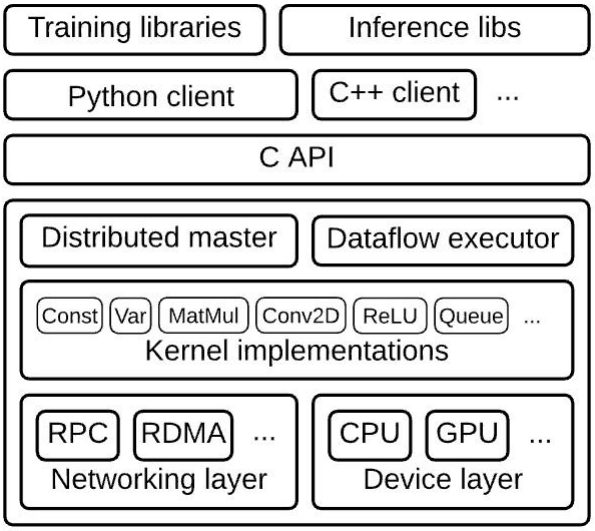

Figure 5: The layered TensorFlow architecture.

5 Implementation

We implement TensorFlow as an extensible, crossplatform library. Figure 5 illustrates the system architecture: a thin C API separates user-level in various languages from the core library. In this section, we discuss the implementation of the various components.

The core TensorFlow library is implemented in C++ for portability and performance: it runs on several operating systems including Linux, Mac OS X, Android, and iOS; the x86 and various ARM-based CPU architectures; and NVIDIA's Kepler, Maxwell, and Pascal GPU microarchitectures. The implementation is open-source, and we have accepted several external contributions that enable TensorFlow to run on other architectures.

The distributed master translates user requests into execution across a set of tasks. Given a graph and a step definition, it prunes (§3.2) and partitions (§3.3) the graph to obtain subgraphs for each participating device, and caches these subgraphs so that they may be re-used in subsequent steps. Since the master sees the overall computation for a step, it applies standard optimizations such as common subexpression elimination and constant folding; pruning is a form of dead code elimination. It then coordinates execution of the optimized subgraphs across a set of tasks.

The dataflow executor in each task handles requests from the master, and schedules the execution of the kernels that comprise a local subgraph. We optimize the dataflow executor for running large, fine-grained graphs with low overhead; our current implementation dispatches approximately 2,000,000 null operations per second. The dataflow executor dispatches kernels to local devices and runs kernels in parallel when possible: e.g., by using multiple cores in a CPU device, or multiple streams on a GPU.

The runtime contains over 200 standard operations, including mathematical, array

manipulation, control flow, and state management operations. Many of the operation kernels are implemented using Eigen::Tensor [34], which uses C++ templates to generate efficient parallel code for multicore CPUs and GPUs; however, we liberally use libraries like cuDNN [13] to implement kernels where a more efficient specialization is possible. We have also implemented support for quantization, which enables faster inference in environments such as mobile devices and high-throughput datacenter applications, and use the gemmlowp low-precision matrix multiplication library [33] to accelerate quantized computation.

We specialize Send and Recv operations for each pair of source and destination device types. Transfers between local CPU and GPU devices use thecudaMemcpyAsync() API to overlap computation and data transfer; transfers between two local GPUs use DMA to relieve pressure on the host. For transfers between tasks, TensorFlow supports multiple protocols, including gRPC over TCP, and RDMA over Converged Ethernet.
We are also investigating optimizations for GPU-to-GPU communication that use collective operations [57].
Section 4 describes features that we implement totally above the C API, in user-level code. Typically, users compose standard operations to build higher-level abstractions, such as neural network layers, optimization algorithms (§4.1), and sharded embedding computations (§4.2). TensorFlow supports multiple client languages, and we have prioritized support for Python and C++, because our internal users are most familiar with these languages. As features become more established, we typically port them to C++, so that users can access an optimized implementation from all client languages.

If it is difficult or inefficient to represent a subcomputation as a composition of operations, users can register additional kernels that provide an efficient implementation written in C++. We have found it profitable to hand-implement fused kernels for some performance critical operations, such as a the ReLU and Sigmoid activation functions and their corresponding gradients. We are currently investigating automatic kernel fusion using Halide [61] and other compiler-based techniques.

In addition to the core runtime, our colleagues have built several tools that aid users of TensorFlow. These include serving infrastructure for running inference in production, a visualization dashboard that enables users to follow the progress of a training run, a graph visualizer that helps users to understand the connections in a model, and a distributed profiler that traces the execution of a computation across multiple devices and tasks. We describe these tools in an extended whitepaper [1], and they can be downloaded from the project repository.