CUDA Graph Multi-Streaming in XLA

Summary
A strategy for building CUDA graphs with multi-streaming support

Summary

CUDA Graphs support “multi-streaming” by allowing building a DAG of operations that can run concurrently. Strictly speaking it’s not multi-streaming because CUDA graphs are always launched on a single stream, but to construct concurrent CUDA graphs we use multiple streams in a graph capture mode (see example). CUDA runtime decides on what stream to run graph nodes, and guarantees that all effects will be properly ordered to be consumed on the “main” stream that the graph was launched on.

This document is a high level overview of the XLA runtime support to capture CUDA graphs that can exploit concurrency available in the XLA programs.

Input IR properties

When XLA runtime gets input IR it has few important properties that allows us to use relatively simple stream assignment strategy:

1. All buffer arguments in XLA do not alias. If we have two memref.view operations with different source buffer arguments, it’s guaranteed that they do not alias. We can use simple offset + size buffer aliasing to detect aliased buffers constructed from the same argument
2. XLA operations do not have any side-effects beyond writing to its buffer arguments. There is no side channel between different kernel launches or library calls. We can safely reorder operations if they do not have write conflicts.
3. We have information about read-only and read-write buffer arguments.

Stream Assignment Strategy

```
func @xla.cuda.graph.capture(...) {
  // (1)
  call @xla.cuda.graph.begin_concurrent_region(num_streams = 5)

  // (2)
  call @xla.kernel.launch @foo(%A, =>%B) { stream = 0 }
  call @xla.kernel.launch @bar(%C, =>%D) { stream = 1 }
```

```
  // (3)
  call @xla.kernel.launch @baz(%B, =>%X) { stream = 0 }

  // (4)
  call @xla.stream.await { from = 0, to = 1 }
  call @xla.kernel.launch @qux(%B, %D, =>%Y) { stream = 1 }

  // (5)
  call @xla.stream.await { from = ..., to = 0}
  call @xla.fft()

  // (6)
  call @xla.cuda.graph.end_concurrent_region()
  return
}
```

(1): Start Concurrent Region

Whole CUDA graph capture function will be executed within the concurrent region with a statically known number of threads. For example for num_threads = 5, we'll borrow 5 extra streams from the stream pool, in addition to the "main" compute stream. The number of streams should be configurable, because borrowing streams is relatively expensive, and we do not want to pay the extra cost for tiny cuda graphs. Once you start the concurrent region, all additional streams are synchronized with the capture stream, and implicitly switched to a capture mode.

(2): Explicit stream assignment for supported ops

Today multi-streaming is implicit at run time and streams assigned in a round-robin fashion. We should make stream assignment explicit in the IR, this requires adding new "stream aware" operations (or alternatively adding default values attribute to all existing operations). This will simplify testing of stream assignment strategy.

In this example two kernels can be launched on two different streams, because they do not have any write conflicts.

(3): Stream assignment follows dependencies

If operation depends on a written-to buffer, it should be scheduled on the same stream where the result was produced. In this example @baz should run on stream 0, because it's where %B was written to.

(4): Explicit stream synchronization if needed

If operation depends on multiple buffers written-to on multiple stream, it should be assigned to a stream with largest number of submitted operations (no need to block a stream with smaller amount of work, we might have an opportunity to schedule other kernels on it), and assigned stream should be explicitly synchronized with all streams that produce arguments.

(5): Unsupported operations

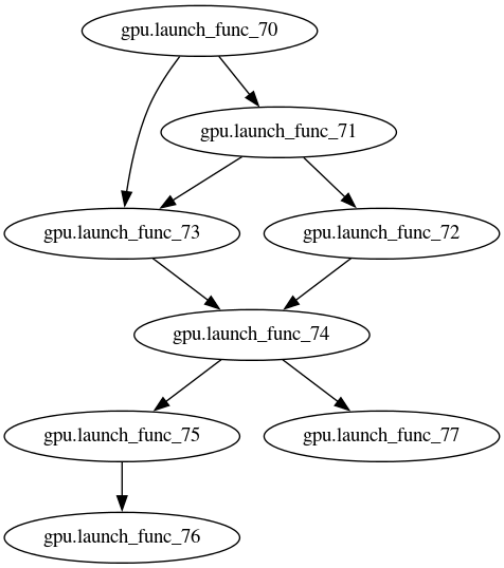
All operations that do not support multi-streaming are actually running on stream 0 (capture stream), and we should insert correct await operations to guarantee that all arguments are ready.

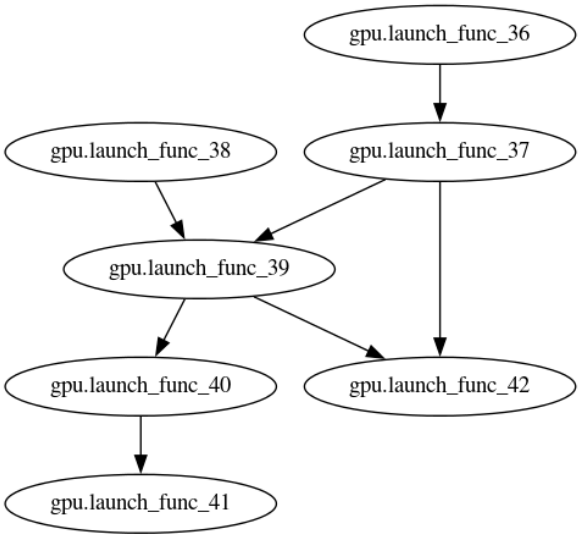
(6): Synchronize all streams

At the very end all extra streams should be synchronized with a capture stream, because it's illegal to build a CUDA graph with multiple concurrent "exit" nodes.

Analysis on dataflow graphs in real workloads

The following dataflow graphs are retrieved from the ULM ■■■■■■■ model. The config used is xla_gpu_min_graph_size=5, xla_gpu_cuda_graph_level=1.

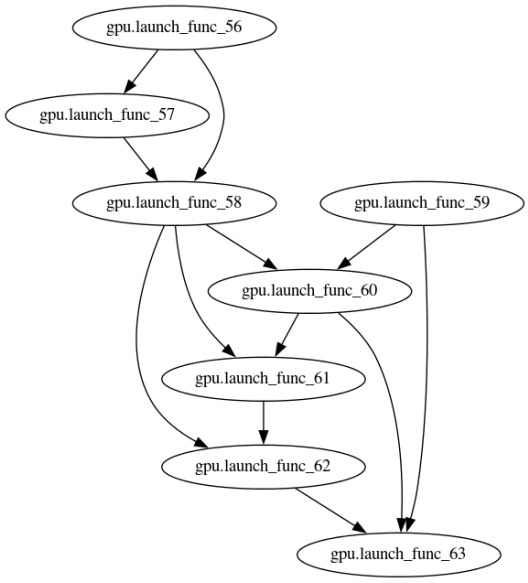




We currently have limited support for "multi-streaming" by outlining regions where ALL kernels can run concurrently (called a local concurrent region, see AddConcurrentRegionsPass in passes.td), and we create events before/after to sync with a "main" stream. But in practice we don't see many opportunities for such concurrency in the models. For example, in the dataflow graph on the left, the parallelism between gpu.launch_func_75 and gpu.launch_func_77 cannot be exploited by the local concurrent region, since they are not adjacent. The same pattern

appears In the graph on the right hand side, gpu.launch_func_40 and gpu.launch_func_42 can run concurrently.

There are some dataflow graphs that doesn't contain more parallelism that we can utilize: For example, in the following data flow graph, the parallelism between gpu.launch_func_58 and gpu.launch_func_59 is already exploited by the local concurrent region. However, the advanced multistreaming is still beneficial because we reduced the number of stream synchronizations: We only need to synchronize once after executing gpu.launch_func_59. In comparison, we needed 4 stream synchronizations before to run 58 and 59 concurrently.



Implementation

Overview

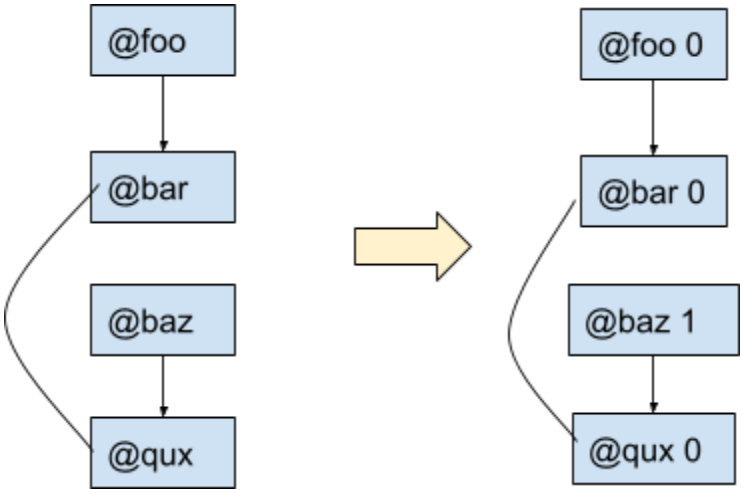
We want to implement a transformation pass that takes in a cuda graph capture function:

```
func @xla.cuda.graph.capture(...) {
  call @xla.kernel.launch @foo(%A, =>%B)
  call @xla.kernel.launch @bar(%B, =>%C)
  call @xla.kernel.launch @baz(%D, =>%E)
  call @xla.kernel.launch @qux(%C, %E, =>%F)
}
```

We will convert it to a data-flow graph that represents the data dependency between kernels. Then we will assign streams to each kernel so that independent kernels can run parallel on different streams. The following example shows the data-flow graph before and after stream assignment.

The left-hand side shows the data-flow graph without stream assignment. The kernels @foo @bar and @qux have to run sequentially since they form data dependency. @baz can run in parallel with @foo and @bar, but is dependent on @qux.

The right-hand side shows the graph after stream assignment. The @baz kernel is assigned to stream 1 because it can run in parallel with @foo and @bar.



Then we use the data-flow graph to assign streams to the kernel launches. Note that we have to add the explicit synchronization operation to indicate that @qux needs to wait for @baz to complete.

```
func @xla.cuda.graph.capture(...) {
  call @xla.kernel.launch @foo(%A, =>%B) { stream = 0 }
  call @xla.kernel.launch @bar(%B, =>%C) { stream = 0 }
  call @xla.kernel.launch @baz(%D, =>%E) { stream = 1 }
  call @xla.stream.await { from = 0, to = 1 }
  call @xla.kernel.launch @qux(%C, %E, =>%F) { stream = 0 }
}
```

Dataflow analysis

The goal of this phase is to construct a dataflow graph that represents the data dependency of operations inside the cuda graph capture function.

Since the cuda graph capture function is a straight-line program, it is trivial to do dataflow analysis on it. We can do a quadratic iteration on the operation list and see if there is a data dependency between each pair of operations. If two operations have a write-conflict, then there should be an edge between the two operations.

For kernels that do not support parallel execution, we simply add dependencies to all kernels before it.

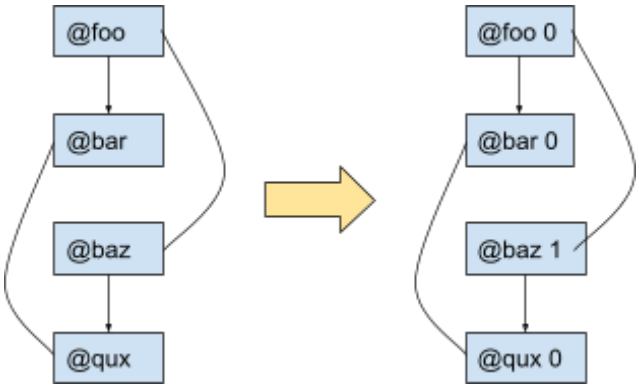
Stream assignment

We assign streams to each kernel to exploit the parallelism captured by the dataflow graph. There are two options to do this: (1) greedy stream assignment (2) Rescheduling + stream assignment.

Option 1: Greedy stream assignment

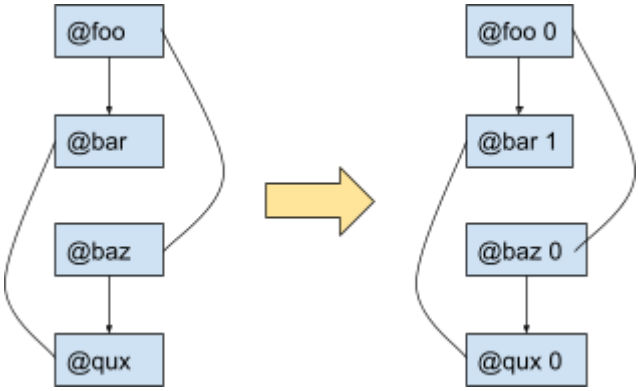
This option does not change the scheduling of the operations. Given a dataflow graph, we will need to first determine the number of streams required by looking at the maximum number of streams that can run parallel at a certain point.

Consider the simple dataflow graph below, where the order of scheduling is top to bottom. There is an opportunity to parallel execute @bar and @baz since there is no dependency between them. It is tempting to assign @foo and @bar to stream 0 since they are adjacent in the execution order, and @baz to stream 1 to make it parallel with @bar.



However, this is not ok. Because we need to wait for @foo to complete before executing @baz, which means that @baz needs to wait for stream 0 to finish execution. So there is no parallelism created between @bar and @baz.

One way to mitigate this is to always assign the same stream to the last child. So we assign stream 0 to @baz, instead of @bar. We can see in the assignment below, both @baz and @bar only need to wait for @foo to finish, so we have the parallelism.

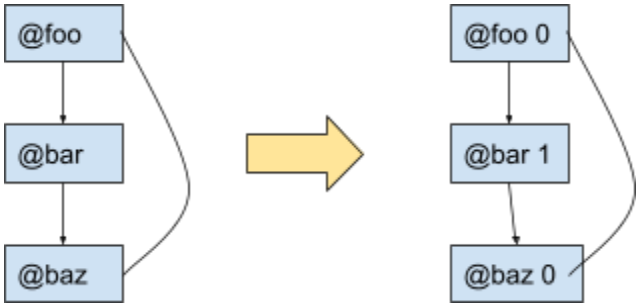


The pseudo code for the stream assignment algorithm:

```
while(there exists an operation unassigned):
      current_op <- the first unassigned op in the sequence
      stream <- next available stream
      assign stream to current_op
      while(there exists an unassigned operation in children(current_op))
            current_op = the last unassigned op in children(current_op)
            assign stream to current_op
```

Transitive reduction of dependency graph

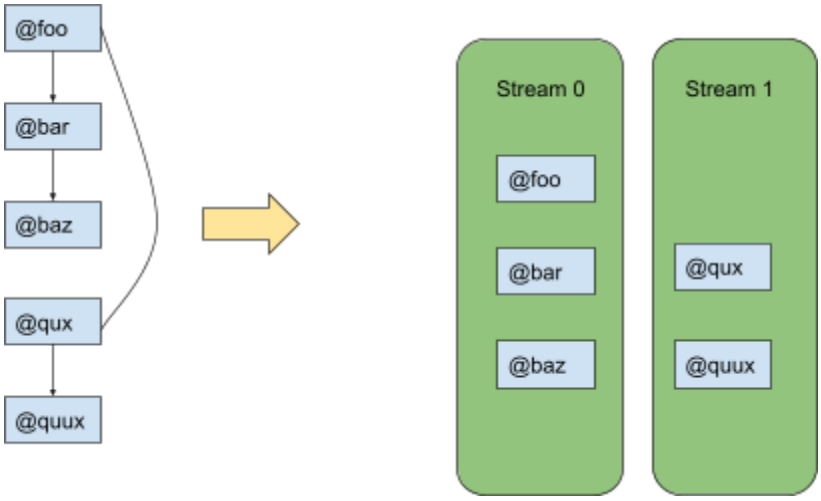
We should compute a transitive reduction of the dependency graph in order to make the stream assignment more efficient. Consider the following dependency graph. Using the algorithm we described, we assign both @foo and @bar to stream 0 since there is an edge connecting them. And we assign bar to stream 1. However this is inefficient since all kernels should be on the same stream. Computing a transitive reduction of the graph will avoid creating unnecessary stream assignments like this.



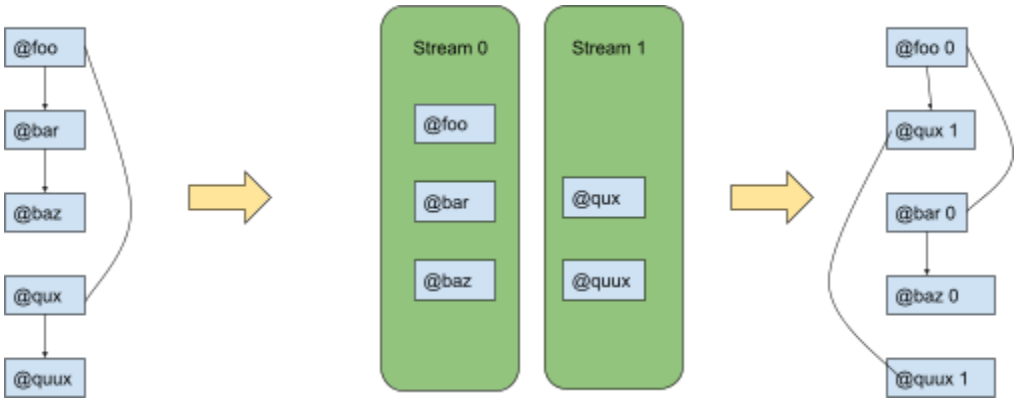
Option 2: Rescheduling + stream assignment

Another strategy is to allow reordering the kernels. Since we are reordering the kernels, we don't need to use the heuristic we developed in option 1. Instead, we assign streams using a scheduling algorithm, so that we assign an operation to a stream whenever its dependencies are finished.

We need to add the stream schedule as another intermediate representation. In the following example, we form the stream schedule on the right hand side by analyzing the dataflow graph on the left hand side.

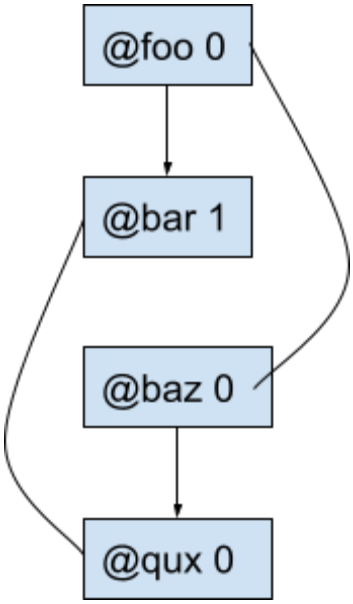


Then we can reschedule the operations and assign the stream according to the stream schedule. Specifically, we should put @qux before @bar so that @qux doesn't need to wait for @bar to finish. The dataflow graph after rescheduling and stream assignment will look like this:



Stream synchronization

Once we have the dataflow graph after stream assignment, we can add the assignment and the explicit synchronization to the cuda graph capture function in LMHLO. For example, from this stream-assigned dataflow graph, we can get positions to insert synchronization by looking at the edges that go from one stream to another stream.



The resulting cuda graph capture function will be this:

```
func @xla.cuda.graph.capture(...) {
  call @xla.kernel.launch @foo(%A, =>%B) { stream = 0 }
  call @xla.stream.await { from = 1, to = 0 }
  call @xla.kernel.launch @bar(%B, =>%C) { stream = 1 }
  call @xla.kernel.launch @baz(%B, =>%E) { stream = 0 }
  call @xla.stream.await { from = 0, to = 1 }
  call @xla.kernel.launch @qux(%C, %E, =>%F) { stream = 0 }
}
```

Implementation Plan

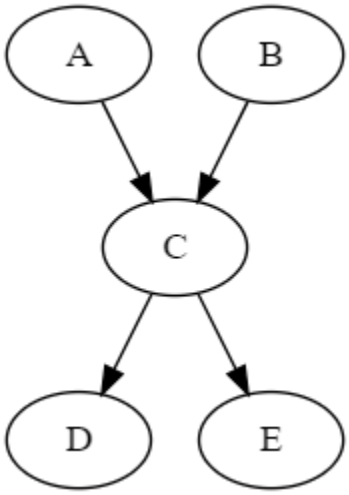
(1) Dataflow analysis (1 week)
  (a) We have a local dataflow analysis already implemented here. We can easily extend that to construct a dataflow graph. We will need to export the graph to the DOT file to investigate the dependencies.
(2) Stream assignment
  (a) The option 1 is relatively easy to implement, and should take 1-2 weeks.
  (b) After implementing option 1, evaluate the performance then consider if we should try option 2. Option 2 would take an additional ~2 weeks.

Appendix: CUDA graph capture creates redundant edges between nodes

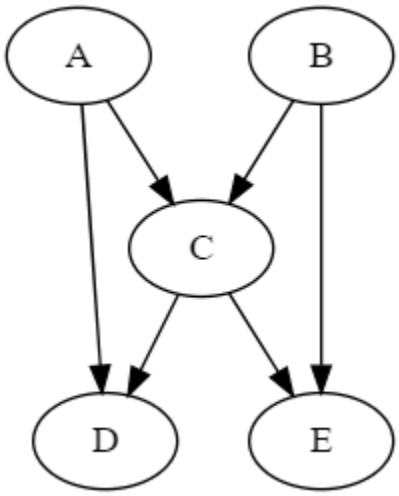
The following is an email that I sent about strange extra dependencies we see in capture cuda graphs, we never got a response, which might be relevant here.

I am writing to you today to ask for your assistance with a CUDA graph capture issue that I am experiencing. I am trying to capture CUDA graphs that contain cross-stream dependencies. My goal is to concurrently execute kernels in XLA. To achieve this, I dispatched each concurrent kernel to a separate stream and stream-captured the CUDA graph, so that the captured CUDA graph will know which kernels can run in parallel. The method I used to capture the graph is documented here: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#cross-stream-dependencies-and-events

The issue I am having is that, if I have two groups of concurrent kernels, graph edges will be created between nodes that belong to different groups. For example, I want to capture a CUDA graph where kernels A and B are parallel, kernels D and E are parallel, and the two groups of parallel kernels have dependencies with C. So the desired graph layout will be something like this:



But the actual result of the graph capture is this:



As you can see, edges A->D and B->E are redundant.

I believe that these redundant edges are formed because we reuse streams for the two concurrency groups to avoid allocating too many streams. For example, we run A and D on the same stream, and run B and E on another stream. However, I am not sure if this is the best practice, and I am also not sure if there will be an impact on performance.

Would you be able to help me understand why these redundant edges are formed, and whether they will affect the performance of the graph execution?

Other CUDA Graph questions

1. We have performance regression after enabling parallelism in CUDA graphs, which simply makes 2-3 independnent operations in each CUDA graph to be parallel. We had %93 percent regression in some models. What could be the mechanism that caused such regression? Could it be related to the redundant edges described above?