Computation Graph and Operator-level Optimizations

Lecture 11 for Advanced Deep Learning Systems

Aaron Zhao, Imperial College London, a.zhao@imperial.ac.uk

Table of contents

1. Introduction

2. A High-level Overview

3. The Computational Graph

4. The Operator-level Optimizations

Introduction

Computation graphs and operator-level optimizations

We have looked a few algorithmic optimizations, and they are mainly 'lossy' optimizations. Most of them rely on the redundancy of NNs and also the recovering power of SGDs.

On the compiler stack, we also have the opportunity to issue a range of classic lossless optimizations.

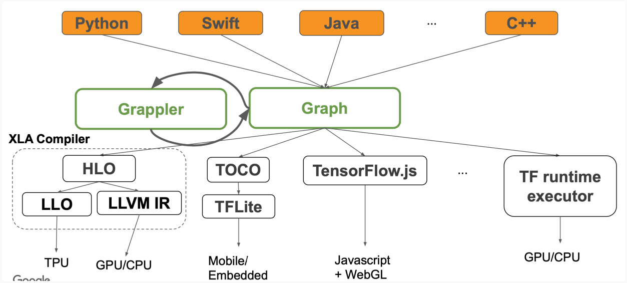
- Graph-level optimizations, we refer to a computation graph in this case, an example could be the MaseGraph.
- Operator-level optimizations, how do we perform lower-level optimization if given hardware information.

A High-level Overview

A High-level Overview

Grappler is designed for Tensorflow, to optimize the Graph.

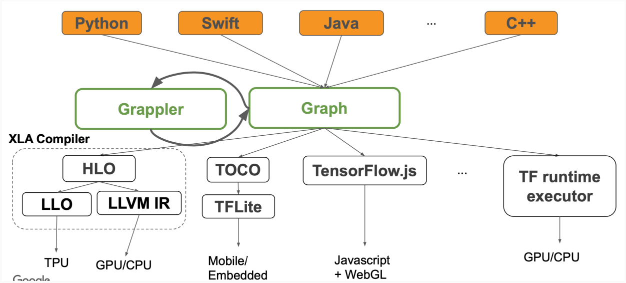
This is normally on the middleware level. Grappler takes various front-end languages (Python, Swift ... C++). The middleware obtains a Graph and we have to deploy this to various hardware backends.



A High-level Overview

These optimizations are normally at two levels:

- Graph-level
- Operator-level



The Computational Graph

Constant folding

Rewrites certain operations in the graph, if we know the values ahead-of-time.

Constant propagation

$Add(c1, Add(x, c2)) => Add(x, c1 + c2)$

Operations with neutral & absorbing elements

$x * Ones(s) => Identity(x), if\ shape(x) == output\_shape$

Graph Operation Fusion

Replaces commonly occurring subgraphs with optimized fused kernels

Examples of patterns fused:

- Conv2D + BiasAdd + Activation
- Conv2D + FusedBatchNorm + Activation
- Conv2D + Squeeze + BiasAdd
- MatMul + BiasAdd + Activation

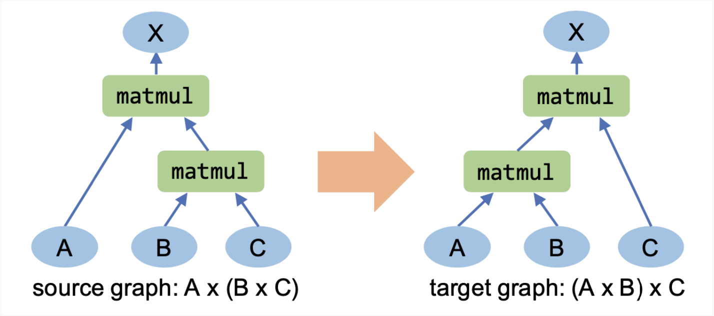
Graph Operation Fusion

Fusing ops together provides several performance advantages:

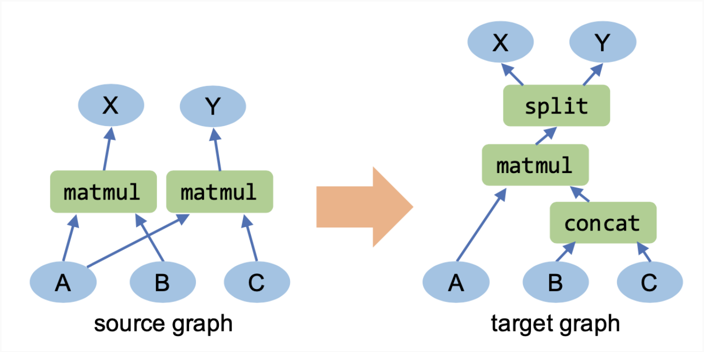
- Completely eliminates the scheduling overhead (great for cheap ops)
- Increases opportunities for ILP, vectorization etc.
- Improves temporal and spatial locality of data access. E.g. MatMul is computed block-wise and bias and activation function can be applied while data is still “hot” in cache.

Graph Operation Rewrites

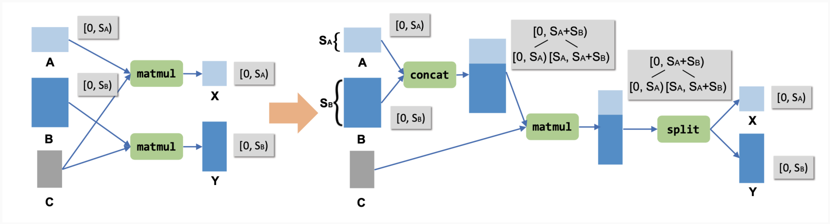
Rely on heuristics to repack operations on the graph level.



Graph Operation Rewrites



Graph Operation Rewrites



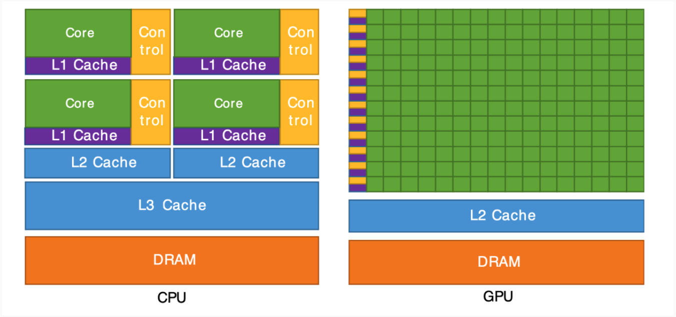
Graph-level Optimizations

Graph level optimizations are normally at a coarse-grained level.

More importantly, most of them have to rely on certain heuristics, for instance, this could be the re-write rules for fusion strategies.

The Operator-level Optimizations

Operator-level optimizations



On the graph-level, we care about large operations (more like layers).

On the operator level, this is now closer to the hardware. We typically prioritize the most compute-intensive tasks, thereby concentrating on loop manipulations.

Loop Tiling

The following is an example of matrix vector multiplication.

There are three arrays, each with 100 elements. The code does not partition the arrays into smaller sizes.

```
int i, j, a[100][100], b[100], c[100];
int n = 100;
for (i = 0; i < n; i++) {
  c[i] = 0;
  for (j = 0; j < n; j++) {
    c[i] = c[i] + a[i][j] * b[j];
  }
}
```

Loop Tiling

After loop tiling is applied using 2 * 2 blocks, the code looks like:

```c
int i, j, x, y, a[100][100], b[100], c[100];
int n = 100;
for (i = 0; i < n; i += 2) {
  c[i] = 0;
  c[i + 1] = 0;
  for (j = 0; j < n; j += 2) {
    for (x = i; x < min(i + 2, n); x++) {
      for (y = j; y < min(j + 2, n); y++) {
        c[x] = c[x] + a[x][y] * b[y];
      }
    }
  }
}
```

Loop Tiling

Loop tiling

- Tiling partitions a loop's iteration space into smaller chunks or blocks
- This ensures data used in a loop stays in the cache until it is reused.
- This leads to partitioning of a large array into smaller blocks, thus fitting accessed array elements into cache size, enhancing cache reuse and eliminating cache size requirements.

Loop Unrolling

```c
int x;
for (x = 0; x < 100; x++)
{
  dosomething(x);
}

for (x = 0; x < 100; x += 5 )
{
  dosomething(x);
  dosomething(x + 1);
  dosomething(x + 2);
  dosomething(x + 3);
  dosomething(x + 4);
}
```

Loop Unrolling

The amount of unrolling is normally associated with the underlying hardware parallelism.

This looks simple, but can become very complex if we have loop nests. It becomes more complex if we start to think about it in conjunction with tiling and other operations.

Loop Permutation for Data Layout Optimization

What is Data Layout?

Data layout optimization converts data into one that can use better internal data layouts for execution on the target hardware.

For instance, a DL accelerator might exploit 4×4 matrix operations, requiring data to be tiled into 4×4 chunks to optimize for access locality. A good data layout

- Improves locality
- Reduces the number of off-chip memory accesses
- May reduce Ops (eg. unnecessary transpose)

This is normally achieved by loop permutation.

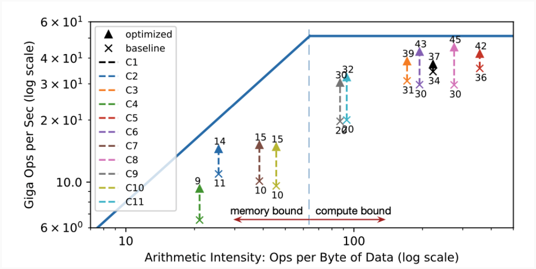
More on Loop Optimizations

- Loop Fusion/Fission: break large loop to smaller ones or fuse small loops, normally for locality.
- Loop Skewing (Polyhedral Optimization): Normally for deep nested loops, re-arrange loops to achieve a better access pattern.
- Loop Splitting: attempts to simplify a loop or eliminate dependencies by breaking it into multiple loops which have the same bodies but iterate over different portions of the index range.
- and more...

Why we doing this?

Loop-level manipulation is very complex, however, the core idea is to push the performance to the optimal point!

- Improve locality for memory-bound scenarios
- Improve parallelism for compute-bound scenarios



Arithmetic Operator Optimization

- Arithmetic simplification
  - Hoisting: $Add(x * a, x * bx * c) = x * Add(a, b, c)$
  - Node reduction: $x + x + x = 3x$, one op instead of two ops
- Broadcast minimization:
  $(matrix1 + scalar1) + (matrix2 + scalar2) =>$
  $(matrix1 + matrix2) + (scalar1 + scalar2)$
- Better use of intrinsics
  $Matmul(Transpose(x), y) => Matmul(x, y, transpose_x = True)$

Memory Operator Optimization

- Swap-in and Swap-out Optimizations: actively estimate memory usage and swap in/out idle data to host memory
- Recomputation optimization: if moving in/out the data is slow, why not just re-compute that value.

Summary

- The Computational Graph
    - Constant folding
    - Graph operation fusion
    - Graph operation rewrites
- The Operator-level Graph
    - Loop tiling
    - Loop unrolling
    - Loop permutation
    - Arithmetic operator optimization
    - Memory operator optimization