Deep Learning as a Polyhedral Compiler's Killer App

From Research to Industry Transfer and Back and Again
Albert Cohen - Inria and Facebook
(work in progress)
Nicolas Vasilache, Oleksandr Zinenko,
Theodoros Theodoridis, Priya Goyal, Zach DeVito,
William S. Moses, Sven Verdoolaege,
Andrew Adams, Albert Cohen
Deep learning has become massively popular over the last 10 years. Machine Learning (ML) frameworks are all over the place. Is this good enough?
Programming Language & Systems Research... 
... for Deep Learning?

A tale of many layers

Input

\( W_1 \) \rightarrow Conv

\( B_1 \) \rightarrow Add

\rightarrow ReLU

... 

\text{Caffe2}

\text{caffe2.python.brew.conv()}

\text{...}

\text{PyTorch}

torch.nn.conv2d()

\text{...}

\text{TensorFlow*}

tf.contrib.layers.conv2d()

\text{...}

\text{NVIDIA cuDNN}

cudnnConvolutionForward()

\text{...}

\text{Intel MKL}

dnnConvolutionCreateForward_F32()

\text{...}

* TF also can compile via XLA, discussed later
Writing “Good” Neural Network Layers

Tedious experience out of reach from Machine Learning (ML) users and researchers

- **Existing layers**: already in vendor libraries from Intel, Nvidia, etc.
  - Can reach really great performance, 95%+ efficiency on a few, almost ideal kernels
  - But the practice is often far from machine peak
- **New layer or neural net architecture** → performance bottleneck
  - Heavy engineering burden to port across ML frameworks
  - HPC library coders are scarce, not all genius, don’t scale, tune by hand

→ Goal: capture the ML expert’s idea, concisely, automate the rest
Our Approach

Don’t write programs, synthesize them

Derive ML-specific techniques from software synthesis and compilers

- A mini-language, close to the mathematics and easy to manage by automatic tools
- A compiler for algebraic/algorithmic optimization and automatic differentiation
- A compiler for “polyhedral” scheduling and mapping
- Generates efficient kernel implementations, e.g., for GPU acceleration
- Just-in-time specialization of the model (hyper)parameters, automatic tuning
- Works transparently: “A New Op” for machine learning and applications
- Integrated with industry-standard frameworks (currently Caffe2, PyTorch)
“Abstraction without regret”

- To make development efficient, we need abstractions that provide productivity without sacrificing performance.
- Given the enormous number of potential kernels, suggests a dynamic-code-generation approach.
Prior work

- “Direct generation” such as active library [2] or built-to-order (BTO) [3] provide usability, but miss optimization.
- DSLs such as Halide [4] provide usability, and permit scheduling transformations, though manually specify.
- Compilers like XLA [5] or Latte [6] optimize and fuse operators, though performance lacking as the language can’t represent complex schedules crucial to GPU/others.

Our Approach

Tensor Comprehensions
Our Approach

TC language

Concise, emits 1000’s of optimized LOC

```python
def mv(float(M,K) A, float(K) x) -> (C)
{
    C(i) +=! A(i,k) * x(k)
}

    -> (01, 02, 03) {
    01(n, o, h, w) +=! I(n, c, h + kh, w + kw) * W1(o, c, kh, kw)
    01(n, o, h, w) = fmax(01(n, o, h, w), 0) // relu
    02(n, d, h, w) +=! 01(n, d, h + kh, w + kw) * W2(d, o, kh, kw)
    02(n, d, h, w) = fmax(02(n, d, h, w), 0)
    03(n, e, h, w) +=! 02(n, c, h + kh, w + kw) * W3(e, d, kh, kw)
    03(n, e, h, w) = fmax(03(n, e, h, w), 0)
}
```

Iteration bounds inferred

Variables only on one side are reduced
Synthesize From Mathematical Model...

Tight mathematical model, emits 1000s optimized lines of code

- **Goup Convolution**

```python
def g_conv_hwcgn(float I(H, W, C, G, N), float W(0, G, C, KH, KW)) -> (0) {
    0(h, w, c, g, n) += I(h + kh, w + kw, c, g, n) * W(o, g, c, kh, kw)
}
```

- **Kronecker Recurrent Units (KRU)**

  → algorithmic exploration of storage/recompute tradeoffs

```python
def 3KRU_v0(float(D0,N0) W0, float(D1,N1) W1, float(D2,N2) W2, 
    float(M,N0,N1,N2) X) -> (Y) {
    Y(m,d0,d1,d2) += X(m,n0_r,n1_r,n2_r) * W2(d2,n2_r) 
    * (W1(d1,n1_r) * W0(d0,n0_r))
}

def 3KRU_v1(float(D0,N0) W0, float(D1,N1) W1, float(D2,N2) W2, 
    float(M,N0,N1,N2) X) -> (Y,XW2) {
    Y(m,d0,d1,d2) += X(m,n0,n1,r2) * W2(d2,r2) 
    * W1(d1,n1_r) * W0(d0,n0_r)
    XW2(m,n0,n1,d2) += X(m,n0,n1,r2) * W2(d2,r2) 
    * W1(d1,n1_r) * W0(d0,r0)
}

def 3KRU(float(D0,N0) W0, float(D1,N1) W1, float(D2,N2) W2, 
    float(M,N0,N1,N2) X) -> (Y,XW1,XW2) {
    Y(m,d0,d1,d2) += X(m,n0,n1,r2) * W2(d2,r2) 
    * W1(d1,n1_r) * W0(d0,r0)
    XW1(m,n0,d1,d2) += XW2(m,n0,r1,d2) * W1(d1,r1) 
    * W0(d0,r0)
    XW2(m,n0,d1,d2) += XW1(m,r0,d1,d2) * W0(d0,r0)
}
```
Synthesize From Mathematical Model...

Tight mathematical model, emits 1000s optimized lines of code

```python
def 2LUT(float(E1,D) LUT1, int(B,L1) I1,
            float(E2,D) LUT2, int(B,L2) I2) -> (01,02) {
    01(i,j) += ! LUT1(I1(i,k),j)
    02(i,j) += ! LUT2(I2(i,k),j)
}

def MLP1(float(B,M) I, float(0,N) W1, float(0) B1) -> (01) {
    01(b,n) = B1(n)
    01(b,n) += I(b,m) * W1(n,m)
    01(b,n) = fmaxf(01(b,n), 0)
}

def MLP3(float(B,M) I, float(0,N) W2, float(0) B2,
            float(P,O) W3, float(P) B3,
            float(Q,P) W4, float(Q) B4) -> (01,02,03,04) {
    02(b,o) = B2(o)
    02(b,o) += 01(b,n) * W2(o,n)
    02(b,o) = fmaxf(02(b,o), 0)
    03(b,p) = B3(p)
    03(b,p) += 02(b,o) * W3(p,o)
    03(b,p) = fmaxf(03(b,p), 0)
    04(b,q) = B4(q)
    04(b,q) += 03(b,p) * W4(q,p)
    04(b,q) = fmaxf(04(b,q), 0)
}

def prodModel(float(E1,D) LUT1, int(B,L1) I1,
               float(E2,D) LUT2, int(B,L2) I2,
               float(B,WX) I3, float(WY,WX) W,
               float(N,M) W1, float(N) B1,
               float(O,N) W2, float(Q) B2,
               float(P,O) W3, float(P) B3,
               float(Q,P) W4, float(Q) B4) -> (C1,C2,C3,I,01,02,03,04) {
    (C1,C2) = 2LUT(LUT1,I1,LUT2,I2)
    C3(b,wy) += I3(b,wx) * W(wy,wx)
    I(b,m) = concat(C1, C2, C3) # not implemented yet
    01 = MLP1(I, W1, B1)
    (02,03,04) = MLP3(01, W2, B2, W3, B3, W4, B4)
    # 04 goes out to binary classifier, omitted here
```
... Rather Than Write Accelerator Code
Our Approach

Polyhedral + TC

- High Level Polyhedral IR (ISL) => Easy Transformations
- Schedule heuristic folds into a single kernel
- Schedule tiled to facilitate the mapping and reuse of memory hierarchy of GPU/CPU
- GPU mapping borrows from PPCG, with extensions for more complex/imperfectly nested structures
- Memory promotion into shared cache
def sgemm(float a, float b float(N,M) A, float(M,K) B) -> (C) {
    C(i,j) = b          // S(i,j)
    C(i,j) += a * A(i,k) * B(k,j)  // T(i,j,k)
}

Sequence node: order-dependent collection of nodes
Band node: (partial) execution
Filter node: partition iteration space
Heard That Before?

- 30 years of parallelizing and optimizing compiler research
- ... wrapped into a robust, automated tool, w/ much ML specialization
- ... with modern C++ interface and tight ML framework integration
- Embed the most complex compositions of loop nest and tensor optimizations
Heard That Before?

- 30 years of parallelizing and optimizing compiler research
- ... wrapped into a robust, automated tool, w/ much ML specialization
- ... with modern C++ interface and tight ML framework integration

![Diagram of optimization steps for sgemm](Figure 3: Optimization steps for sgemm from Figure 1)
Algorithmic Contributions

Extending ISL scheduling

- Extended ISL’s scheduler to allow additional constraints
  - Affine constraint added to the LP
  - Supply clustering decision for graph component combining
- Clustering allows for conventional minimum and maximum fusion targets AND maximum fusion that preserves at least three nested parallel loops (i.e. for mapping to CUDA blocks / threads)
Memory Promotion

$$0[l + \text{Idx}[i][j]][k] \Rightarrow \text{shared}_0[l][i][j][k]$$

- Cache indirectly accessed arrays
- Only done when 0 and \text{Idx} are only read (not written)
- Promote directly accesses if tile of fixed size, elements reused, and \( \geq 1 \) access without memory coalescing
- Promote indirectly accessed arrays in same way (ignore coalescing)
Performance?

Autotuning

- Even with heuristics, there’s a large space of options
- Derive schedule (and other parameters) by searching via genetic algorithm with fixed search-time.
Performance?

End-to-end benchmarks

Baseline CUDA 8.0, CUBLAS 8.0, CUDNN 6.0, CUB recent

Pascal GPU Benchmarks

8 Pascal nodes with 2 socket, 14 core Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz, with 8 Tesla P100-SXM2 GPUs and 16GB of memory each.
Median runtime out of a batch of 1000
Algorithmic Exploration

Autotuning benchmarks

<table>
<thead>
<tr>
<th>KRU Research Layer, 2 orders mag. faster than GEMM</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MD0D1D2N0N1N2</strong></td>
</tr>
<tr>
<td>-------------------</td>
</tr>
<tr>
<td><strong>KRU3_3</strong></td>
</tr>
<tr>
<td>Caffe2</td>
</tr>
<tr>
<td>ATen</td>
</tr>
<tr>
<td>TC (manual)</td>
</tr>
<tr>
<td>TC (autotuned)</td>
</tr>
<tr>
<td><strong>KRU3_3</strong></td>
</tr>
<tr>
<td>Caffe2</td>
</tr>
<tr>
<td>ATen</td>
</tr>
<tr>
<td>TC (manual)</td>
</tr>
<tr>
<td>TC (autotuned)</td>
</tr>
</tbody>
</table>

Figure 5: Wall-clock execution of kernels (in μs). Each kernel ran 1000 times. The top half of each table is Tesla M40 (Maxwell) and the bottom half is Tesla P100 (Pascal). N/A denotes the framework lacked an implementation.
Recurrent Networks?

Courtesy Chris Olah, Google Brain
Simple RNN
Long Short Term Memory (LSTM)
Recurrent Networks?

- Work in progress

- Inspiration from functional languages with combinators, reactive and dataflow synchronous languages

  Related to the modeling of finite state machines on infinite streams with Kahn semantics in Lucid Synchrone / Esterel Technologies Scade 6

  Challenge: combining data flow on init/last value with storage of tensor slices into higher dimensional tensors

- Good news: natural modeling of nested loops, space and time, in a polyhedral framework
TC overview

“Natural ML math running faster than libraries”

- Available stand-alone and in Caffe2/PyTorch bindings [public in a few days]
- Open source: https://github.com/facebookresearch/tensorcomprehensions
Where From? Polly Labs Initiative

Mathematical core “isl”: parametric linear optimization, Presburger arithmetic

Industry transfer lab founded in February 2015

Building on 12 years of collaboration with AMD (Austin, Bangalore) and IISc (CSA, Bangalore)

INRIA-CEFIPRA Associate Team with IISc

Google Summer of Code(s) with IITH

IIT Hyderabad

IISc

Qualcomm

Xilinx

Facebook

Partners

ETH Zürich

Mission statement
Promote polyhedral compilation techniques in production, open source platforms, with an emphasis on LLVM/Polly

Founding partners

Polly Labs founded!
http://pollylabs.org
Research and Industry Transfer - Virtual Lab
https://www.pollylabs.org

• Bilateral contract between INRIA and ARM, in cooperation with Qualcomm (QuIC), Xilinx, Facebook (FAIR)

• Focus on LLVM ecosystem: http://llvm.org → exploitation & developer community

• Mutualization of core polyhedral compilation infrastructure
• Contributing to domain-specific – deep learning, image processing, solvers, linear algebra – and research compilers
• Training and tutorials
Timeline

• **isl** started in 2008, licensed under **LGPLv2.1**
  Used by GCC as its polyhedral library since 5.0
  … virtually every smartphone runs code generated using our algorithms
  [http://repo.or.cz/w/isl.git](http://repo.or.cz/w/isl.git)

• 2013: Relicensed under **MIT**, through **CARP EU project**
  Used by LLVM through the Polly project

• 2014: Triggered **ARM** to release tools to generate linear algebra kernels
• 2014: **Qualcomm**, then **ARM**, push for **Polly Labs**
• 2015: Qualcomm Snapdragon LLVM uses Polly, **compiles Android OSP**
• 2016: **Xilinx** starts an isl-based project within **Vivado HLS**
• 2017: **Facebook** works on a **deep learning compiler** using isl
• 2018: **Safran, Morphing Machines** (IISc spin-off), **CEFIPRA** project to work on correct-by-construction signal processing and data analytics
Context: economic challenge

Funding a sustainable software tool development

→ Mutualize investments by large and not-so-large users
→ Leverage many different research funding instruments
→ Build a sustainable ecosystem of tool vendors
Polly Labs experience so far: importance of both upstream and application-driven, cross-domain research, open source, distributed teams and world-class recruitment, advanced prototyping of software artifacts for successful industry transfer.

but very hard to work out labor law and inflexible rules of academic institutions...

We are OPEN, HIRING and Looking for COLLABORATIONS

Albert.Cohen@inria.fr