Computation Graph and Operator-level Optimization

Aaron Zhao, Imperial College London

Introduction

Computation graphs

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

Grappler

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.

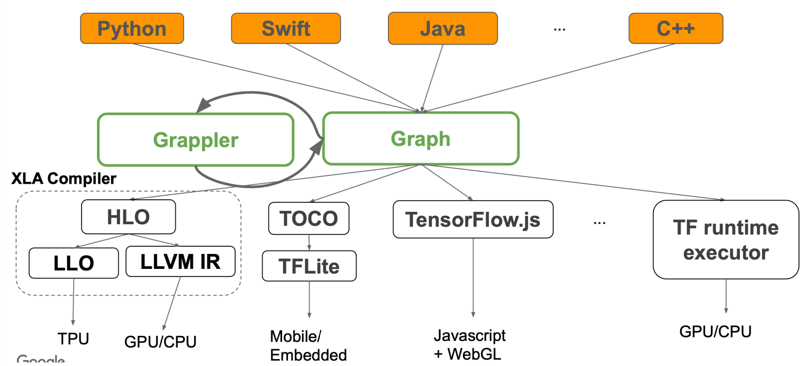


Figure 1: TensorFlow stack with Grappler optimization layer

A high-level overview

These optimizations are normally at two levels:

- Graph-level: High-level transformations on computation graphs
- Operator-level: Low-level optimizations closer to hardware

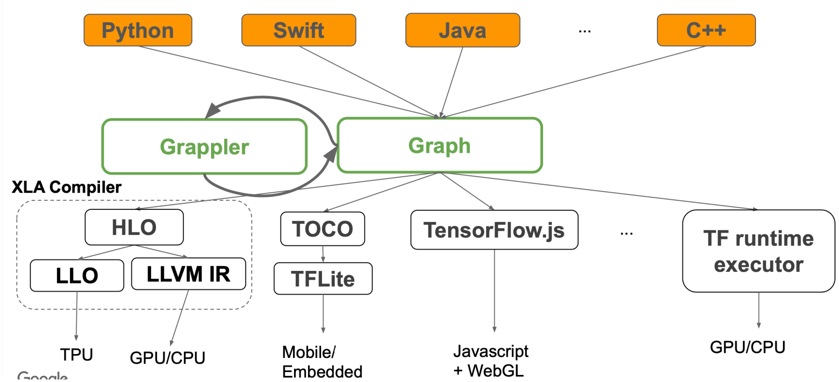


Figure 2: Two-level optimization approach in deep learning compilers

Optimizations

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

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

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.

Graph Operation Fusion (ii)

- 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.

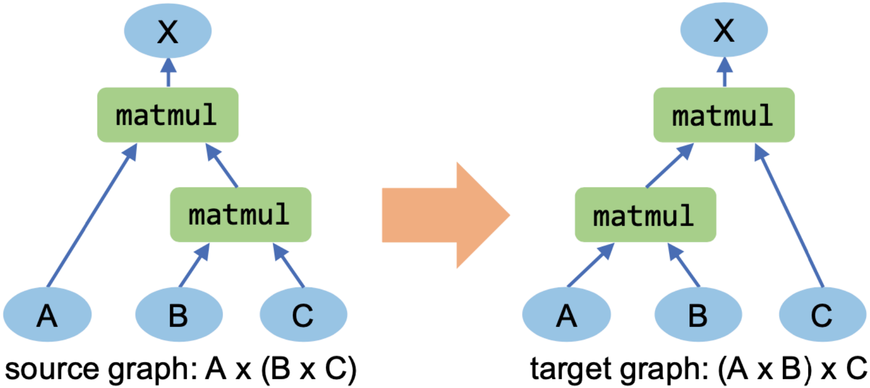


Figure 3: Exploiting associativity for graph rewrites

Graph Operation Rewrites (ii)

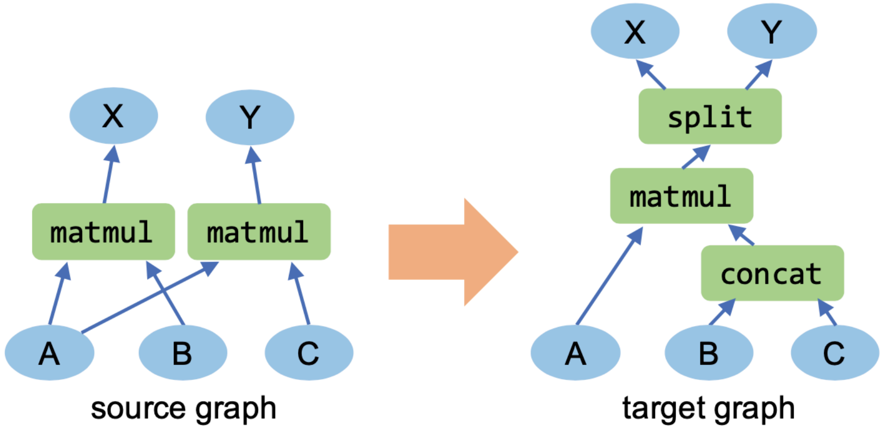


Figure 4: Eliminating redundant operations

Graph Operation Rewrites (iii)

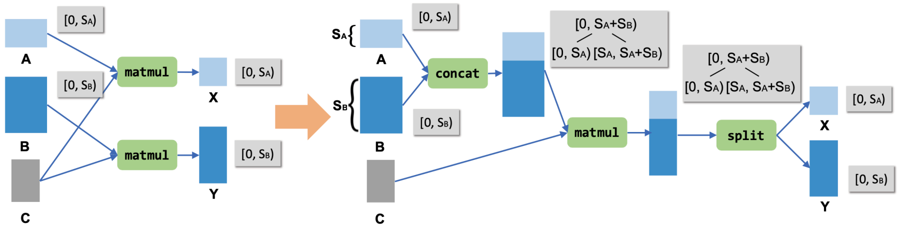


Figure 5: Example of graph rewrite transformations

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

Operator-level optimizations

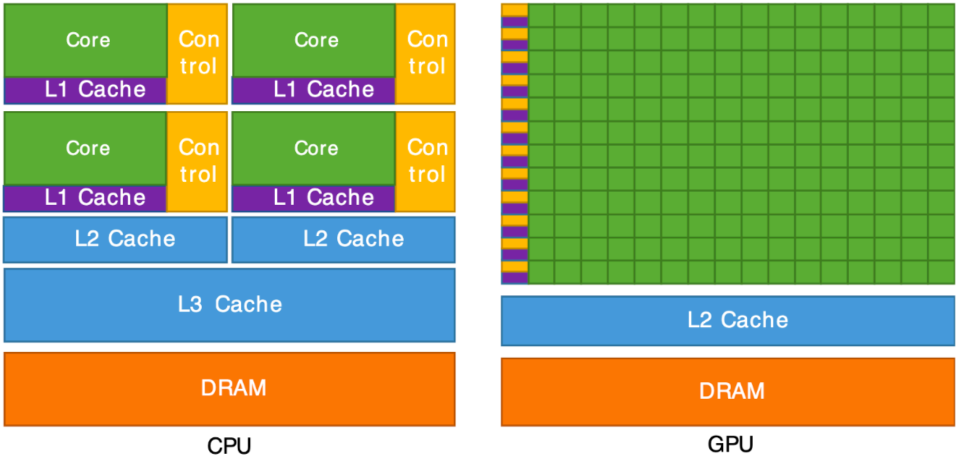


Figure 6: GPU memory hierarchy and compute units

Operator-level optimizations (ii)

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 mostly focus 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];
```

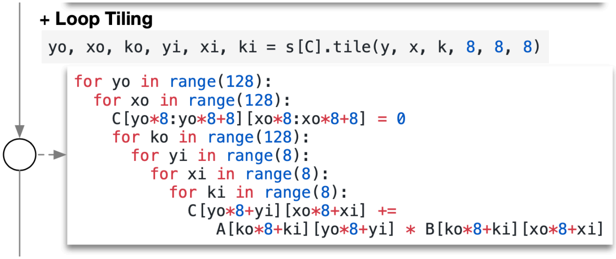
Loop tiling

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

```
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

- 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 Tiling

```
yo, xo, ko, yi, xi, ki = s[C].tile(y, x, k, 8, 8, 8)
```

```
for yo in range(128):
  for xo in range(128):
    C[yo*8:yo*8+8][xo*8:xo*8+8] = 0
    for ko in range(128):
      for yi in range(8):
        for xi in range(8):
          for ki in range(8):
            C[yo*8+yi][xo*8+xi] +=
               A[ko*8+ki][yo*8+yi] * B[ko*8+ki][xo*8+xi]
```

Figure 7: Visualization of loop tiling - dividing iteration space into blocks

Loop unrolling

```
int x;
for (x = 0; x < 100; x++):
  do_something(x);

for (x = 0; x < 100; x += 5):
  do_something(x);
  do_something(x + 1);
  do_something(x + 2);
  do_something(x + 3);
  do_something(x + 4);
```

The amount of unrolling is associated with the 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 and fission: this breaks 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 do 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

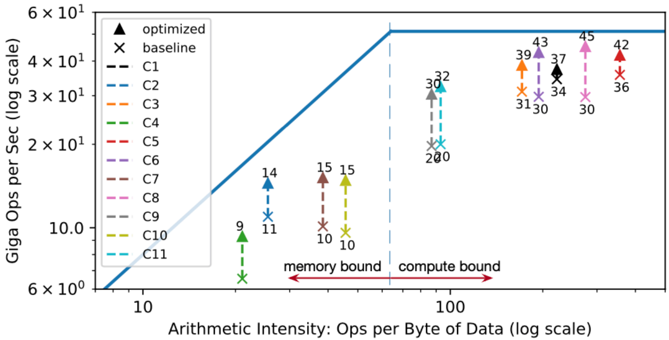


Figure 8: Roofline model showing performance optimization targets

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.

Automated optimization?

Automated low-level optimization was very popular in the past few years:

- **Auto-tuning frameworks**: TVM, Ansor, Halide
- **Polyhedral compilers**
- **Search-based approaches**: AutoTVM, AutoScheduler, Triton AutoTune

However, recent trends show:

- Manual optimization often outperforms automated approaches for critical kernels
- Hardware-specific optimizations require deep domain knowledge
- Flash Attention is a prime example of expert hand-crafted optimization

Flash Attention

Flash Attention: A case study in expert manual optimization

Key ideas:

- Tiling and recomputation to reduce memory access
- Leverages GPU memory hierarchy (SRAM vs HBM)
- Reduces memory access

Standard attention is memory-bound:

```
Q, K, V are loaded from HBM to SRAM
S = Q @ K.T computed and written to HBM
P = softmax(S) loaded from HBM
O = P @ V computed
```

Flash Attention optimizes this by:

1. **Tiling**: Process attention in blocks that fit in SRAM
2. **Recomputation**: Recompute attention scores instead of loading from HBM

Flash Attention (ii)

3. **Fused kernels**: Combine matmul + softmax + matmul in one kernel

Result: 2-4x faster than standard implementations, enables longer context lengths

This shows that understanding the hardware and algorithm together can beat automated optimization!

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