Woodpecker-DL
an efficient compiler for accelerating deep learning on heterogeneous computing architectures

Yong Chen
Ant Financial, China
Dec. 19, 2019
Woodpecker-DL Overview

Key Components and Technology

- Graph Optimization
- Auto Search with GA, RL Algorithm
- DSL Compiler (Halide)

Integration with TensorRT and Experiment Figures
Introduction

- Accelerating model training and inference is crucial to deep learning
  - Graph-level optimizations to increase inner-node compute efficiency and reduce inter-node data movement overhead
  - Operator-level optimizations to speed up mathematical function execution.
  - Exploration of specialized hardware targeting deep learning

- Aims to accelerate deep learning on heterogeneous computing architectures by means of compiling techniques
  - Explores multi-level optimizations from compute graph to hardware
  - Exploits machine-learning based fast hyper-parameter search approaches to yield better math function kernels.
  - Supports diverse hardware including CPUs, GPUs, FPGAs and et al.

Woodpecker-DL is part of Woodpecker, a generic compiler framework for heterogeneous computing developed in Ant Financial.
Deep Learning Compilers

Compute Graph Optimization

Tensor Expressions of Graph

DSL Compiler

Optimization Framework

Auto-Tuners

Software DL (LLVM, CUDA, Metal)

Hardware DL (Verilog, HLS, Spatial)

Expert-Optimized Libraries

- TensorFlow
- Keras
- PyTorch
- Caffe
Woodpecker-DL Architecture

Woodpecker Frontend
- Graph Optimization (in-place, pruning, fusion)
- Shape Inference

Math functions in optimized graph

Woodpecker AutoSearch Optimizer
- Ordinary Functions
- Composite Functions

CUDA assembly codes generated

Woodpecker Runtime Engine
- Proprietary Engine
- TensorRT
- TensorFlow
- PyTorch

Woodpecker Addons
- Model Safeguard
- TensorRT Plugins
- Custom TF Ops
- Custom PyTorch Extensions
Outline

- Woodpecker-DL Overview

- Key Components and Technology
  - **Graph Optimization**
    - Auto Search with GA, RL Algorithm
    - DSL Compiler (Halide)

- Integration with TensorRT and Experiment Figures
Graph Optimization

- Support multiple deep learning frameworks
  - TensorFlow, PyTorch, Caffe, CoreML

- Compute graph optimization
  - Simplification, removal and fusion
  - Horizontal or vertical compositional transformation.

- Shape inference of operators

![Diagram showing graph optimization example]

- Convolution before merging
  \[ W \times X + B \]

- Batch Normalization
  \[
  \mu \leftarrow \frac{1}{m} \sum_{i=1}^{m} x_i, \quad \sigma^2 \leftarrow \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu)^2
  \]
  \[
  \hat{x}_i \leftarrow \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}}, \quad y_i \leftarrow \gamma \hat{x}_i + \beta
  \]

- Merge conv and bn
  \[
  \alpha = \frac{\gamma}{\sqrt{\sigma^2 + \epsilon}}
  \]
  \[
  W_{merged} = W \times \alpha
  \]
  \[
  B_{merged} = B \times \alpha + (\beta - \mu \times a)
  \]
Outline

- Woodpecker-DL Overview
- Key Components and Technology
  - Graph Optimization
  - Auto Search with GA, RL Algorithm
  - DSL Compiler (Halide)
- Integration with TensorRT and Experiment Figures
AutoSearch Optimizer

- A machine learning-based framework for automated mathematical kernel optimizations

Algorithms from various domains:
- Deep learning
- Graph computing
- Math optimizations
- Data analysis

Parameterized program:
- Halide
- GraphIt
- Weld
- Spatial
- CUDA
- ...

Efficient program or hardware

Optimization algorithms:
- Genetic
- RL
- Bayesian
- MCMC
- SA
- ...

Feedback:
- Measurement
  - Profiling
  - Historical data
- Perf model
  - Program
  - Hardware

Hardware:
- CPU
- GPU
- FPGA
- Ali-NPU
- Plasticine
- Mobile/Embed
AutoSearch: Genetic Algorithm

- **Genetic Algorithm**
  - Varies population size as per scales of real search space
  - Joins all hyper-parameters (genes) in order to form a chromosome
  - Uses Roulette wheel selection

- **Roulette wheel selection**
  - Candidate
    - 5 4 1 1 1 4 4 ...
    - Fitness 0.4
  - Candidate
    - 8 1 1 1 1 8 4 ...
    - Fitness 0.2
  - Candidate
    - 5 4 1 1 1 4 4 ...
    - Fitness 0.1

- **Crossover**
  - Child candidate
    - 8 1 1 1 1 4 4 ...
  - We replace the child candidate with the most similar candidate in the valid search space

- **Mutation**
  - Child candidate
    - 8 1 3 1 1 4 4 ...

- **N**
  - N = K random filled

- **Fitness**
  - 0.4
  - 0.2
  - 0.1

- **Roulette wheel selection**
  - Chromosome 1: 11%
  - Chromosome 2: 18%
  - Chromosome 3: 41%
  - Chromosome 4: 28%
  - Chromosome 5: 11%
AutoSearch: Search Space

- Take convolution as an example:
  - Image size (1, 64, 56, 56), filter size (64, 64, 1, 1)
  - 9 optimizing dimensions: data splitting dimension, granularity, processing order, caching or not
  - $56 \times 56 \times 64 \times 8 \times 8 \times 8 \times 6 \times 4 \times 6 = 14$ billion choices
  - Brute force: $14$ billion * 100 ms per iteration $\rightarrow$ 22 years
  - Brute force with pruning: 230 thousands choices $\rightarrow$ 1.35 days
  - Genetic search: $1600$ choices $\rightarrow$ 12 minutes

<table>
<thead>
<tr>
<th>Options</th>
<th>ThreadX</th>
<th>ThreadY</th>
<th>ThreadZ</th>
<th>TileX</th>
<th>TileY</th>
<th>TileZ</th>
<th>TileRZ</th>
<th>Shared Mem</th>
<th>Thread LoopOrder</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range</td>
<td>(1, 56)</td>
<td>(1, 56)</td>
<td>(1, 64)</td>
<td>(1, 8)</td>
<td>(1, 8)</td>
<td>(1, 8)</td>
<td>(1, 6)</td>
<td>(1, 4)</td>
<td>(1, 6)</td>
</tr>
</tbody>
</table>
AutoSearch Performance: Genetic Algorithm

- Converges in 10 minutes with a population size of 64
- 2.8x faster than NVIDIA cuDNN, 1.5x faster than TVM
AutoSearch: Reinforcement Learning

- Reinforcement Learning
  - Customized environment and policy graph
  - Uses RLlib scalable reinforcement learning framework
AutoSearch Performance: Reinforcement Learning

- Operations taken from a convolutional model for Ant Financial Payment business
- RL finds better performance than GA in some cases (within the same time)

RL does not always outperform GA
Outline

- Woodpecker-DL Overview

- Key Components and Technology
  - Graph Optimization
  - Auto Search with GA, RL Algorithm
  - **DSL Compiler (Halide)**

- Integration with TensorRT and Experiment Figures
A Domain-Specific Language (DSL) and compiler for image processing pipelines.

- Separates algorithm from schedule
- Enables more efficient and flexible optimizations
- Open source: [https://github.com/halide/Halide](https://github.com/halide/Halide)

**Algorithm:**

\[ g(x, y) = x + y \]

\[ f(x, y) = \frac{g(x, y - 1) + g(x, y) + g(x, y + 1)}{3} \]

**Schedule:**

\[ f. \text{gpu\_tile}(x, y, x_o, y_o, x_i, y_i, 8, 8) \]
Intermediate Codes Generated by Halide

```c
gpu_block (f.s0.y.yo._block_id_y, 0, t32) {
  gpu_block (f.s0.x.xo._block_id_x, 0, t33) {
    gpu_thread (.__thread_id_y, 0, 8) {
      gpu_thread (.__thread_id_x, 0, 8) {
        let f.s0.y.yi.base.s = min((f.s0.y.yo._block_id_y * 8), t34)
        if ((f.s0.x.xo._block_id_x < t35)) {
          f[(((f.min.1 + f.s0.y.yi.base.s) + .__thread_id_y) * f.stride.1) +
            ((f.s0.x.xo._block_id_x * 8) + t16)) + .__thread_id_x] =
            (g[(((.__thread_id_y + f.s0.y.yi.base.s) + 1) * f.extent.0) +
              ((f.s0.x.xo._block_id_x * 8) - f.extent.0)) + .__thread_id_x] +
            (g[(((f.s0.x.xo._block_id_x * 8) + ((.__thread_id_y +
                f.s0.y.yi.base.s) + 1) * f.extent.0)) + .__thread_id_x]) +
            g[(((.__thread_id_y + f.s0.y.yi.base.s) + 1) * f.extent.0) +
              ((f.s0.x.xo._block_id_x * 8) + f.extent.0)) + .__thread_id_x))) / 3)
        } else{
          f[(((f.min.1 + f.s0.y.yi.base.s) + .__thread_id_y) * f.stride.1) +
            .__thread_id_x)] =
            (g[(((.__thread_id_y + f.s0.y.yi.base.s) + 1) * f.extent.0) +
              .__thread_id_x] -8)] + (g[(((.__thread_id_y +
                f.s0.y.yi.base.s) + 2) * f.extent.0) + .__thread_id_x] -8)] +
            g[(((.__thread_id_y + f.s0.y.yi.base.s) + 3) * f.extent.0) +
              .__thread_id_x] -8)]) / 3)
      }
    }
  }
}
```
Halide Schedules

- **Drawbacks**
  - Still needs domain-specific knowledge and skills to get good performance.
  - Given a specific architecture, there are considerable number of schedules to explore.
  - Some schedules are architecture-aware, and thus different hardware needs to exploit different schedules.

- **Example schedules**
  - Loop
    - split, reorder, unroll, tile, storage layout and et al.
  - Stage granularity
    - Coarse-grained: insufficient shared memory, limiting other schedules (storage granularity)
    - Fine-grained: insufficient data reuse and inefficient load/store

- **Schedules are crucial for gaining high performance given a math function**
  - Thus motivated the development of automated search approaches for optimal schedules.
An Example Schedule without Layout Optimization

Storage transform (put $C_o$ inner-most)
- $N$: batch size
- $C_o$: output channels
- $H$: output height
- $W$: output width

Performance

<table>
<thead>
<tr>
<th></th>
<th>layout optimize</th>
<th>w/o layout optimize</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>1.625</td>
<td>1</td>
</tr>
</tbody>
</table>

Profiling

<table>
<thead>
<tr>
<th></th>
<th>layout optimize</th>
<th>w/o layout optimize</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global Load Efficiency</td>
<td>96.30%</td>
<td>14.10%</td>
</tr>
<tr>
<td>Global Store Efficiency</td>
<td>100%</td>
<td>50.00%</td>
</tr>
<tr>
<td>Efficiency</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Occupancy</td>
<td>4.50%</td>
<td>8.90%</td>
</tr>
</tbody>
</table>

(N, $C_o$, $H$, $W$) → (N, $H$, $W$, $C_o$)
Outline

- Woodpecker-DL Overview

- Key Components and Technology
  - Graph Optimization
  - Auto Search with GA, RL Algorithm
  - DSL Compiler (Halide)

- Integration with TensorRT and Experiment Figures
Supports multiple inference engines
  - Proprietary engine, external serving (TensorFlow, PyTorch, and TensorRT)

Diagram showing Woodpecker-DL Integration with TensorRT
Performance: ResNet-18 (Breakdown)

- For separate convolution operations

<table>
<thead>
<tr>
<th>c1</th>
<th>c2</th>
<th>c3</th>
<th>c4</th>
<th>c5</th>
<th>c6</th>
<th>c7</th>
<th>c8</th>
<th>c9</th>
<th>c10</th>
<th>c11</th>
<th>c12</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.79</td>
<td>0.53</td>
<td>1.81</td>
<td>2.17</td>
<td>5.40</td>
<td>0.76</td>
<td>2.66</td>
<td>3.60</td>
<td>0.90</td>
<td>3.18</td>
<td>3.89</td>
<td>1.80</td>
</tr>
<tr>
<td>0.86</td>
<td>0.61</td>
<td>2.56</td>
<td>2.56</td>
<td>3.46</td>
<td>0.84</td>
<td>3.67</td>
<td>4.15</td>
<td>2.36</td>
<td>3.41</td>
<td>1.69</td>
<td></td>
</tr>
</tbody>
</table>

Resnet-18 (Higher is better)

- cuDNN
- TVM
- Woodpecker

Relative Speedup
Performance: ResNet-18 (Summation)

Sum up the runtimes of all convolution operations
Performance: Ant Financial Payment Model

Ant Financial Payment Business (Higher is better)

<table>
<thead>
<tr>
<th>Batch Size</th>
<th>Relative Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.12</td>
</tr>
<tr>
<td>2</td>
<td>2.00</td>
</tr>
<tr>
<td>3</td>
<td>1.77</td>
</tr>
<tr>
<td>4</td>
<td>1.52</td>
</tr>
<tr>
<td>5</td>
<td>1.48</td>
</tr>
<tr>
<td>6</td>
<td>1.43</td>
</tr>
<tr>
<td>7</td>
<td>1.40</td>
</tr>
<tr>
<td>8</td>
<td>1.50</td>
</tr>
<tr>
<td>9</td>
<td>1.31</td>
</tr>
<tr>
<td>10</td>
<td>1.26</td>
</tr>
<tr>
<td>11</td>
<td>1.34</td>
</tr>
<tr>
<td>12</td>
<td>1.23</td>
</tr>
<tr>
<td>13</td>
<td>1.20</td>
</tr>
<tr>
<td>14</td>
<td>1.23</td>
</tr>
<tr>
<td>15</td>
<td>1.33</td>
</tr>
<tr>
<td>16</td>
<td>1.24</td>
</tr>
</tbody>
</table>

- TensorRT
- Woodpecker

Dynamic batching enabled
References

Team Members

Liu, Yongchao
Jin, Yue
Chen, Yong
Ou, Hang
Zhao, Rui
Teng, Teng
Zhang, Yao

Thank You!