CS3423: Compilers - II

Mini-Assignment #5

Soumi Chakraborty

ES19BTECH11017

1 Summary of the given Links

1.1 A friendly introduction to machine learning compilers and optimizers

1.1.1 Cloud computing vs. Edge computing

ML models can be deployed on cloud services like AWS, but these models are computationally heavy, and computational resources are very expensive. To avoid such bills, companies try to do their computations on consumer/edge devices. Edge computing also allows models to run without a stable Internet connection. Edge computing also reduces network latency. Lastly, since edge computing doesn't require storing data on a cloud, it is safer to ensure there is no breach in the security.

1.1.2 Compiling: compatibility

Many companies are working towards building their own AI chips to reap the benefits of edge computing. Making sure an ML model is compatible with the hardware it will run on is a mammoth task as it requires complete understanding from both ends. Thus, framework developers provide support only for a few well know hardware infrastructures as making sure it will run on any kind of hardware smoothly is a very labour intensive task.

To deal with this bottleneck, frameworks are translated to an intermediate representation (IR) which can run on any type of hardware using a code generator like LLVM. This process is called lowering.

1.1.3 Optimizing: performance

All lowered code might run on the hardware of your choice, but it may not be the most optimised version that utilises all the available resources. Models that work well in development environments might run much slower during production, whose performance is then boosted by optimisation engineers. Compilers that optimize code are another option to boost performance.

1.1.4 How to optimize your ML models

ML models can be optimised locally (only optimise a few operations etc.) or globally (optimise the whole pipeline). Local optimisation techniques usually revolve around parallelisation of code. Some other methods are: vectorising, loop tiling and operator fusion. Global optimisation speeds up the pipeline much better than local optimisation. One such method is collapsing the computational graph along one direction by fusing operators.

1.1.5 Hand-designed vs. ML-based compilers

Optimization engineers decide the best way to execute computational graphs for a particular hardware infrastructure. These are hand-designed optimisation rules, they might neither be the most optimal solution nor very adaptive to a new framework. The ultimate goal is to optimize all kinds of computational graphs, not just that of one framework, by finding a suitable set of solutions to try and then pick the best one like autoTVM does. The only drawback of compilers like autoTVM is that the first time it runs, it takes hours to find the best optimisation.

1.1.6 Different types of compilers

Some popular compilers that target a certain combination of hardware infrastructure and framework: NVCC, XLA, PyTorch Glow - all of which are from very well known tech giants. Some good third party compilers are Apache TVM. MLIR is another project initiated by the creator of LLVM, and it is a tool that allows developers to build their own compilers. WebAssembly or WASM allows us to run models on browsers using JavaScript.

Conclusions

Running ML models on regular hardware can be very non-optimised, and research on ML-compilers is underway to come up with a generalised method to come up with the best optimisation of any ML model. Currently, such ML compilers try to search through a select few optimization methods, run tests and return the best method to use.

1.2 Machine Learning in compiler optimization

Despite software and hardware development proceeding with leaps and bounds in the last few years, the gap between them hasn't been bridged appropriately to get the best of both worlds. This is because bridging the gap requires one to have extensive knowledge of both the ML models and hardware infrastructures. Recently, efforts are being made to lessen this gap. However because hardware also is evolving rapidly, the concept of allowing ML to figure out the best way to optimize a compiler is becoming very popular.

The process described in this link is very similar to the process described in this link, which has been described in Section 1.1.

1.3 Graph compilers for artificial intelligence training and inference

With the advancement of many Deep Learning networks, graph compilation techniques are being studied extensively to find the best way to compile such models by using all resources available effectively.

DL models are represented as computational graphs which are needed to optimize hardware backends using various techniques like operator fusing, etc. Though DL frameworks are already based on optimised libraries, the final product is not the most optimised version since libraries do not support the new complex operators created in the NN.

2 MLIR

2.1 MLIR and its Dialects

MLIR can be confused to be a compiler, but in reality it allows us to build our own compilers to meet the needs of the ever-evolving ML models that we build, and reduces the need for domain-specific compilers. It is a hybrid IR.

MLIR dialects are tools which allow users to create their own compilers. They are used to define new operators, etc. Each operator associated with a particular dialect is prefixed with the name of the dialect it belongs to.

2.2 Need for different abstraction levels of IR

MLIR needs different abstraction levels so that it can efficiently convert any kind of ML model to an appropriate IR which can be lowered to run on any kind of hardware infrastructure universally. Having multiple abstraction levels of IRs ensures that if any one of them fails, another one of them will work as expected.