating flow in the following way:

1. We always call ProgramFactory::create to obtain the desired ProgramDescriptor
2. We automatically calculate the program hash based on it (for example like this)
3. On cache miss, we create a new Program based on the descriptor
4. On cache hit, we get the previously created Program, and update according to the new descriptor. This basically involves calling `memcpy` for runtime args (full implementation here)

This way we free the users from obligations to manually write code for program cache support, guarantee its correctness, and allow them to use a simple clearly-defined API.

Feasibility Checks

I believe we need to perform a few feasibility checks, to ensure that we can actually make this proposal work as expected. More specifically:

1. Ensure we can correctly automatically calculate program hash for all cases
2. Ensure we can automatically implement an analog of `override_runtime_arguments`
3. Check that ProgramDescriptor API is descriptive enough to represent OPs that we have
4. Ensure that the new API is comfortable to use for writing OPs
5. Get a feeling of the effort required to port OPs to the new APIs
6. Ensure that we don't sacrifice performance to achieve the goals

For points **1** and **2** (automatic hash and override_runtime_arguments), I have a draft, which implements a slightly different inefficient approach of creating a full Program each time, but automates hash calculation and overrides without any changes to the OPs. It passes all APC tests, proving that we indeed can automatically calculate the hash and update Programs with new arguments. Moreover, it actually results in more Program reuse and found another bug of an incorrect hashing in an OP.

For points **3-5**, I implemented the ProgramDescriptor approach in this draft, and ported Matmul operation (including all of its program factories) to descriptors in this draft. The API looks sufficient and easy to use. The porting process is straightforward and mostly just replacing one kind of calls with another, but will be quite involved, considering we have a lot of OPs and each one should be (eventually) ported. The good thing is that we can port OPs one by one, getting additional benefits with each OP ported.

Performance (point **6**) requires special attention, and will be considered in detail in its own separate section.

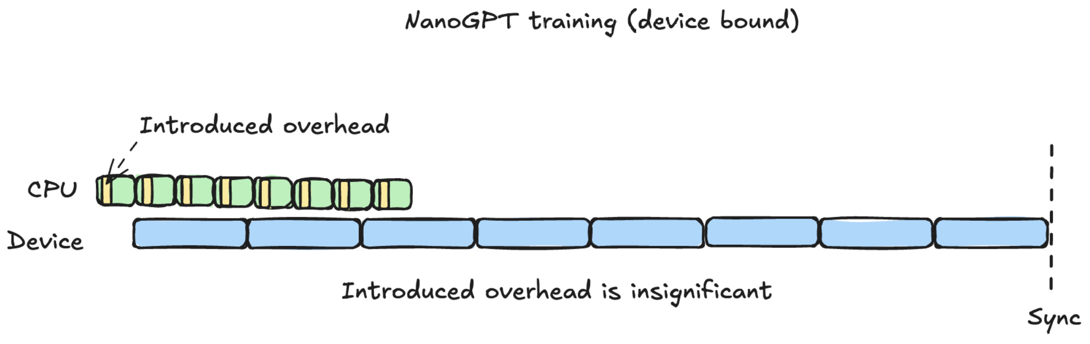
Performance Implications

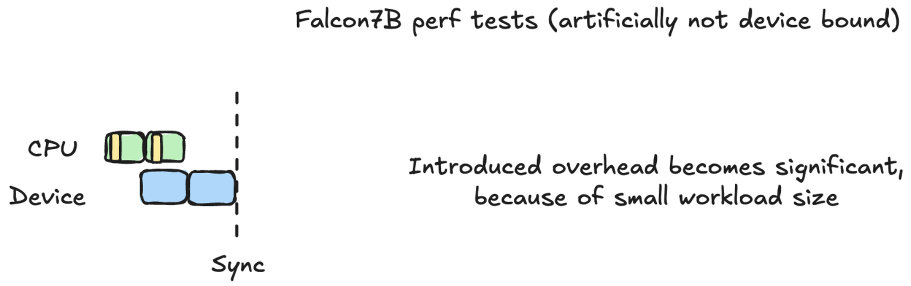
As you may expect, there are some performance implications related to the proposed solution. Based on the profiling of the mentioned above prototypes, we can conclude that calculating hash for ProgramDescriptor and updating runtime arguments using it is about as fast as with the original implementation.

The only meaningful overhead of the proposed solution is creating a ProgramDescriptor every time an operation gets called. The exact cost of this depends on the operation implementation itself. To get the rough idea, they typically do the following: calculate how to split the work onto cores, fill out circular buffer configs, kernel paths, compile time args, runtime args, each with a trivial computation. In practice it seems to consume something in the order of 10 microseconds of CPU time.

Let’s think about how this extra overhead influences the total performance of the program. I want to make the claim that it actually doesn't. Let’s assume that we’re running a reasonable model (for either training or inference). This compute should be clearly device bound, not CPU bound (otherwise something already went terribly wrong). This means that adding this extra CPU overhead would only a little increase the latency at the start of the computational workload, not affecting the perf during the actual computation. So, the introduced overhead should be diminished to zero assuming a reasonable workload size.

This analysis is consistent with the experimental data. I ran training with nanogpt which does not show any slowdown from the proposed solution. However, running T3K model perf tests shows a different picture. There Falcon 40B tests succeed, but Falcon 7B tests fail showing perf decrease. Those tests run a single iteration (which takes 0.0 seconds) and then call `device_synchronize` after it, resulting in the initial extra latency to be significant. I don’t think this is representative of any practical use-case.


NanoGPT training (device bound)
Introduced overhead
CPU
Device
Introduced overhead is insignificant
Sync


Falcon7B perf tests (artificially not device bound)
CPU
Device
Sync
Introduced overhead becomes significant, because of small workload size

Hybrid solution

As you can see from the performance section, performance is quite tricky and the overhead actually depends on the specific OP implementation of the `create` function. To ensure that we can always get the optimal performance, we can do a hybrid solution. Allow for some OPs who have especially heavy `create` implementations to define a custom faster implementation for program hash and custom `override_runtime_arguments` which would take ProgramDescriptor and update it. This would guarantee the same level of performance. Moreover, we can provide automatic validation for the correctness of those hash computation in debug mode by comparing the results with the automatic implementation.