TensorFlow — Graph, GraphDef, Grappler, XLA, MLIR, LLVM, etc

Subrata Goswami · Follow
26 min read · Jul 23, 2022

7 1

TensorFlow is a large and evolving code base with mix of mostly C++ and Python code. It has grown immensely since its first public appearance in 2015. One of its fundamental parts is the processing and optimization of computation graph. This article is an attempts to understand some aspects of TensorFlow's graph processing and optimizations. The article is not expected to be complete, comprehensive and even fully coherent by any imagination. A lot of the texts here are directly and liberally copied from comments in code in TensorFlow github repo.

A list of glossary provides introduction to the many terms and acronyms used in the article , TensorFlow code base and other texts. Some of these are broad subjects by themselves.

The model code used for this publication is one of the TensorFlow XLA examples [1] — the source is also in the Appendix section. Although XLA did not accelerate this particular code, it is small enough to allow probing, as verbose logging from a complex model can be so voluminous so as to make it intractable. Several different techniques were used in addition to reading the source code — very verbose logging, profiling, graph dumping, gdb stack tracing, etc. Details of how these were done are provided in the appendix. Mostly TensorFlow version 2.9.1 was used for this article.

This article's focus is Grappler and XLA .However, some peripheral areas get touched upon also for better understanding.

Grappler (non-XLA) Path :

Internally the computation graph is held in a structure called Graph (see the Glossary for more details). GraphDef is a serialized form of the computation graph that can be written to a file.

Grappler is the main graph optimizer in TensorFlow and transforms the computation graph to run better and faster. It runs several types of optimizations on the graph [2]. It divides the optimizations into 4 groups (*OptimizationPassRegistry::RunGrouping*) — **PRE_PLACEMENT, POST_PLACEMENT, POST_REWRITE_FOR_EXEC,** and **POST_PARTITIONING.** Each group runs a number of optimization called phases. A phase may contain a number of pass. See Appendix 5 for the complete list.

Grappler phases are initiated from *ProcessFunctionLibraryRuntime::InstantiateMultiDevice* in *process_function_library_runtime.cc* .In non-XLA execution, the example code runs each phase 7 times.

The pass *generic_layout_optimzier* converts from NHWC layout to NCWH for running some kernels optimally.

Even when XLA jit is not used, passes like *MarkForCompilation* are still used. This pass marks nodes of the graph as clusterable for XLA.

Non-XLA execution has more Grappler calls than in XLA executions. Primarily, *analytical_cost_estimator.cc* , *op_level_cost_estimator.cc* , *graph_memory.cc* , and *vitual_scheduler.cc* . These appear to be called outside of the groups/phases/passes.

The *AnalyticalCostEstimator* estimates the cost of running a Grappler item based on the theoretical performance of the hardware that will run the model. This internally uses static shape inference. An option for aggressive shape inference is provided to minimize unknown shapes, and this is only applicable with static shape inference.

The *op_level_cost_estimator* makes cost estimate based on the given operations count and the given total IO size in bytes.

In the execution of the example code, the first iteration used more kernels than succeeding iterations. The additional kernels are *Conv2D*, *Conv2DBackpropFiler*, *Conv2DBackpropInput*, and *Fill*. For example, in the forward pass of the first iteration, during the second convolution (*conv2d_1*), 5 different kernels were used. Whereas in the succeeding iterations, only the one with shortest time is used The convolution kernels tried and chosen are shown below.

```
iter0 conv2d_1:
```

```
maxwell_scudnn_winograd_128x128_ldg1_ldg4_relu_tile228n_nt_v1,
maxwell_scudnn_winograd_128x128_ldg1_ldg4_relu_tile418n_nt_v1,
maxwell_scudnn_winograd_128x128_ldg1_ldg4_relu_tile244t_nt_v1,
maxwell_scudnn_winograd_128x128_ldg1_ldg4_relu_tile424n_nt_v1,
maxwell_scudnn_winograd_128x128_ldg1_ldg4_mobile_relu_tile148t_nt_v0

iter1+ conv2d_1:

maxwell_scudnn_winograd_128x128_ldg1_ldg4_mobile_relu_tile148t_nt_v0
```

The first iteration includes tuning to find the fastest algorithm. It is controlled by a variable called *TF_CUDNN_USE_AUTOTUNE* in *tensorflow/core/util/use_cudnn.cc* . Shown in the following code-snippet is one such call stack.

```
tensorflow/core/kernels/conv_ops.cc Conv2DOp.Compute()
tensorflow/core/kernels/conv_ops.cc LaunchConv2DOp()
tensorflow/core/kernels/conv_ops_gpu.cc AutotuneUnfusedConv()
tensorflow/core/kernels/gpu_utils.cc BestCudnnConvAlgorithm()
```

With VLOG ≥2, the fastest kernel for an op is printed to the log.

```
tensorflow/core/kernels/gpu_utils.cc:296] fastest algorithm:
580.864us with algo ConvFwd_eng12_k5=1_k6=0_k7=1_k10=1, workspace
bytes 102528
```

When profiling is enabled (*cupti_tracer.cc:384*), the kernel can also be viewed in a timeline trace — see figure below. However the names in these two domains are not same. For example, the name *ConvFwd_eng12_k5=1_k6=0_k7=1_k10=1* in *gpu_uitls.cc* and *cuda/cuda_dnn.cc* corresponds to *maxwell_scudnn_winograd_128x128_ldg1_ldg4_mobile_relu_tile148t_nt_v0* in *cupti_tracer.cc*.

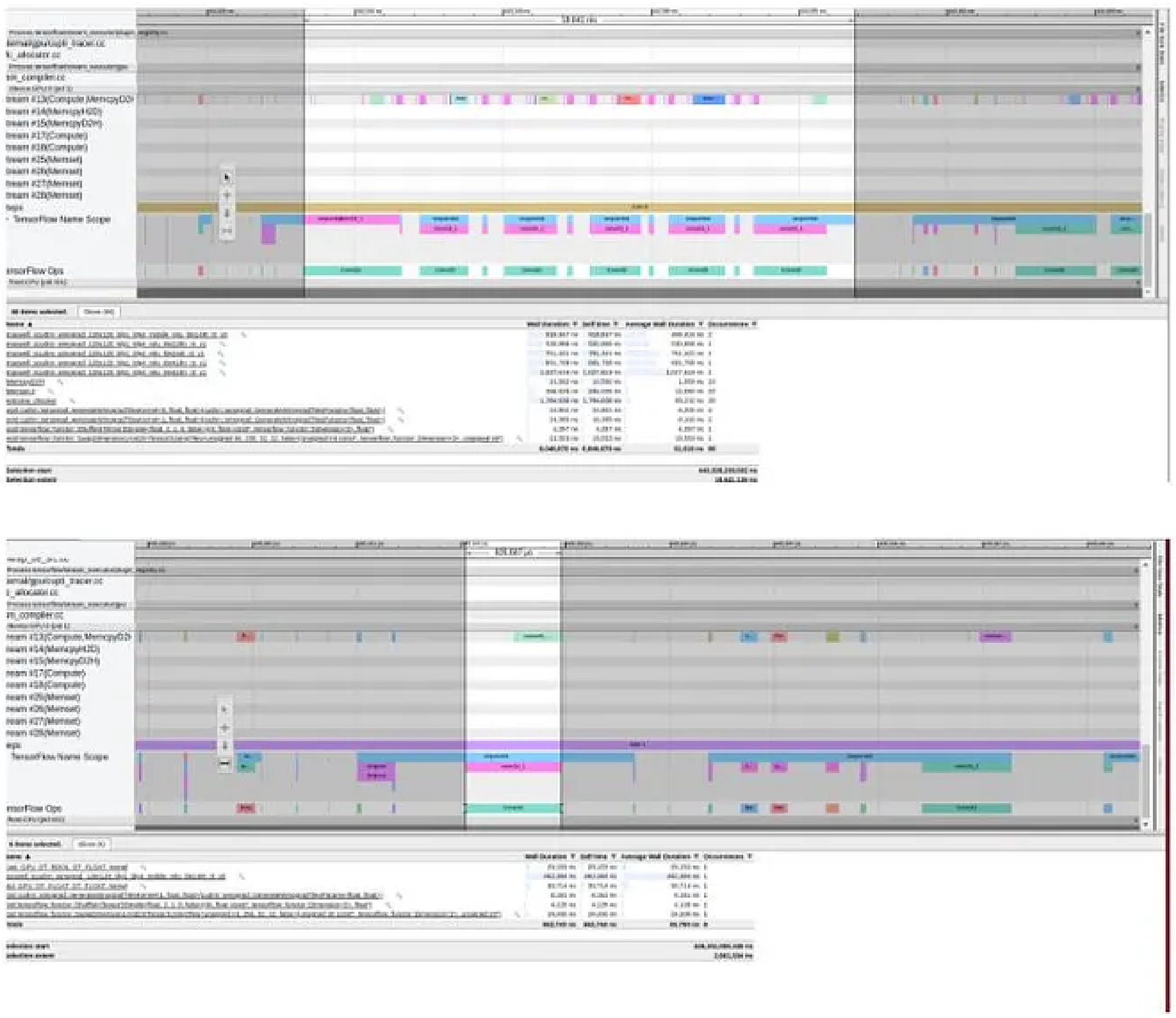


Fig 1: Autotunning to select best convolution kernel iter0 (top) and using the fastest in iter1 (bottom)

The following code snippet shows the call stack trace in *cupti_tracer.cc* .

```
#0  tensorflow::profiler::(anonymous
namespace)::AddKernelEventUponApiExit () at
tensorflow/core/profiler/internal/gpu/cupti_tracer.cc:384
#1
tensorflow::profiler::CuptiDriverApiHook::AddDriverApiCallbackEvent
() at tensorflow/core/profiler/internal/gpu/cupti_tracer.cc:1537
```

```
#2  tensorflow::profiler::(anonymous
namespace)::CuptiDriverApiHookWithActivityApi::OnDriverApiExit () at
tensorflow/core/profiler/internal/gpu/cupti_tracer.cc:869
#3  tensorflow::profiler::CuptiTracer::HandleCallback () at
tensorflow/core/profiler/internal/gpu/cupti_tracer.cc:1889
#4  tensorflow::profiler::(anonymous namespace)::ApiCallback () at
tensorflow/core/profiler/internal/gpu/cupti_tracer.cc:335
#5  libcupti.so.11.4
#6  libcupti.so.11.4
#7  libcupti.so.11.4
#8  libcuda.so.1
#9  libcudnn_cnn_infer.so.8
#10 libcudnn_cnn_infer.so.8
#11 cask_cudnn::WinogradShader::run() libcudnn_cnn_infer.so.8
#12
cudnn::cnn::infer::WinogradInferSubEngine<>::execute_internal_impl()
() from libcudnn_cnn_infer.so.8
#13 cudnn::cnn::EngineInterface::execute()() from
libcudnn_cnn_infer.so.8
#14 cudnn::cnn::EngineContainer () from libcudnn_cnn_infer.so.8
#15 cudnn::cnn::EngineInterface::execute()() from
libcudnn_cnn_infer.so.8
#16 cudnn::cnn::AutoTransformationExecutor::execute_pipeline()
libcudnn_cnn_infer.so.8
#17 cudnn::cnn::BatchPartitionExecutor::operator()()
libcudnn_cnn_infer.so.8
#18
cudnn::cnn::GeneralizedConvolutionEngine<>::execute_internal_impl()
() from libcudnn_cnn_infer.so.8
#19 cudnn::cnn::EngineInterface::execute(cudnn::backend::VariantPack
const&, CUstream_st*) () libcudnn_cnn_infer.so.8
#20 cudnn::backend::execute() () libcudnn_cnn_infer.so.8
#21 cudnnBackendExecute () libcudnn_cnn_infer.so.8
#22 cudnnBackendExecute () at
./tensorflow/stream_executor/cuda/cudnn_8_0.inc:1398
#23 stream_executor::gpu::CudnnExecutionPlanRunner<>::operator()() at
tensorflow/stream_executor/cuda/cuda_dnn.cc:4367
.
.
.
#30 tensorflow::ExecutorState<>::ProcessSync ()
tensorflow/core/common_runtime/executor.cc:584
#31 tensorflow::ExecutorState<>::Process () at
tensorflow/core/common_runtime/executor.cc:830
#32 tensorflow::ExecutorState<>::<lambda()>::operator()(void) const
() at tensorflow/core/common_runtime/executor.cc:1197
#33 tensorflow::ExecutorState<>::<lambda()>::operator()(void) () at
tensorflow/core/common_runtime/executor.cc:468
```

TensorFlow guards memory with *redzone_checker*. It is an allocator that allocates a bit of extra memory around the beginning/end of every allocation and checks that this memory is unmodified.

A *ptx* cache is also maintained , *CompileGpuAsmOrGetCached*. Only *redzone_checker* appears to have triggered access to the cache.

RunAsync() executes the graph computation (*executor.cc, executor.h*). It eventually gets called from *GraphRunner::Run(raph* graph ,….)* (*graph_runner.cc*) .

From the *tf2xla* directory, *xla_op_registry.cc* and *const_analysis.cc* are used in the Grappler only flow. *const_analysis.cc* is backwards dataflow analysis that finds nodes in a graph that must be compile-time constants to be able to lower the graph to XLA.

cudaLaunchKernel (from *libcudart.so* , *tensorflow/stream_executor/cuda/cuda_runtime_XX*) is the API used for launching kernels in GPU. Which is eventually called from *cudnnBackendExecute* in *tensorflow/stream_executor/cuda/cuda_dnn.cc* .

XLA (+Grappler) Path:

There are number of debug flags that can be set to see what XLA is doing [3] . Nvidia also provides some additional info on XLA [4]. A high level roadmap of XLA migration to MLIR in TensorFlow is provided in [5]. A few papers describe features and limitations of XLA at a high level [6,7]. XLA focuses on the memory bound ops and uses library kernels (*cudnn*, *cublas*, etc.) for compute intensive ops such as convolutions and gemms.

XLA path files :

XLA execution runs more codes from *jit*, *tf2xla* and *xla* directories in *tensorflow/compiler* than the Grappler-only path. A number of flies in the *tf2xla* and *jit* directories are also used in the Grappler-only path. It should be no surprise that 88 additional files are used from the *xla* directory in the XLA path. The file names, at a high level, hints at the transforms and optimizations applied to the ops and the graph.

Common in both XLA and Grappler paths.

```
tensorflow/compiler/jit/build_xla_ops_pass.cc
tensorflow/compiler/jit/compilability_check_util.cc
tensorflow/compiler/jit/deadness_analysis.cc
tensorflow/compiler/jit/encapsulate_subgraphs_pass.cc
tensorflow/compiler/jit/encapsulate_xla_computations_pass.cc
tensorflow/compiler/jit/introduce_floating_point_jitter_pass.cc
tensorflow/compiler/jit/mark_for_compilation_pass.cc
tensorflow/compiler/jit/resource_operation_safety_analysis.cc
tensorflow/compiler/jit/xla_activity_logging_listener.cc
tensorflow/compiler/jit/xla_cluster_util.cc
tensorflow/compiler/jit/xla_cpu_device.cc
tensorflow/compiler/jit/xla_gpu_device.cc

tensorflow/compiler/tf2xla/const_analysis.cc
tensorflow/compiler/tf2xla/xla_op_registry.cc

tensorflow/compiler/xla/parse_flags_from_env.cc

tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc
```

Code: Files used in both XLA and Grappler paths

Additional files used in XLA path.

```
tensorflow/compiler/jit/clone_constants_for_better_clustering.cc
tensorflow/compiler/jit/kernels/xla_ops.cc
```

```
tensorflow/compiler/jit/xla_compilation_cache.cc
tensorflow/compiler/jit/xla_launch_util.cc

tensorflow/compiler/tf2xla/graph_compiler.cc
tensorflow/compiler/tf2xla/kernels/reduction_ops_common.cc
tensorflow/compiler/tf2xla/kernels/reshape_op.cc
tensorflow/compiler/tf2xla/xla_compilation_device.cc
tensorflow/compiler/tf2xla/xla_compiler.cc
tensorflow/compiler/tf2xla/xla_context.cc

tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc
tensorflow/compiler/mlir/tensorflow/translate/import_model.cc
tensorflow/compiler/mlir/tensorflow/utils/bridge_logger.cc
tensorflow/compiler/mlir/tensorflow/utils/dump_mlir_util.cc
tensorflow/compiler/mlir/xla/transforms/xla_legalize_tf.cc

tensorflow/compiler/xla/literal_comparison.cc
tensorflow/compiler/xla/parse_flags_from_env.cc
tensorflow/compiler/xla/service/all_gather_combiner.cc
tensorflow/compiler/xla/service/all_reduce_combiner.cc
tensorflow/compiler/xla/service/batchnorm_expander.cc
tensorflow/compiler/xla/service/bfloat16_normalization.cc
tensorflow/compiler/xla/service/buffer_assignment.cc
tensorflow/compiler/xla/service/call_graph.cc
tensorflow/compiler/xla/service/call_inliner.cc
tensorflow/compiler/xla/service/conditional_canonicalizer.cc
tensorflow/compiler/xla/service/conditional_simplifier.cc
tensorflow/compiler/xla/service/copy_insertion.cc
tensorflow/compiler/xla/service/dfs_hlo_visitor.cc
tensorflow/compiler/xla/service/dfs_hlo_visitor_with_default.h
tensorflow/compiler/xla/service/dot_merger.cc
tensorflow/compiler/xla/service/dump.cc
tensorflow/compiler/xla/service/dynamic_dimension_inference.cc
tensorflow/compiler/xla/service/dynamic_dimension_simplifier.cc
tensorflow/compiler/xla/service/dynamic_padder.cc
tensorflow/compiler/xla/service/executable.cc
tensorflow/compiler/xla/service/flatten_call_graph.cc
tensorflow/compiler/xla/service/generic_transfer_manager.cc
tensorflow/compiler/xla/service/gpu/buffer_comparator.cc
tensorflow/compiler/xla/service/gpu/cudnn_fused_conv_rewriter.cc
tensorflow/compiler/xla/service/gpu/fusion_merger.cc
tensorflow/compiler/xla/service/gpu/gemm_algorithm_picker.cc
tensorflow/compiler/xla/service/gpu/gemm_thunk.cc
tensorflow/compiler/xla/service/gpu/gpu_compiler.cc
tensorflow/compiler/xla/service/gpu/gpu_conv_algorithm_picker.cc
tensorflow/compiler/xla/service/gpu/gpu_conv_rewriter.cc
tensorflow/compiler/xla/service/gpu/gpu_conv_runner.cc
tensorflow/compiler/xla/service/gpu/gpu_executable.cc
tensorflow/compiler/xla/service/gpu/hlo_to_ir_bindings.cc
tensorflow/compiler/xla/service/gpu/horizontal_input_fusion.cc
tensorflow/compiler/xla/service/gpu/horizontal_loop_fusion.cc
tensorflow/compiler/xla/service/gpu/ir_emitter_unnested.cc
```

```
tensorflow/compiler/xla/service/gpu/kernel_thunk.cc
tensorflow/compiler/xla/service/gpu/launch_dimensions.cc
tensorflow/compiler/xla/service/gpu/llvm_gpu_backend/gpu_backend_lib.
cc
tensorflow/compiler/xla/service/gpu/multi_output_fusion.cc
tensorflow/compiler/xla/service/gpu/nvptx_compiler.cc
tensorflow/compiler/xla/service/gpu/nvptx_helper.cc
tensorflow/compiler/xla/service/gpu/parallel_loop_emitter.cc
tensorflow/compiler/xla/service/gpu/reduction_dimension_grouper.cc
tensorflow/compiler/xla/service/gpu/reduction_layout_normalizer.cc
tensorflow/compiler/xla/service/gpu/reduction_splitter.cc
tensorflow/compiler/xla/service/gpu/stream_assignment.cc
tensorflow/compiler/xla/service/gpu/tree_reduction_rewriter.cc
tensorflow/compiler/xla/service/heap_simulator.cc
tensorflow/compiler/xla/service/hlo_alias_analysis.cc
tensorflow/compiler/xla/service/hlo_computation.cc
tensorflow/compiler/xla/service/hlo_computation.h
tensorflow/compiler/xla/service/hlo_constant_folding.cc
tensorflow/compiler/xla/service/hlo_cse.cc
tensorflow/compiler/xla/service/hlo_dataflow_analysis.cc
tensorflow/compiler/xla/service/hlo_dce.cc
tensorflow/compiler/xla/service/hlo_evaluator.cc
tensorflow/compiler/xla/service/hlo_graph_dumper.cc
tensorflow/compiler/xla/service/hlo_instruction.cc
tensorflow/compiler/xla/service/hlo_instructions.cc
tensorflow/compiler/xla/service/hlo_instructions.h
tensorflow/compiler/xla/service/hlo_memory_scheduler.cc
tensorflow/compiler/xla/service/hlo_module.cc
tensorflow/compiler/xla/service/hlo_parser.cc
tensorflow/compiler/xla/service/hlo_pass_fix.h
tensorflow/compiler/xla/service/hlo_pass_pipeline.cc
tensorflow/compiler/xla/service/hlo_phi_graph.cc
tensorflow/compiler/xla/service/hlo_proto_util.cc
tensorflow/compiler/xla/service/hlo_schedule.cc
tensorflow/compiler/xla/service/hlo_verifier.cc
tensorflow/compiler/xla/service/instruction_fusion.cc
tensorflow/compiler/xla/service/layout_assignment.cc
tensorflow/compiler/xla/service/llvm_ir/fused_ir_emitter.cc
tensorflow/compiler/xla/service/local_service.cc
tensorflow/compiler/xla/service/platform_util.cc
tensorflow/compiler/xla/service/reduce_scatter_combiner.cc
tensorflow/compiler/xla/service/reshape_mover.cc
tensorflow/compiler/xla/service/service.cc
tensorflow/compiler/xla/service/shape_inference.cc
tensorflow/compiler/xla/service/slow_operation_alarm.cc
tensorflow/compiler/xla/service/sort_simplifier.cc
tensorflow/compiler/xla/service/stream_pool.cc
tensorflow/compiler/xla/service/tuple_points_to_analysis.cc
tensorflow/compiler/xla/service/while_loop_constant_sinking.cc
tensorflow/compiler/xla/service/while_loop_simplifier.cc
tensorflow/compiler/xla/shape.cc
```

```
tensorflow/compiler/xla/shape_util.cc
tensorflow/compiler/xla/util.cc
```

Code: Additional files used in XLA-path

XLA Ops:

XlaLaunch, *_XlaCompile*, *XlaRun* , *_XlaMerge*, and *XlaClusterOutput* are the ops that XLA rewrites a Grappler graph with (*tensorflow/compiler/jit/ops/xla_ops.cc*).

The *encapsulate_xla_computations_pass/EncapsulateXlaComputationsPass* rewrites computations generated by *xla.compile()* Python code into *XlaLaunch* nodes. *xla.compile()* does two things — marks operators that make up an XLA computation with a unique id (e.g. *_xla_compile_id=XYZ*), and add *XlaClusterOutput* nodes to represent outputs of the computation.

XlaLaunch (XlaLocalLaunchOp) is used to replace a region of the TensorFlow graph that will be compiled and executed using XLA. The *XlaLocalLaunchOp* is responsible for handling interactions with the TensorFlow executor. Once all inputs are present, and their shapes are known, the op can use a '*XlaCompilationCache*' to compile and execute code which is specific to the shapes of input Tensors.

_XlaCompile compiles a TensorFlow function into an XLA LocalExecutable and returns a key that *_XlaRun* can use to look up the LocalExecutable and execute it. *XlaCompiler* (*tensorflow/compiler/tf2xla/xla_compiler.h*) is typically invoked from an *XlaLaunch* operator once the shapes of all input parameters to the computation are known, as symbolic execution requires known shapes for all operations. Eventually the nvtx compiler (e.g. nvcc) is invoked here to compile llvm into machine code (*tensorflow/compiler/xla/service/gpu/nvptx_compiler.cc*)

XlaCompileOnDemandOp (*tensorflow/compiler/jit/xla_compile_on_demand_op.h*) is an *OpKernel* when its Compute method is called, generates *xla::Computation* and runs it asynchronously. Unlike *XlaLaunch,* it does not rely on any rewrites of the GraphDef — it runs a vanilla TensorFlow op as long as the bridge supports it.

The *XlaCompiler* class is responsible for compilation of a self-contained subgraph of a TensorFlow computation using XLA linear algebra runtime. It does a symbolic execution of the graph starting from specific input shapes, using a JIT device to convert operators into XLA computations. It is typically invoked from an `XlaLaunch` operator once the shapes of all input parameters to the computation are known. This is because the symbolic execution requires known shapes for all operations.

The *XlaCompilationCache* class caches the results of the XlaCompiler class, which converts a Tensorflow graph into a compiled XLA compilation. Since XLA computations must have static shapes, the cache generates a new XLA computation for each new set of input shapes.

_XlaMerge merges the outputs from the *PartitionedCall* node and the *_XlaRun* node. Unlike the TensorFlow Merge op, which requires inputs of some types to be placed on the host, the *_XlaMerge* op can merge inputs of all types when placed on the device. This prevents the need for copy operations, in particular when an XLA cluster has int32 outputs. The *_XlaMerge* up does not have a value_index output that identifies the chosen input.

GraphCompiler (*tensorflow/compiler/tf2xla/graph_compiler.h/cc*) compiles the graph in topological order. It also resolves the nondeterminism in the graph by enforcing a total order on all inputs to a node. This abstraction helps in creating the same XLA computation given two structurally equivalent TensorFlow graphs. If a function call is visited during the graph traversal, it is then compiled through the *xla_context* into a computation and a `Call` operation is inserted to call into that computation. It is called from *XlaCompiler* class.

XLA Flow:

TensorFlow computation graph starts to get processed for XLA in *MarkForCompilationPassImpl::Run()*. It runs *RunEdgeContractionLoop(), DeclusterNodes(), CreateClusters(),* and *DumpDebugInfo()* .

RunEdgeContractionLoop() method contracts as many edges as possible to create XLA clusters. After this finishes clustering , decisions made are stored in name `*clusters_X*`, where X is an integer.

Autoclustering sometimes be overeager. For example, clustering large constants (or large broadcasts of constants) can increase the live range of those constants, and increase overall memory usage. *DeclusterNodes()* removes "obviously bad" cases like these.

MarkForCompilationPassImpl::CreateClusters () , marks clusters for compilation that are placed on a device that requires compilation (an *XlaDevice*), are explicitly marked for compilation (*_XlaCompile=true*), or have more than *debug_options_.xla_min_cluster_size* elements (applicable only if compilation is enabled, otherwise there will be no such candidates). Marks nodes that are to be compiled with attribute *kXlaClusterAttr*. Nodes with the same cluster ID will be compiled together. Marks a subset of nodes in the graph which are to be clustered with an attribute _*XlaCluster=<cluster id>* so they are picked up by the *EncapsulateSubgraphsPass*. This method makes call to *ShouldCompileCluster ()* to determine whether to proceed with clustering (e.g. Grappler-only path when XLA is not enabled, etc.). Hence, the XLA-path has lot more activity during this period.

The call stack at this point is shown below.

```
tensorflow/compiler/jit/mark_for_compilation_pass.cc:980,1534,1825,18
56
#0  MarkForCompilationPassImpl::CreateClusters
#1  MarkForCompilationPassImpl::Run (this=0x7fffffff8f60)
#2  MarkForCompilation ( options=..., debug_options=...)
#3  MarkForCompilationPass::Run ( this=0x555555ea7ef0, options=...)

tensorflow/core/common_runtime/optimization_registry.cc:73
#4
OptimizationPassRegistry::RunGrouping(grouping=POST_REWRITE_FOR_EXE,.
.)

tensorflow/core/common_runtime/process_function_library_runtime.cc:94
9,1466
#5  ProcessFunctionLibraryRuntime::InstantiateMultiDevice
(function_name=..., )
#6  ProcessFunctionLibraryRuntime::Instantiate (function_name=...,)

tensorflow/core/common_runtime/eager/kernel_and_device.cc:239,247
#7  KernelAndDeviceFunc::InstantiateFunc (ndef=...,
graph_collector=0x0)
#8  KernelAndDeviceFunc::Init (ndef=..., graph_collector=0x0)

tensorflow/core/common_runtime/eager/execute.cc:1123,1289,1694
#9  GetOrCreateKernelAndDevice ()
#10 EagerLocalExecute ()
#11 EagerExecute ()

tensorflow/core/common_runtime/eager/core.cc:204
#12 EagerOperation::Execute ()

tensorflow/core/common_runtime/eager/custom_device_op_handler.cc:95
#13 tensorflow::CustomDeviceOpHandler::Execute ()

tensorflow/c/eager/c_api.cc:879
#14 TFE_Execute ()
```

`Code: MarkForCompilationPass call stack`

The TF computation graph is then converted into MHLO in *BuildHloFromGraph* . The call stack at this stage is listed below. *MlirXlaOpKernel* is an *XlaOpKernel* that is implemented by lowering MLIR TensorFlow to HLO legalization. MLIR Tensoflow is defined in *tensorflow/compiler/mlir/tensorflow/ir/tf_dialect.h*. *BuildHloFromGraph* first calls *GraphToModule* to generate an MLIR module from graph. It lowers TF IR to MHLO and insert HLO into the XlaBuilder, xla_params are HLO-level inputs to module_op that have already been added to the XlaBuilder, returns are the XlaOps.

BuildHloFromTfInner converts MLIR module to XLA HLO proto contained in XlaComputation. *BuildHloFromTfInner* calls *mlir::BuildHloFromMlirHlo* , which in turn calls *ConvertToHloModule.lower()* that eventually lowers to XLA HLO .

```
tensorflow/compiler/mlir/xla/mlir_hlo_to_hlo.cc:1907,2526
#0  ConvertToHloModule::Lower () at
#1  mlir::BuildHloFromMlirHlo ()

tensorflow/compiler/mlir/tensorflow/utils/compile_mlir_util.cc
#2 BuildHloFromTfInner ()
#3 BuildHloFromTf ()
#4 BuildHloFromModule ()
#5 BuildHloFromGraph ()

tensorflow/compiler/tf2xla/mlir_xla_op_kernel.cc
#6 MlirXlaOpKernel::ConstructXlaOp ()
#7 MlirXlaOpKernel::Compile (this=..., ctx=...)

tensorflow/compiler/tf2xla/xla_op_kernel.cc
#8 XlaOpKernel::Compute (this=..., context=...)

tensorflow/compiler/tf2xla/xla_compilation_device.cc
#9 XlaCompilationDevice::Compute ()
```

```
tensorflow/compiler/tf2xla/graph_compiler.cc
#10 GraphCompiler::Compile (this=...)

tensorflow/compiler/tf2xla/xla_compiler.cc
#11 ExecuteGraph ()
#12 XlaCompiler::CompileGraph ()
#13 XlaCompiler::CompileFunction ()

tensorflow/compiler/jit/xla_compilation_cache.cc
#14 XlaCompilationCache::<lambda()>::operator()(void) const ()
#15 XlaCompilationCache::CompileStrict ()
#16 XlaCompilationCache::CompileImpl ()
#17 XlaCompilationCache::Compile ()

tensorflow/compiler/jit/kernels/xla_ops.cc
#18 CompileToLocalExecutable ()
#19 XlaCompileOp::Compute (this=..., ctx=...)

tensorflow/core/common_runtime/gpu/gpu_device.cc
#20 BaseGPUDevice::Compute ()

tensorflow/core/common_runtime/executor.cc
#21 ExecutorState<tensorflow::PropagatorState>::ProcessSync ()
```

Code: BuildHloFromGraph call stack

The first MLIR file *build_hlo_tf_before.mlir*, is dumped around here also. Some good comments are in the header file. *v2.9.1/tensorflow/compiler/mlir/tensorflow/utils/compile_mlir_util.h* .

HLO instructions are in DAG form and represent the atomic computations built up via the XLA service interface [11]. They are ultimately lowered in a platform-aware way by traversing the HLO DAG and emitting a lowered form (e.g. *DfsHloVisitor*) . Nodes do not have total order within their computation, only have a partial ordering determined by data and control dependencies. HLO does not have basic blocks or explicit "branch" instructions. Instead, certain *HloInstructions* — namely, *kWhile*, *kConditional*, and *kCall* — encode control flow. For example, the *kConditional* HLO executes one of two possible computations, depending on the runtime value of a predicate. As of TF 2.9.1, HLO IR has 117 opcodes [11].

HloModule is the top-level unit in the HLO IR. It corresponds to a whole "program". Running a module, from beginning to end, is the only way to run an XLA program. A module contains one "entry computation", this *HloComputation* is like main() in a C program. The result of running the module is the result of running this computation. A module also contains some number of "nested computations". Each nested computation is attached to an *HloInstruction* within some other computation.

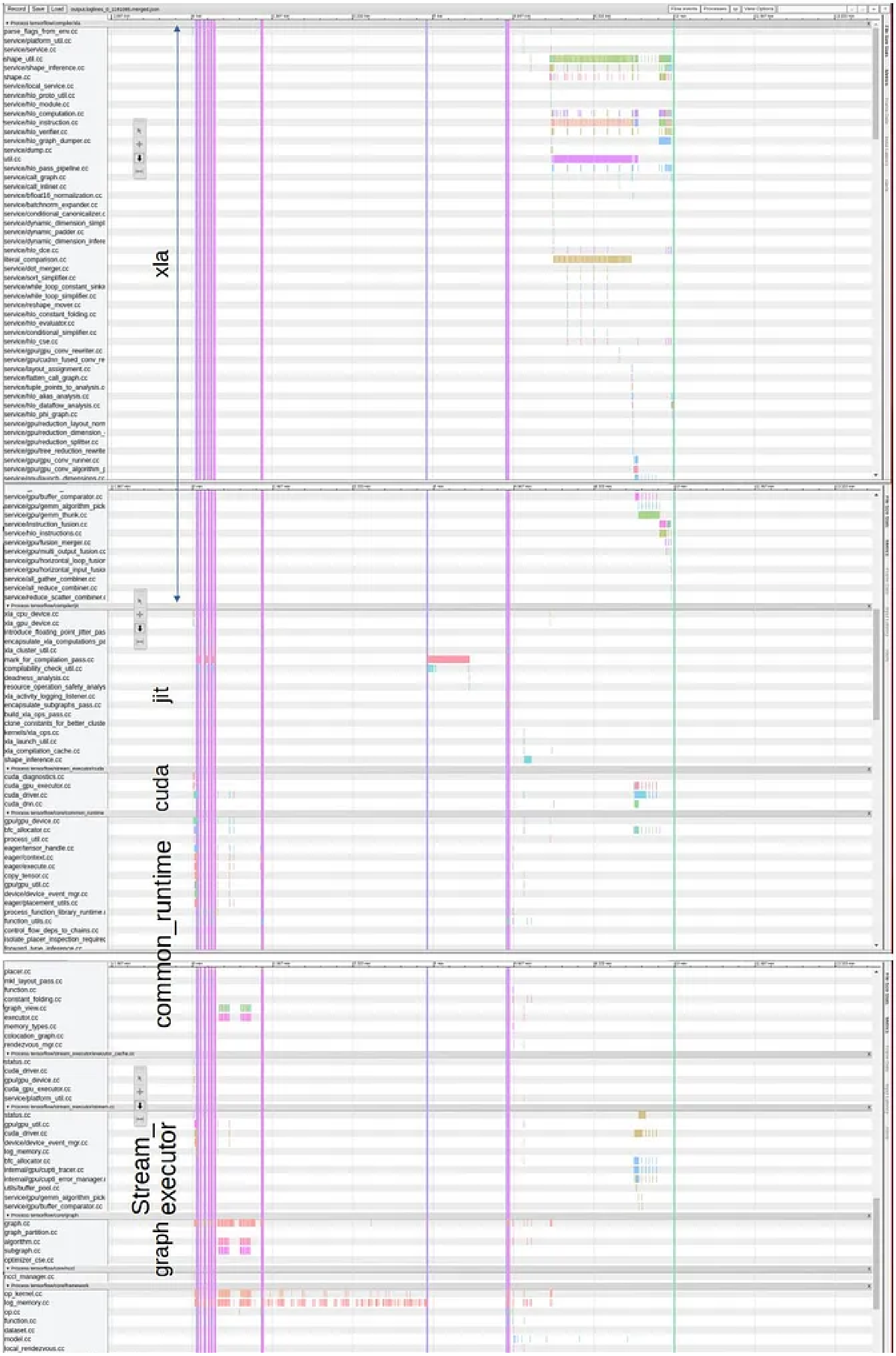
Then XLA related optimizations are done through codes in files in *tensorflow/compiler/xla/** — see **Code: Additional files used in XLA-path** above, for the complete list of files . XLA does not JIT build convolutions and gemms, but picks appropriate kernels from libraries (*gpu_conv_algorithm_picker.cc* , *gemm_algorithm_picker.cc*) . The following shows the call stack at the convolution kernel picking moment.

```
compiler/xla/service/gpu/gpu_conv_algorithm_picker.cc:497,716,392,915
,992,1009
#0  xla::gpu::GpuConvAlgorithmPicker::AutotuneOneConvRunner ( )
#1  xla::gpu::GpuConvAlgorithmPicker::PickBestAlgorithmNoCacheCuda ()
#2  xla::gpu::GpuConvAlgorithmPicker::PickBestAlgorithm ()
#3  xla::gpu::GpuConvAlgorithmPicker::RunOnInstruction ()
#4  xla::gpu::GpuConvAlgorithmPicker::RunOnComputation ()
#5  xla::gpu::GpuConvAlgorithmPicker::Run ()

compiler/xla/service/hlo_pass_pipeline.h:123 180 272
#6  xla::HloPassPipeline::RunHelper ()
#7  xla::HloPassPipeline::RunPassesInternal<xla::HloModule> ()
#8  xla::HloPassPipeline::Run ()

compiler/xla/service/gpu/gpu_compiler.cc:749
#9  xla::gpu::GpuCompiler::OptimizeHloPostLayoutAssignment ()

compiler/xla/service/gpu/nvptx_compiler.cc:145
#10 xla::gpu::NVPTXCompiler::OptimizeHloPostLayoutAssignment ()

compiler/xla/service/gpu/gpu_compiler.cc:548 763
#11 xla::gpu::GpuCompiler::OptimizeHloModule ()
#12 xla::gpu::GpuCompiler::RunHloPasses ()

compiler/xla/service/service.cc:797
#13 xla::Service::BuildExecutable ()
```

```
compiler/xla/service/local_service.cc:172
#14 xla::LocalService::CompileExecutables ()

compiler/xla/client/local_client.cc:393
#15 xla::LocalClient::Compile ()

compiler/jit/xla_compilation_cache.cc:288 539 803 335
#16 ::XlaCompilationCache::BuildExecutable ()
#17 ::XlaCompilationCache::CompileStrict ()
#18 ::XlaCompilationCache::CompileImpl ()
#19 ::XlaCompilationCache::Compile ()

compiler/jit/kernels/xla_ops.cc:271 457 679
#20 ::CompileToLocalExecutable ()
#21 ::XlaCompileOp::Compute ()
#22 ::BaseGPUDevice::Compute ()

core/common_runtime/executor.cc:584 830 1197 468
#23 ::ExecutorState<::PropagatorState>::ProcessSync ()
#24 ::ExecutorState<::PropagatorState>::Process ()
#25 ::ExecutorState<::PropagatorState>::<lambda()>::operator()()
const ()
#26 ::ExecutorState<::PropagatorState>::<lambda()>::operator()()()
```

`Code: gpu_conv_algorithm_picker call stack`

The following trace diagram shows the various XLA transformations optimizations , compiling cascading through time. On the left are the file names in the *xla* directory — same ones as in in **Code: Additional files used in XLA-path**. Along horizontal is the time as was logged in the log file. Each log line (lines emitted with VLOG ≤ 5 setting) is charted as one event of 1 ms fixed duration. The trace provides a high level visual view along with finer details on how XLA optimization and compilation progresses. However, not all files have log outputs. In those cases a tool like gdb is used.


xla
parse_flags_from_env.cc
service/platform_util.cc
service/service.cc
shape_util.cc
service/shape_inference.cc
shape.cc
service/local_service.cc
service/hlo_proto_util.cc
service/hlo_module.cc
service/hlo_computation.cc
service/hlo_instruction.cc
service/hlo_verifier.cc
service/hlo_graph_dumper.cc
service/dump.cc
util.cc
service/hlo_pass_pipeline.cc
service/call_graph.cc
service/call_inliner.cc
service/bfloat16_normalization.cc
service/batchnorm_expander.cc
service/conditional_canonicalizer.c
service/dynamic_dimension_simpl
service/dynamic_padder.cc
service/dynamic_dimension_infere
service/hlo_dce.cc
literal_comparison.cc
service/dot_merger.cc
service/sort_simplifier.cc
service/while_loop_constant_sinki
service/while_loop_simplifier.cc
service/reshape_mover.cc
service/hlo_constant_folding.cc
service/hlo_evaluator.cc
service/conditional_simplifier.cc
service/hlo_cse.cc
service/gpu/gpu_conv_rewriter.cc
service/gpu/cudnn_fused_conv_re
service/layout_assignment.cc
service/flatten_call_graph.cc
service/tuple_points_to_analysis.c
service/hlo_alias_analysis.cc
service/hlo_dataflow_analysis.cc
service/hlo_phi_graph.cc
service/gpu/reduction_layout_norm
service/gpu/reduction_dimension_
service/gpu/reduction_splitter.cc
service/gpu/tree_reduction_rewrite
service/gpu/gpu_conv_runner.cc
service/gpu/gpu_conv_algorithm_p
service/gpu/buffer_comparator.cc
service/gpu/gemm_algorithm_pick
service/gpu/gemm_thunk.cc
service/instruction_fusion.cc
service/hlo_instructions.cc
service/gpu/fusion_merger.cc
service/gpu/multi_output_fusion.cc
service/gpu/horizontal_loop_fusion
service/gpu/horizontal_input_fusio
service/all_gather_combiner.cc
service/all_reduce_combiner.cc
service/reduce_scatter_combiner.c
jit
xla_cpu_device.cc
xla_gpu_device.cc
introduce_floating_point_jitter_pas
encapsulate_xla_computations_pa
xla_cluster_util.cc
mark_for_compilation_pass.cc
compilability_check_util.cc
deadness_analysis.cc
resource_operation_safety_analys
xla_activity_logging_listener.cc
encapsulate_subgraphs_pass.cc
build_xla_ops_pass.cc
clone_constants_for_better_cluste
kernels/xla_ops.cc
xla_launch_util.cc
xla_compilation_cache.cc
shape_inference.cc
cuda
cuda_diagnostics.cc
cuda_gpu_executor.cc
cuda_driver.cc
cuda_dnn.cc
common_runtime
gpu/gpu_device.cc
bfc_allocator.cc
process_util.cc
eager/tensor_handle.cc
eager/context.cc
eager/execute.cc
copy_tensor.cc
gpu/gpu_util.cc
device/device_event_mgr.cc
eager/placement_utils.cc
process_function_library_runtime.
function_utils.cc
control_flow_deps_to_chains.cc
isolate_placer_inspection_required
forward_type_inference.cc
placer.cc
mkl_layout_pass.cc
function.cc
constant_folding.cc
graph_view.cc
executor.cc
memory_types.cc
colocation_graph.cc
rendezvous_mgr.cc
Stream_ executor
status.cc
cuda_driver.cc
gpu/gpu_device.cc
cuda_gpu_executor.cc
service/platform_util.cc
status.cc
gpu/gpu_util.cc
cuda_driver.cc
device/device_event_mgr.cc
log_memory.cc
bfc_allocator.cc
internal/gpu/cupti_tracer.cc
internal/gpu/cupti_error_manager.
utils/buffer_pool.cc
service/gpu/gemm_algorithm_pick
service/gpu/buffer_comparator.cc
graph
graph.cc
graph_partition.cc
algorithm.cc
subgraph.cc
optimizer_cse.cc
nccl_manager.cc
op_kernel.cc
log_memory.cc
op.cc
function.cc
dataset.cc
model.cc
local_rendezvous.cc

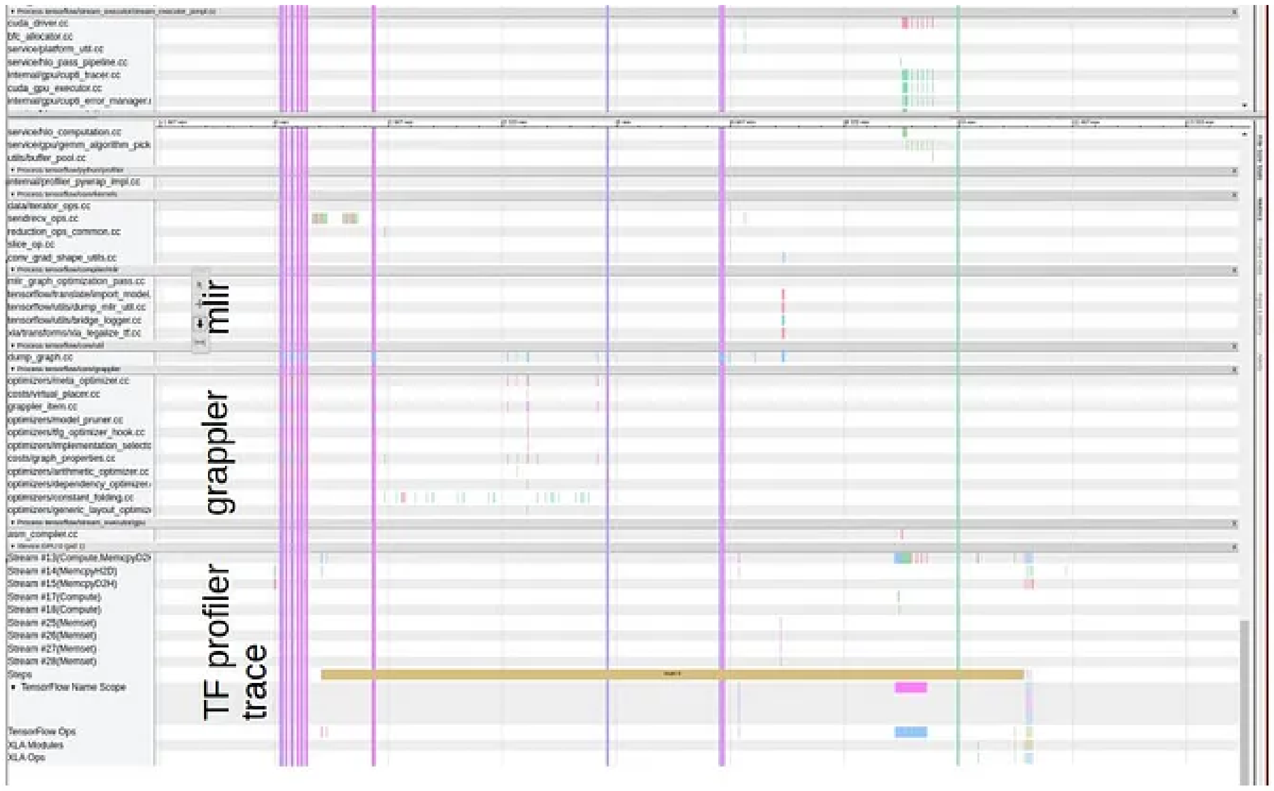


Fig 2: Time trace of log outputs from files in the xla , jit, grappler, mlir, etc directories.

At a broad visual level, the top part of the above chart corresponds to the files in the XLA directory. The bottom most part is TensorFlow profiler output. The blue rectangle there corresponds to _*XlaCompile*, where HLO gets lowered into LLVM , then to NVPTX (through nvcc compiler). The _*XlaCompile* op is registered to code in *XlaCompileOp*. The following lists the call stack at nvptx compilation.

```
compiler/xla/service/gpu/nvptx_compiler.cc:480 369 1063 1137
#0  xla::gpu::NVPTXCompiler::CompileGpuAsmOrGetCachedResult ()
#1  xla::gpu::NVPTXCompiler::CompileTargetBinary[abi:cxx11]()
#2  xla::gpu::GpuCompiler::<lambda()>::operator()()
#3  xla::gpu::GpuCompiler::CompileToTargetBinary[abi:cxx11]()()

tensorflow/compiler/xla/service/gpu/gpu_compiler.cc: 1280
#4  xla::gpu::GpuCompiler::RunBackend ()
```

```
compiler/xla/service/service.cc:801
#5  xla::Service::BuildExecutable ()

compiler/xla/service/local_service.cc:172
#6  xla::LocalService::CompileExecutables ()

compiler/xla/client/local_client.cc:393
#7  xla::LocalClient::Compile ()

compiler/jit/xla_compilation_cache.cc:288 539 803 335
#8  tensorflow::XlaCompilationCache::BuildExecutable ()
#9  tensorflow::XlaCompilationCache::CompileStrict ()
#10 tensorflow::XlaCompilationCache::CompileImpl ()
#11 tensorflow::XlaCompilationCache::Compile ()

compiler/jit/kernels/xla_ops.cc:271 457
#12 tensorflow::CompileToLocalExecutable ()
#13 tensorflow::XlaCompileOp::Compute ()

core/common_runtime/gpu/gpu_device.cc:679
#14 tensorflow::BaseGPUDevice::Compute ()

core/common_runtime/executor.cc:597
#15 tensorflow::()::ExecutorState<>::ProcessSync()
```

Code: NNPTX compiler call stack

Just before *CompileToTargetBinary()* in *RunBackend()*, *CompileModuleToLlvmIrImpl()* is called to lower HLO to llvm

XLA Optimizations:

XLA does several optimizations on HLO. Each of these are called pass (*tensorflow/compiler/xla/service/hlo_pass_pipeline.cc*). C++ classes for pass inherits from *HloModulePass* , *HloModuleGroupPass* and *HloPassInterface*. The passes are located in the *tensorflow/compiler/xla/service* directory. Some examples of passes follow.

The *hlo_constant_folding.h* pass performs constant folding in order to eliminate unnecessary computation on constants (e.g. 11.1*22.2 , 0*x, etc.) at runtime by performing these computations at compile time.

The *batchnorm_expander.h* pass breaks batch norm operations into more operations. Breaking a big operation into smaller operations helps leverage XLA's generic fusion logic.

The *hlo_cse*.h pass performs common-subexpression elimination. Identical constants and identical instructions with the same operands are "commoned". The pass iterates over the instructions in topological order which enables the pass to find arbitrarily large common expressions. A toy example is shown in the code snippet below.

```
a = b * c + g;
d = b * c * e;

// b*c is saved in tmp as a cse.

tmp = b * c;
a = tmp + g;
d = tmp * e;
```

The *gemm_algorithm_picker.h* pass, as the name suggests, picks the best gemm.

The *gemm_rewriter.h* pass pattern-matches the most general form of the following HLO instruction (assuming transposes are already folded), and rewrites it into a custom call, where (A, B, C) are three operands respectively, and `alpha` and `beta` are stored in the backend config.

```
cuBLAS GEMM in the most general form can run the following operation:

(kAdd
    (kMultiply (kDot A B) alpha)
    (kMultiply C beta))
```

```
where A, B, C are matrixes and `alpha` and `beta` are host constants.
The additional requirement is that C has no other users (otherwise,
it does not make sense to fuse it inside the custom call).
```

The *dot_merger.h* pass merges dots that share an operand as follows.

```
Transforms
//
//   x = dot(a, b)
//   y = dot(a, c)
//
// into
//
//   z = dot(a, concat(b, c))
//   x = slice(z)
//   y = slice(z).
//
// This requires that x and y are independent -- that is,
// x does not transitively depend on y, and
// y does not transitively depend on x.
```

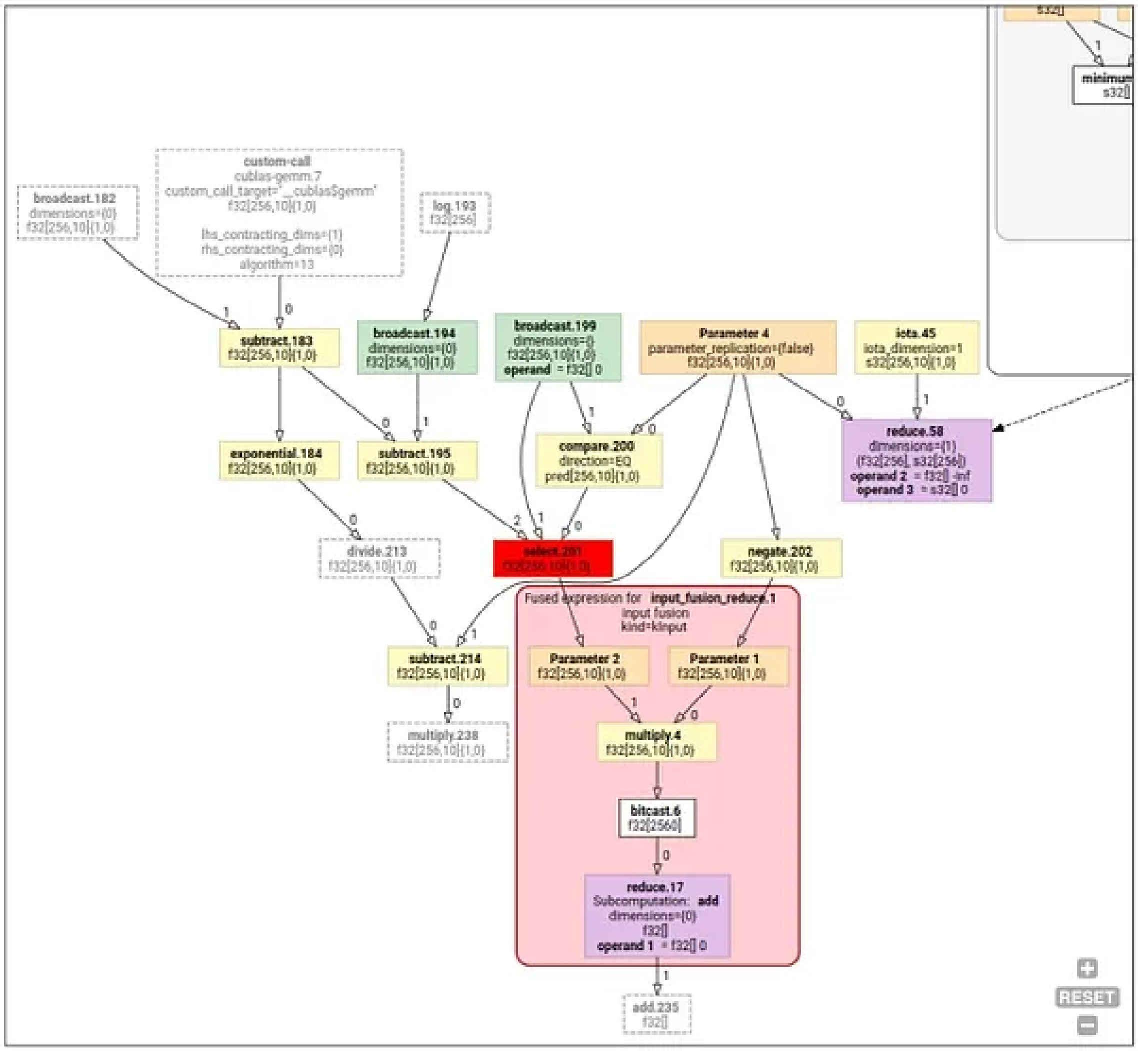
The *hlo_dce.h* pass removes dead instructions from each computation in the module and removes dead computations from the module. An instruction is dead if it is not reachable from the root. A computation is dead if it is not the entry computation of the module and it is not reachable from the entry computation.

The *gpu_conv_algorithm_picker.h* pass modifies *CustomCalls* to cudnn convolutions, choosing the best algorithm for each and adding explicit scratch space to the *CustomCalls*. Custom calls allow invoking code written in a programming language like C++ or CUDA from an XLA program (*tensorflow/compiler/xla/g3doc/custom_call.md*)

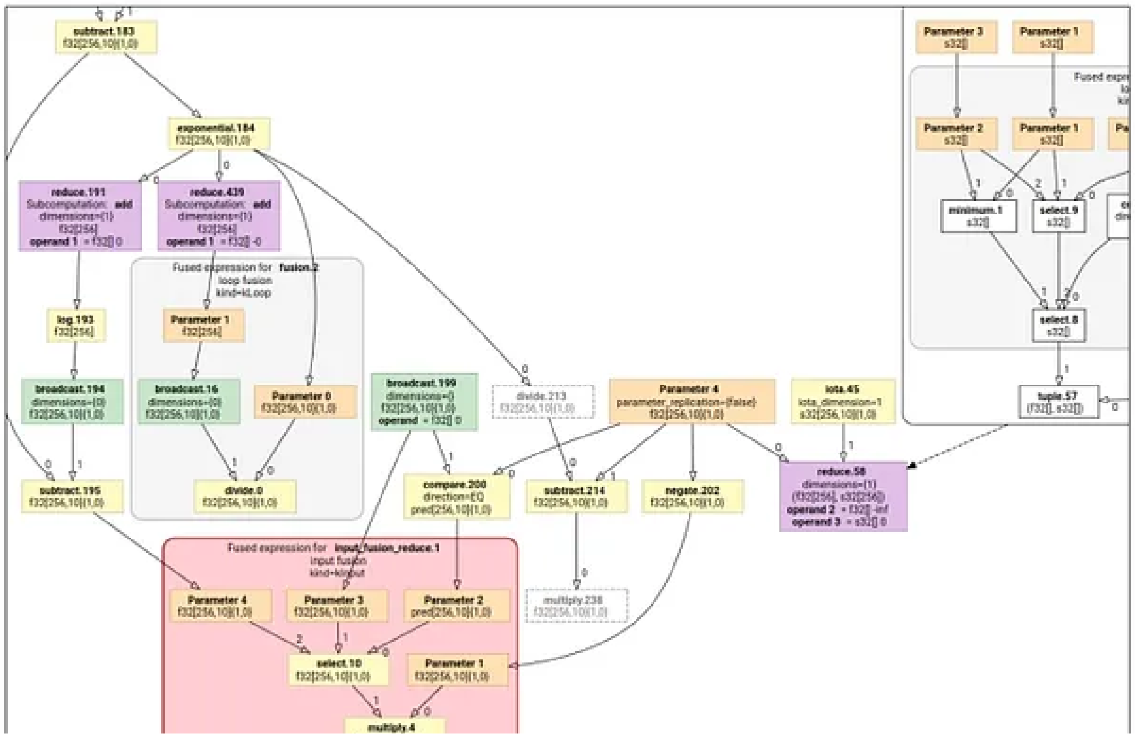
The *cudnn_vectorize_convolutions.h* pass changes the shape of cudnn convolutions to use faster “vectorized” algorithms as follows.

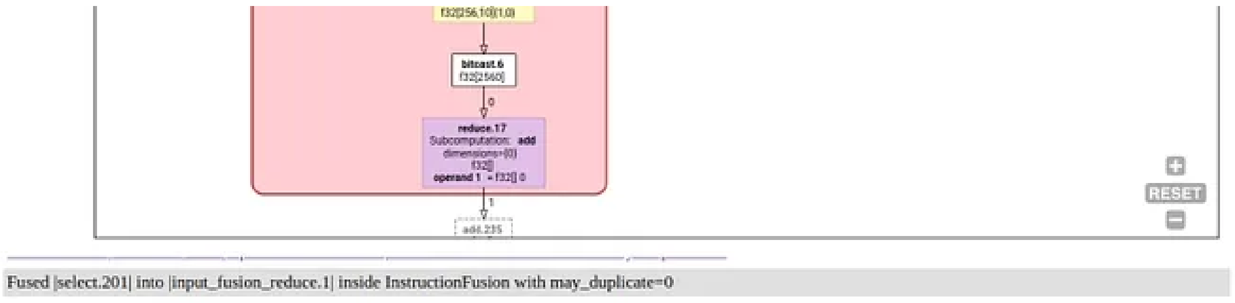
```
// On sm61+ will convert int8_t convolutions from
//
//   - [N, C, H, W] to [N, C/4, H, W, 4],
//
// assuming C is divisible by 4.
//
// On sm75+ will convert int8_t convolutions from
//
//   - [N, C, H, W]      to [N, C/32, H, W, 32],
//   - [N, C/4, H, W, 4] to [N, C/32, H, W, 32], and
//   - [N, C, H, W]      to [N,  C/4, H, W,  4] (same as sm61+),
//
// assuming C is divisible by 4 or 32.
```

The *instruction_fusion.h* pass performs, as the name suggests, HLO instruction fusion. Instructions are fused “vertically”, meaning producing instructions are fused into their consumers with the intent that the loops which compute their values will be fused in code generation. Derived classes define *ShouldFuse* method to select which instructions to fuse. The following TensorFlow debug html (see Appendix 2:Generating logs and graphs, on how these were generated) output shows one such example.



About to fuse |select.201| into |input_fusion_reduce.1| inside InstructionFusion with may_duplicate=0





Fused |select.201| into |input_fusion_reduce.1| inside InstructionFusion with may_duplicate=0

Fig 3: XLA HLO instruction fusion .

The *fusion_merger.h* pass attempts to merge fusion instructions to reduce memory bandwidth requirements and kernel launch overhead. Which is one the major goals of XLA. See the example below on what this does.

```
On the left-hand side, op A is the producer and ops B and C are its
consumers. FusionMerger duplicates producer ops and fuses them into
all consumers. The result is depicted on the right-hand side below.

          p                    p
          |                  /   \
          v                 /     \
          A            +fusion+  +fusion+
        /   \          |  A'   |  |  A"   |
       |     |         |  |    |  |  |    |
       v     v         |  v    |  |  v    |
       B     C         |  B    |  |  C    |
                       +-------+  +-------+

Op A has been cloned twice and fused with B and C. The kernel launch
overhead is reduced from 3 to 2. The memory bandwidth requirements
may be reduced. We trade 1 read of input(A) + 1 write and 2 reads of
output(A) for 2 reads of input(A). In general the achievable savings
in memory bandwidth depend on the differences in memory read and
written and the number of consumers. The FusionMeger pass takes this
into account when making fusion decisions.

The pass traverses the HLO module in reverse post-order (defs before
uses). Fusion instructions are merged into their users if some
conditions are met:
* The result of merging the fusion instruction into its users would
not increase bytes transferred.
* Producer ops are fusible with _all_ consumers. If they are not
fusible with at least one consumers, they won't be fused at all.
* Producers are kLoop fusion ops. None of these restrictions are
```

```
necessary for correctness. In fact, lifting the latter two could be
beneficial.
```

Glossary:

tf.Module — In TensorFlow, most high-level implementations of layers and models, such as Keras or Sonnet, are built on the same foundational class: tf.Module . Modules and, by extension, layers are deep-learning terminology for "objects": they have internal state, and methods that use that state.

Graph is a C++ structure defined in *tensorflow/core/graph/graph.h* . Node ans Edges are parts of the graph. 31 different types of NodeClass are defined as shown below.

```
— NC_UNINITIALIZED, NC_SWITCH, NC_MERGE, NC_ENTER, NC_EXIT,
NC_NEXT_ITERATION, NC_LOOP_COND, NC_CONTROL_TRIGGER, NC_SEND,
NC_HOST_SEND, NC_RECV, NC_HOST_RECV, NC_CONSTANT, NC_VARIABLE,
NC_IDENTITY, NC_GET_SESSION_HANDLE, NC_GET_SESSION_TENSOR,
NC_DELETE_SESSION_TENSOR, NC_METADATA, NC_SCOPED_ALLOCATOR,
NC_COLLECTIVE, NC_FAKE_PARAM, NC_PARTITIONED_CALL, NC_FUNCTION_OP,
NC_SYMBOLIC_GRADIENT,NC_IF, NC_WHILE, NC_CASE, NC_ARG, NC_RETVAL,
NC_OTHER,
```

GraphDef is a serialized version of Graph. This is the low-level definition of a TensorFlow computational graph, including a list of nodes and the input and output connections between them. During training of Python model, this is usually saved out in the same directory as checkpoints, and usually has a '.pb' or '.pbtxt' extension. Should not need to use GraphDefs directly in TF2. To load GraphDefs in TF2, SavedModel can be used.

SaverDef , a protocol buffer for saver object. A Saver class saves and restores variables to and from checkpoints.

MetaGraphDef is a protobuf containing meta information regarding the graph to be exported. It contains GraphDef and SaverDef

pb is the file extension for protocol buffer format files. Tensorflow Saved Models on disk are in this format.

SavedModel contains a complete TensorFlow program, including trained parameters (i.e, tf.Variables) and computation. SavedModel contains the GraphDef, and does not require the original model code to run. SavedModel is a directory containing serialized signatures and the state needed to run them, including variable values and vocabularies. The saved_model.pb file stores the actual TensorFlow program, or model, and a set of named signatures, each identifying a function that accepts tensor inputs and produces tensor outputs. SavedModels may contain multiple variants of the model (multiple v1.MetaGraphDefs, identified with the — tag_set flag to saved_model_cli). TensorFlow Serving uses SavedModel to run inference.

Freezing model refers to converting the checkpoint values into embedded constants within the graph file itself. Takes GraphDef proto, SaverDef proto, and a set of variable values stored in a checkpoint file, and output a GraphDef with all of the variable ops converted into const ops containing the values of the variables.

Grappler is the default graph optimization system in the TF runtime. Grappler applies optimizations in graph mode (within tf.function) to improve performance of TensorFlow computations through graph simplifications and other high-level optimizations such as inlining function bodies to enable inter-procedural optimizations. In the code it is called MetaOptimizer.

Gradient tape is used to differentiate automatically. TensorFlow needs to remember what operations happen in what order during the forward pass. During the backward pass, TensorFlow traverses this list of operations in reverse order to compute gradients. TensorFlow "records" operations executed inside the context of a tf.GradientTape onto a "tape".

AutoGraph is a library that is on by default in tf.function, and transforms a subset of Python eager code into graph-compatible TensorFlow ops — if, for, while. TensorFlow control flow ops like tf.cond and tf.while_ exists, but it is easier to write and understand in Python.

stream — performs actions with a linear stream of dependencies.

swig — Simplified Wrapper and Interface Generator for C/C++ code for languages like Python.

Affine — An affine function is linear — (e.g. 2*x+10 is affine, but not x^2+10) . Affine Loops are where the loop bounds and array references are affine functions of loop iterators and program parameters. The MLIR polyhedral loop dialect is called affine.

in programs. Polyhedral model can be used to reason about Affine Loops.

MLIR — Multi-level Intermediate Representation. MLIR dialects define operations, types, type constraints, rewrite rules, lowerings, etc. Has 14 builtin dialects. Examples of dialects supported by MLIR are Affine dialect, Func dialect, GPU dialect, LLVM dialect, SPIR-V dialect, Vector dialect , etc.

MLIR handles parsing, type checking/inference, line-number tracking, etc. MLIR also provides tools for testing and parallel compilation, documentation, CLI, etc. MLIR allows for multiple dialects, even those outside of the main tree, to co-exist together within one module. Dialects are produced and consumed by certain passes. MLIR provides a framework to convert between, and within, different dialects.

MLIR Module [10] — A module represents a top-level container (for IR) operation. It contains a single graph region containing a single block which can contain any operations and does not have a terminator. Operations within this region cannot implicitly capture values defined outside the module, i.e. Modules are *IsolatedFromAbove*. Modules have an optional symbol name which can be used to refer to them in operations.

MLIR Block [10] — list of operations

LLVM — is an Intermediate Representation (IR) that is usually considered a backend.

HLO — High Level Operation

MHLO [8,9] — MLIR HLO dilect

MLIR-HLO — A standalone “HLO” MLIR-based Compiler https://github.com/tensorflow/tensorflow/tree/v2.9.1/tensorflow/compiler/mlir/hlo

TensorFlow IR — MLIR dialect to represent TensorFlow graphs.

LHLO — Late MHLO variant of MLIR dialect, that operates on buffers instead of tensors.

XLA HloInstruction — XLA Input IR

Linalg — the linear algebra dialect of MLIR.

Autograd can automatically differentiate native Python and Numpy code

JAX is Autograd and XLA, brought together for high-performance machine learning research. JAX uses XLA to compile and run NumPy programs on GPUs and TPUs. JAX provides an implementation of NumPy (with a near-identical API) that works on both GPU and TPU. JAX provides functions transformations — grad() through Autograd, jit() through XLA, vmap() - vectorization, pmap() — SPMD parallelization on TPUs.

PartitionedCall represents asynchronously executing a function across multiple devices

PTX (Parallel Thread Execution) — PTX is a programming model and instruction set (ISA) for general purpose parallel programming on Nvidia. High level language compilers for CUDA and C/C++ generate PTX instructions

Thunk — (from tensorflow/compiler/xla/service/gpu/thunk.h) Thunk acts as the bridge between IrEmitter and GpuExecutable. It stores the metadata IrEmitter generates for GpuExecutable to invoke an HloInstruction. Thunk provides the Initialize and ExecuteOnStream interface for GpuExecutable to initialize and execute the invocation respectively. Its subclasses are supposed to override these interfaces to launch a generated kernel or call an external library function (such as operations in cuBLAS). Is runs on CPUs.

Cubin — CUDA binary (cubin) is an ELF-formatted file consisting of CUDA executable code as well as other sections containing symbols, relocators, debug info, etc. By default, the CUDA compiler nvcc embeds cubin files into the host executable file. They can also be generated separately by using the "-cubin" option of nvcc. cubin files are loaded at run time by the CUDA driver API.

Fatbin — CUDA fat binary file that contains multiple PTX and CUBIN files. A host compilation synthesis step embeds the fatbinary and transform CUDA specific C++ extensions into standard C++ constructs. The C++ host compiler eventually compiles the synthesized host code with the embedded fatbinary into a host object.

StreamExecutor — manages a single device, in terms of executing work (kernel launches) and memory management (allocation/deallocation, memory copies to and from the device). It is conceptually the "handle" for a device — Stream objects, which are used to enqueue work to run on the coprocessor, have a StreamExecutor instance as their "parent" object.

Appendices:

Appendix 1: Building TensorFlow debug version

A debug version of TensorFlow is essential to explore the code base. A wheel for debug version is not readily available, hence needs to be built. Debug version build requires more memory during the bazel build linking stage, hence increasing swap space helps. The follow shows the commands for a conda setup

```
cuda driver : 470.129.06 ( Cuda 11.4)

conda create -n tfbuild python=3.9

conda install cudatoolkit-dev=11.4.0 cudatoolkit=11.4.1 cuda-
nvcc=11.4.120 cudnn=8.2.1.32 gcc=9.4.0 gxx=9.4.0

bazel — output_base=./bazelout/outputs build — jobs 2 —
local_ram_resources=4096 — repository_cache=./bazelout/cache —
verbose_failures — config=dbg — config=cuda
//tensorflow/tools/pip_package:build_pip_package

./bazel-bin/tensorflow/tools/pip_package/build_pip_package
/tmp/tensorflow_pkg

source $CONDA_PREFIX/lib/python3.9/site-packages/libpython.py
```

Appendix 2:Generating logs and graphs

The following command was used for generating detailed logs and graph dumps. The .pbtxt format graphs can be viewed on Netron as long as they are not very large. Depending on the network model, the log file can be huge and number of graphs dumped can be large.

```
OUTPUTDIR=./outputs_tf_fmnst/wx_`date +%Y_%m_%d_%H_%m_%S`; mkdir -p
$OUTPUTDIR ; TF_XLA_FLAGS= — tf_xla_clustering_debug
TF_CPP_MAX_VLOG_LEVEL=5 TF_DUMP_GRAPH_PREFIX=$OUTPUTDIR/graphs
XLA_FLAGS=” — xla_gpu_cuda_data_dir=$CONDA_PREFIX/pkgs/cuda-toolkit —
xla_dump_to=$OUTPUTDIR/hlo — xla_dump_hlo_as_html —
xla_dump_fusion_visualization”
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$CONDA_PREFIX/lib python tf_fmnst.py
— epochs 1 — xla --profile 2>&1 | tee $OUTPUTDIR/output.log
```

Appendix 3: Running TensorFlow within gdb , breaking and backtracing.

```
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$VCONDA_PREFIX/lib  gdb --args
python  tf_fmnst.py --epochs 1
(gdb) b tensorflow/core/profiler/internal/gpu/cupti_tracer.cc:383
```

Appendix 4: Codes for network model and log analysis

The two main python files used, tf_fmnst.py and analyze_log.py, are shown below.

```
'''
OUTPUTDIR=./outputs_tf_fmnst/wx_`date +%Y_%m_%d_%H_%m_%S`; mkdir -p
$OUTPUTDIR ; TF_XLA_FLAGS=--tf_xla_clustering_debug
TF_CPP_MAX_VLOG_LEVEL=5 TF_DUMP_GRAPH_PREFIX=$OUTPUTDIR/graphs
XLA_FLAGS="--xla_gpu_cuda_data_dir=$CONDA_PREFIX/pkgs/cuda-toolkit --
xla_dump_to=$OUTPUTDIR/hlo --xla_dump_hlo_as_html --
xla_dump_fusion_visualization"
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$VCONDA_PREFIX/lib  python
tf_fmnst.py --epochs 1  --xla 2>&1 | tee $OUTPUTDIR/output.log
'''

import tensorflow as tf
import argparse
import timeit
import time
import pandas as pd

parser = argparse.ArgumentParser(description='To show xla flow.')
parser.add_argument("--epochs", type=int, default=10, help="maximum
number of epochs to run. ")
parser.add_argument("--iters", type=int, default=4, help="maximum
number of iterations per epoch to run. ")
parser.add_argument("--xla", action='store_true', default=False,
help="enable xla")
parser.add_argument("--warmup", action='store_true', default=False,
help="run warmup")
parser.add_argument('-f', type=str, help='log file')
parser.add_argument('--logdir', type=str, help='log directory')
parser.add_argument('--bs', type=int, default=256, help='batch size')
parser.add_argument("--profile", action='store_true', default=False,
help="run profiler")
args = parser.parse_args()

# Check that GPU is available: cf.
https://colab.research.google.com/notebooks/gpu.ipynb
```

```
assert(tf.test.gpu_device_name())

tf.keras.backend.clear_session()

if args.xla:
 #tf.config.optimizer.set_jit(True)
 tf.config.optimizer.set_jit("autoclustering")
else:
 tf.config.optimizer.set_jit(False)

def load_data():
  (x_train, y_train), (x_test, y_test) =
tf.keras.datasets.cifar10.load_data()
  x_train = x_train.astype('float32') / args.bs
  x_test = x_test.astype('float32') / args.bs

# Convert class vectors to binary class matrices.
  y_train = tf.keras.utils.to_categorical(y_train, num_classes=10)
  y_test = tf.keras.utils.to_categorical(y_test, num_classes=10)
  return ((x_train, y_train), (x_test, y_test))

def generate_model():
  return tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(32, (3, 3), padding='same',
input_shape=x_train.shape[1:]),
    tf.keras.layers.Activation('relu'),
    tf.keras.layers.Conv2D(32, (3, 3)),
    tf.keras.layers.Activation('relu'),
    tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
    tf.keras.layers.Dropout(0.25),

tf.keras.layers.Conv2D(64, (3, 3), padding='same'),
    tf.keras.layers.Activation('relu'),
    tf.keras.layers.Conv2D(64, (3, 3)),
    tf.keras.layers.Activation('relu'),
    tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
    tf.keras.layers.Dropout(0.25),

#tf.keras.layers.Flatten(input_shape=x_train.shape[1:]),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(512),
    tf.keras.layers.Activation('relu'),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(10),
    tf.keras.layers.Activation('softmax')
  ])

class DebugCallback(tf.keras.callbacks.Callback):
    def on_train_batch_begin (self, batch, logs=None):
        print (f"======= batch {batch} begin =========")
    def on_train_batch_end (self, batch, logs=None):
        print (f"======= batch {batch} end =========")

def compile_model(model):
  opt = tf.keras.optimizers.RMSprop(lr=0.0001, decay=1e-6)
  model.compile(loss='categorical_crossentropy',
                optimizer=opt,
                metrics=['accuracy'])
  return model

def train_model(model, x_train, y_train, x_test, y_test, epochs=25):
  model.fit(x_train, y_train, batch_size=args.bs, epochs=epochs,
verbose=2,
            validation_steps=0, validation_data=(x_test, y_test),
shuffle=True,
            callbacks=[DebugCallback()])

def warmup(model, x_train, y_train, x_test, y_test):
  # Warm up the JIT, we do not wish to measure the compilation time.
  initial_weights = model.get_weights()
  train_model(model, x_train, y_train, x_test, y_test, epochs=1)
  model.set_weights(initial_weights)

(x_train, y_train), (x_test, y_test) = load_data()
model = generate_model()
model = compile_model(model)
model.summary()

if args.warmup:
        warmup(model, x_train, y_train, x_test, y_test)

data_size = args.bs*args.iters
time0 = time.time()
if args.profile:
 tf.profiler.experimental.start(args.logdir+"/profile")
train_model(model, x_train[:data_size], y_train[:data_size], x_test,
y_test, epochs=args.epochs)
if args.profile:
 tf.profiler.experimental.stop()
print ('Train time (s):', time.time() - time0)

if False: #  args.xla:
 print ( "hlo ir:")
 print(generate_model.experimental_get_compiler_ir(x,y,z,zz)
(stage='hlo'))
 print ( "optimized hlo ir:")
 print(generate_model.experimental_get_compiler_ir(x,y,z,zz)
(stage='optimized_hlo'))
 print ( "optimized hlo dot:")
 print(generate_model.experimental_get_compiler_ir(x,y,z,zz)
(stage='optimized_hlo_dot'))
#scores = model.evaluate(x_test, y_test, verbose=1)
#print('Test loss:', scores[0], 'Test accuracy:', scores[1])

'''
# We need to clear the session to enable JIT in the middle of the
program.
tf.keras.backend.clear_session()
tf.config.optimizer.set_jit(True) # Enable XLA.
model = compile_model(generate_model())
(x_train, y_train), (x_test, y_test) = load_data()

warmup(model, x_train, y_train, x_test, y_test)
time0 = time.time()
train_model(model, x_train, y_train, x_test, y_test, epochs=EPOCHS)
print ('Train time (s):', time.time() - time0)
'''
```

Code: tf_mnst.py

```
import argparse
import json
import copy
from datetime import datetime, timezone


parser = argparse.ArgumentParser(description='Process log file.')

parser.add_argument('-f', type=str, help='log file')
parser.add_argument('--splitsec', type=float, default=0, help='split
at seconds since start ')
parser.add_argument('--tfpf', type=str, default=None, help='TF trace
file')
parser.add_argument("--its", action='store_true', default=False,
help="use iine number as timestamp")

args = parser.parse_args()

print("args: ", args)

def load_tf_trace(tracef, time_delta):
    tfpf=None
    if tracef is None:
        return tfpf
    with open(tracef) as f:
        tfpf=json.load(f)
    for te in tfpf['traceEvents']:
        if 'ts' in te.keys():
            te['ts'] = time_delta + te['ts']
    return tfpf

def find_start_token(l):
        ltoks = l.split()
        starttok = -1
        for i,tok in enumerate(ltoks):
            if tok.startswith ('tensorflow/'):
                starttok = i
                break
            else:
                pass
        return starttok, ltoks


def runlength_encoding( lines):
    encoded = []
    runtok = ""
    runlength = 0
    for l in lines:

        starttok,ltoks = find_start_token(l)
        if starttok == -1: continue

        if ltoks[starttok] == runtok:
            runlength +=1
        else:
            encoded.append( (runtok, runlength))
            runtok = ltoks[starttok]
            runlength = 1

    return encoded[1:] # removing the first dummy line


def load_log( lines):
    encoded = []
    runtok = ""
    runlength = 0
    startts = 0
    splitts = -1
    splitline = -1
    for idx, l in enumerate(lines):

        starttok,ltoks = find_start_token(l)
        if starttok == -1: continue

        naive = datetime.strptime(" ".join(ltoks[:2]), "%Y-%m-%d
%H:%M:%S.%f:")
        ts = naive.timestamp()*1.0e6
        if startts == 0:
            startts = ts

        if ltoks[starttok] == runtok:
            runlength +=1
        else:
            runtok = ltoks[starttok]
            runlength = 1

        encoded.append( (ltoks[starttok], runlength, ts, idx, "
".join(ltoks[:2]+ltoks[4:])  ) )

        if splitts == -1  and (ts - startts ) >=  (args.splitsec*1e6)
:
```

```
            splitts = ts
            splitline = len(encoded) -1 # index starts at 0

    return encoded[1:], startts, splitts, splitline # removing the
first dummy line

def runlength_encoding( lines):
    encoded = []
    runtok = ""
    runlength = 0
    for l in lines:

        starttok,ltoks = find_start_token(l)
        if starttok == -1: continue

        if ltoks[starttok] == runtok:
            runlength +=1
        else:
            encoded.append( (runtok, runlength))
            runtok = ltoks[starttok]
            runlength = 1

    return encoded[1:] # removing the first dummy line

def unique_count (lines):
    call_dict = {}
    for l in lines:

        starttok,ltoks = find_start_token(l)
        if starttok == -1: continue

        if ltoks[starttok]  not in call_dict.keys():
            call_dict[ltoks[starttok]] = 1
        else:
            call_dict[ltoks[starttok]] += 1

    return call_dict

def create_log_trace (tsel, marker=None):
    tracedict = {"traceEvents":[],
                 "meta_user": "sg",
                }

    multiplier = 1000
    its = 0
    tsdbegin = None
    tsdend   = None

    for tse in tsel:
        fpath,linenum = tse[0].split(":")
        if "optimization of a group" in tse[-1] or "Running
```

```
optimization " in tse[-1]:
            ph = "i"
            s = "g"
        else:
            ph = "X"
            s = None

        if not args.its:
            ts = tse[2]
        else:
            ts = its*multiplier
        fpathl = fpath.split("/")

        if len (fpathl) >3 :
            pid = "/".join(fpathl[:3])
            pid += " "*30
            #tid = fpath
            tid = "/".join(fpathl[3:])
        else:
            pid = fpath

        linerest = " ".join([str(v) for v in tse[3:]])
        linerest_max = 100
        linerest = linerest[:linerest_max] if len (linerest) >
linerest_max else linerest
        tsd = {"pid": pid,
               "tid": tid, #linenum[:-1],
               "ts": ts,
               #"dur":tse[1]*multiplier,
               "dur":multiplier,
               "ph":ph,
               "s":s,
               "name": tse[0][:-1],
               "args": {#"linenum":linenum[:-1],
                        #"seqid":its,
                                                         "runlength":
tse[1],
                        "linerest":linerest},
              }
        #ts += tse[1]
        its += 1
        tracedict["traceEvents"].append(tsd)
        tsdend = tsd
        if tsdbegin is None:
            tsdbegin = tsd

    if marker == "end":
        tsdend = copy.deepcopy(tsdend)
        tsdend["ph"] = "i"
        tsdend["s"] = "g"
        tsdend["name"] = "End of log"
        tsdend["args"] = {}
        tracedict["traceEvents"].append(tsdend)
    elif marker == "begin":
        tsdbegin = copy.deepcopy(tsdbegin)
        tsdbegin["ph"] = "i"
        tsdbegin["s"] = "g"
        tsdbegin["name"] = "Begining of log"
        tsdbegin["args"] = {}
        tracedict["traceEvents"].append(tsdbegin)


    return tracedict


with open(args.f) as f:
    lines = f.readlines()

call_dict = unique_count (lines)
print ( "======unique calls: ", len(call_dict.keys()))
for k in call_dict.keys():
    print (k, call_dict[k])

'''
runlength  = runlength_encoding( lines)
print ("=======runlength calls: ", len(runlength))
for l in runlength:
    print ( l )

print ("summary: ", len(call_dict.keys()) , len(runlength))

jsonf = args.f+".runlength.json"
crate_trace(runlength, jsonf)
'''

logfullline,startts, splitts,splitline = load_log(lines)
tft = load_tf_trace(args.tfpf, startts)
for si in [(0,splitline, "end"), (splitline+1,len(logfullline),
"begin")]:
    if si[0] == si[1]:
        continue
    splitlog = logfullline[si[0]:si[1]]
    logtrace = create_log_trace(splitlog, si[2])
    if tft is not None:
        tft['traceEvents'] =
tft['traceEvents']+logtrace['traceEvents']
    else:
        tft = logtrace['traceEvents']
    jsonf = args.f+f"lines_{si[0]}_{si[1]}.merged.json"
    print("writting to file ", jsonf)
    with open ( jsonf, "w") as fp:
        json.dump(tft,fp)
```

```
print ("done")
```

Code: analyze_log.py

Appendix 5: Grappler Groups, Phases and Passes

```
Starting optimization of a group 0 — PRE_PLACEMENT
Running optimization phase 0
Running optimization pass: MlirV1CompatGraphOptimizationPass
(compiler/mlir/mlir_graph_optimization_pass.h)
Running optimization phase 9
Running optimization pass: ControlFlowDepsToChainsPass
(core/common_runtime/control_flow_deps_to_chains.h)
Running optimization phase 10
Running optimization pass: AccumulateNV2RemovePass
(core/common_runtime/accumulate_n_optimizer.cc)
Running optimization pass: LowerFunctionalOpsPass
(core/common_runtime/lower_functional_ops.h)
Running optimization pass: ParallelConcatRemovePass
(core/common_runtime/parallel_concat_optimizer.cc)
Running optimization phase 35
Running optimization pass: IsolatePlacerInspectionRequiredOpsPass
(core/common_runtime/isolate_placer_inspection_required_ops_pass.h)
Running optimization pass: IntroduceFloatingPointJitterPass
(compiler/jit/introduce_floating_point_jitter_pass.h)
Running optimization phase 36
Running optimization pass: EncapsulateXlaComputationsPass
(compiler/jit/encapsulate_xla_computations_pass.h)
maxwell_scudnn_winograd_128x128_ldg1_ldg4_relu_tile228n_nt_v1 Running
optimization phase 37
Running optimization pass: FunctionalizeControlFlowForXlaPass
(compiler/tf2xla/functionalize_control_flow.h)
Running optimization phase 99999
Running optimization pass: WeakForwardTypeInferencePass
(core/common_runtime/forward_type_inference.h)
Starting optimization of a group 1 — POST_PLACEMENT
Running optimization phase 0
Running optimization pass: NcclReplacePass
(core/nccl/nccl_rewrite.cc)
Starting optimization of a group 2 — POST_REWRITE_FOR_EXEC
Running optimization phase 5
Running optimization pass: CloneConstantsForBetterClusteringPass
(compiler/jit/clone_constants_for_better_clustering.h)
Running optimization phase 9
Running optimization pass: ClusterScopingPass
```

```
(compiler/jit/cluster_scoping_pass.h)
Running optimization phase 10
Running optimization pass: MarkForCompilationPass
(compiler/jit/mark_for_compilation_pass.h)
Running optimization phase 12
Running optimization pass: ForceXlaConstantsOnHostPass
(compiler/jit/force_xla_constants_on_host_pass.h)
Running optimization phase 20
Running optimization pass: IncreaseDynamismForAutoJitPass
(compiler/jit/increase_dynamism_for_auto_jit_pass.h)
Running optimization phase 30
Running optimization pass: PartiallyDeclusterPass
(compiler/jit/partially_decluster_pass.h)
Running optimization phase 40
Running optimization pass: ReportClusteringInfoPass
(compiler/jit/report_clustering_info_pass.h)
Running optimization phase 50
Running optimization pass: EncapsulateSubgraphsPass
(compiler/jit/encapsulate_subgraphs_pass.h)
Running optimization phase 60
Running optimization pass: BuildXlaOpsPass
(compiler/jit/build_xla_ops_pass.h)
Starting optimization of a group 3 — POST_PARTITIONING
Running optimization phase 1
Running optimization pass: MklLayoutRewritePass
XLA- before_optimizations, after_optimization, after 4 and 8 th
```

References:

1. Classifying CIFAR-10 with XLA, https://www.tensorflow.org/xla/tutorials/autoclustering_xla

2. TensorFlow graph optimization with Grappler, https://www.tensorflow.org/guide/graph_optimization#:~:text=Grappler%20is%20the%20default%20graph,in%20graph%20mode%20(within%20tf

3. XLA debug: https://github.com/tensorflow/tensorflow/blob/v2.9.1/tensorflow/compiler/xla/debug_options_flags.cc

4. Nvidia Xla Best Practices , https://docs.nvidia.com/deeplearning/frameworks/tensorflow-user-

guide/index.html#xla-best-practices

5. MLIR CodeGen for XLA https://www.tensorflow.org/mlir/xla_gpu_codegen
6. DLVM: A MODERN COMPILER INFRASTRUCTURE FOR DEEP LEARNING SYSTEMS, https://arxiv.org/pdf/1711.03016.pdf
7. APOLLO: AUTOMATIC PARTITION-BASED OPERATOR FUSION THROUGH LAYER BY LAYER OPTIMIZATION, https://proceedings.mlsys.org/paper/2022/file/069059b7ef840f0c74a814ec9237b6ec-Paper.pdf
8. ‘mhlo’ Dialect, https://www.tensorflow.org/mlir/hlo_ops
9. MLIR dialects , https://www.tensorflow.org/mlir/dialects
10. https://mlir.llvm.org/docs/Dialects/Builtin/ , https://mlir.llvm.org/docs/LangRef/
11. tensorflow/compiler/xla/service/hlo_instruction.h, tensorflow/compiler/xla/service/gpu/thunk.h, tensorflow/compiler/xla/service/hlo_opcode.h

Xla TensorFlow Llvm Computation Graph Nvcc