Modular Convolution Considered Beneficial
Kernel Composition for Neural Network

Chao Liu, Jing Zhang, Jack Chung, Daniel Lowell,
Tejash Shah, Jehandad Khan
Radeon Technologies Group (RTG), AMD
INTRODUCTION

MIOpen (https://github.com/ROCmSoftwarePlatform/MIOpen)

- AMD's open source machine learning GPU kernels library
- Composable kernel

Tensorflow support
MOTIVATION

- Monolithic kernel is complicated
- Kernel fusion is hard
- Various tensor layouts
- Future hardware

Machine learning systems are stuck in a rut, Barham & Isard, HotOS’19
SOLUTION: MODULE & COMPOSITION

- Generalize modules that are:
  - Reusable
  - Composable
  - Hardware abstraction
  - Compiler friendly
  - Tunable

- Use modules to compose:
  - Convolution
  - Batch norm
  - Operator fusion and more
CONVOLUTION NEURAL NETWORK
A 1D CONVOLUTION EXAMPLE

1d Convolution $[K, C, X] \otimes [C, W] = [K, \hat{W}]$, batch size 1, stride 1, dilation 1, pad 0

Weight$[K, C, X]$

Input$[C, W]$

Output$[K, \hat{W}]$

Algo:
- Direct
- GEMM based
- FFT, Winograd
CONVOLUTION ALGORITHM: IM2COL + GEMM

⚠️ Method:

- Im2Col: convert N-D tensor to matrix
- GEMM

⚠️ Advantage:

- Can use existing GEMM library

⚠️ Disadvantages:

- Extra data movement
- Large extra memory footprint

Why GEMM is at the heart of deep learning.
https://petewarden.com/2015/04/20/why-gemm-is-at-the-heart-of-deep-learning/
CONVOLUTION ALGORITHM: IMPLICIT GEMM + PRECOMPUTED INDICES

- **Method:**
  - Indirect buffer for precomputed indices
  - GEMM-like compute kernel

- **Advantages:**
  - No extra data movement due to Im2Col
  - Small additional memory overhead
  - Simple compute kernel

- **Disadvantages:**
  - Reading data is 2-hop memory access
  - No info on original tensor indices

The Indirect Convolution Algorithm. 2019, Marat Dukhan. arxiv.org/abs/1907.02129
Design targets:

- No additional data movement
- No additional memory allocation
- Direct memory access
- Full info on tensor indices
A “generic” tensor contains:

- Raw data in memory
- A mapping function $g: \hat{x} \rightarrow y$
- Coordinate $\hat{x} \rightarrow$ memory address $y$

“Generic” mapping
“Naïve” tensor has:

- “Naïve” mapping function

\[ y = \tilde{s} \cdot \tilde{x} + \alpha \]

All frameworks today operate on “naïve” tensors.
“Transformed” tensor

- Original mapping \( g: \hat{x} \rightarrow y \)
- Coordinate transformation \( f: \overrightarrow{w} \rightarrow \hat{x} \)
- Transformed mapping \( g \circ f: \overrightarrow{w} \rightarrow y \)

Remapping only
No data movement
Permute several dimensions

\[ f: \vec{w} \rightarrow \vec{x} = \mathbf{P} \cdot \vec{w} \]

- \( \mathbf{P} \) is permutation matrix
Pad several dimensions

\[ f: \vec{w} \rightarrow \vec{x} = \vec{w} - \vec{p} \]

\( \vec{p} \) is padding sizes
Merge several dimensions into one:

\[-x_i = \text{floor}(\text{modulo}(w_0, \mu_{i-1})/\mu_i)\]

\[-\mu_i = \prod_{j=i}^{n-1} l_j\]
COORDINATE TRANSFORMATION: FOLD

Fold one dimension into several

\[ f: \mathbf{w} \rightarrow \mathbf{x} = \mathbf{\mu} \cdot \mathbf{w} \]

\[ \mu_i = \prod_{j=i}^{n-1} l_j \]
Embed

\[ f: \vec{w} \rightarrow \hat{\vec{x}} = \hat{\vec{e}} \cdot \vec{w} + \varepsilon \]
Transformations can be chained

- Mapping function
  \[ g \circ f_1 \circ \cdots \circ f_k : \mathbf{x}^k \rightarrow y \]

Our algorithms operate on the top most "generic" tensor

<table>
<thead>
<tr>
<th>Coordinate space #k</th>
<th>( \mathbf{x}^k )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \mathbf{x}^{k-1} = f_k (\mathbf{x}^k) )</td>
<td></td>
</tr>
<tr>
<td>Coordinate space #k-1</td>
<td>( \mathbf{x}^{k-1} )</td>
</tr>
<tr>
<td>Coordinate space #1</td>
<td>( \mathbf{x}^1 )</td>
</tr>
<tr>
<td>( \mathbf{x}^0 = f_1 (\mathbf{x}^1) )</td>
<td></td>
</tr>
<tr>
<td>Coordinate space #0</td>
<td>( \mathbf{x}^0 )</td>
</tr>
<tr>
<td>( y = g (\mathbf{x}^0) )</td>
<td></td>
</tr>
<tr>
<td>Address space</td>
<td>( y )</td>
</tr>
</tbody>
</table>
Reducing index calculation

- Lower delta of coordinate to delta of address
  - $\Delta x^k \rightarrow \Delta y$

By applying “chain rule” from calculus
OPTIMIZING INDEX CALCULATION

Coordinate space #k
\[ \Delta x^k \]
\[ \Delta x^{k-1} = f_k \Delta (\Delta x^k, \Delta x^k) \]

Coordinate space #k-1
\[ \Delta x^{k-1} \]

Coordinate space #1
\[ \Delta x^1 \]
\[ \Delta x^0 = f_1 \Delta (\Delta x^1, \Delta x^1) \]

Coordinate space #0
\[ \Delta x^0 \]
\[ \Delta y = g \Delta (\Delta x^0, \Delta x^0) \]

Address space
\[ \Delta y \]

If linear \( f_k \)

If linear \( f_1 \)

If linear \( g \)

Bad

Good

Polyhedral optimization.

```
for (i = 0; i < I; ++i)
    for (j = 0; j < J; ++j)
        for (k = 0; k < K; ++k)
            addr = get_address(i0 + i, j0 + j, k0 + k);
```

```
addr0 = calculate_address(i0, j0, k0);
for (i = 0; i < I; ++i)
    for (j = 0; j < J; ++j)
        for (k = 0; k < K; ++k)
            addr = addr0 + get_address_delta(i, j, k);
```
Neural network composition = constructing “generic” tensors + operations on “generic” tensors
A 1D CONVOLUTION EXAMPLE

1d Convolution $[K, C, X] \otimes [C, W] = [K, \hat{W}]$, batch size 1, stride 1, dilation 1, pad 0
A 1D CONVOLUTION EXAMPLE

MAP CONVOLUTION INTO GEMM

$\text{Input}[C, W]$  
$\text{Input.embed()}[C, X, \hat{W}]$  

$\text{Embed } W \text{ into } (C, \hat{W})$
MAP CONVOLUTION INTO GEMM
A 1D CONVOLUTION EXAMPLE

\[ GEMM: [K, E] \cdot [E, \hat{W}] = [K, \hat{W}] \]

\[ Weight[K, C, X] \]

\[ Input.embed(C, X, \hat{W}) \]

\[ Merge (C, X) \text{ into } E \]

\[ Output[K, \hat{W}] \]

\[ Weight.merge(K, E) \]

\[ Input.embed().merge(E, \hat{W}) \]
MAP CONVOLUTION INTO GEMM

- Any tensor layout
- Any # of dimensions
- Any image size, filter size, pad, stride, dilation
GCN GPU ARCHITECTURE
Composable coordinate space transformations
- Permute
- Pad
- Merge
- Fold
- Embed

Composable generic tensor data operations
- Sliced copy
- GEMM
- Reduction
- Winograd & FFT tensor data transformation

Due to memory hierarchy of GPU, each generic tensor data operation will be implemented on several cooperative levels
Normalized Performance (ResNet50 & Inception3, Batch Size 256)

Composable implicit GEMM vs. monolithic kernels in MIOpen
A solution of kernel composition for neural network

- Constructing “generic” tensors
  - Through composable coordinate transformations
  - Able to compose complex yet still structured tensor-like data structure
  - Compiler-friendly address calculation
- Operations on generic tensor data

Example: Composing convolution neural network (GEMM based)

- No extra data movement or memory allocation
- No need for precomputed indices
- Direct memory access
ALREADY IN YOUR ROCM INSTALLATIONS

- Already shipping since ROCm 2.7.
- Supports both graph execution path and XLA.
- Applicable on >50% of configurations used in TensorFlow CNN benchmarks and keep improving.
ONWARD TO MLIR

- Expose kernel composition process via new MIOpen dialect in MLIR
  - Similar high-level syntax with Linalg dialect for easy integration
  - AMD-specific optimizations to derive high-performance kernels
- Upstreaming building blocks to enable ROCm and MIOpen in MLIR
Example: Drive convolution kernel as a single device function in MLIR

MIOpen high-level operators have a similar syntax with Linalg dialect for easy integration

```mlir
module attributes { gpu.kernel_module } {
  func @miopen_conv2dex_f32(%arg0: memref<128x128x17x17xf32>,
             %arg1: memref<128x128x3x3xf32>,
             %arg2: memref<128x128x?x?xf32>) attributes { gpu.kernel } {
    miopen.conv2dex.f32(%arg0, %arg1, %arg2) {dilations=[1, 1], paddings=[0, 0], strides=[3, 3]}:
    memref<128x128x17x17xf32>, memref<128x128x3x3xf32>, memref<128x128x?x?xf32>
      return
  }
}
```
CONVERT TO LOW-LEVEL OPERATOR WHICH DEALS WITH LLVM TYPES

GENERATE COMMAND FOR MIOPEN KERNEL COMPOSITION PROCESS AND SAVE IN THE ATTRIBUTE

```mlir
module attributes {gpu.kernel_module} {
  func @miopen_conv2dex_f32(%arg0: !llvm<{ float*, i64, [4 x i64], [4 x i64] }*>),
    %arg1: !llvm<{ float*, i64, [4 x i64], [4 x i64] }*>),
    %arg2: !llvm<{ float*, i64, [4 x i64], [4 x i64] }*>) attributes {gpu.kernel} {

    %0 = llvm.load %arg0 : !llvm<{ float*, i64, [4 x i64], [4 x i64] }*>"
    %1 = llvm.load %arg1 : !llvm<{ float*, i64, [4 x i64], [4 x i64] }*>"
    %2 = llvm.load %arg2 : !llvm<{ float*, i64, [4 x i64], [4 x i64] }*>"
    %3 = llvm.extractvalue %0[0 : index] : !llvm<{ float*, i64, [4 x i64], [4 x i64] }*>"
    %4 = llvm.extractvalue %1[0 : index] : !llvm<{ float*, i64, [4 x i64], [4 x i64] }*>"
    %5 = llvm.extractvalue %2[0 : index] : !llvm<{ float*, i64, [4 x i64], [4 x i64] }*>"

    miopen.conv2d.kernelex.f32 %3, %4, %5 {kernel_name = "some_name", kernel_path = "some_where",
    miopen_driver_command = "conv -n 128 -c 128 -H 17 -W 17 -k 128 -y 3 -x 3 -u 3 -v 3 -p 0 -q 0 -l 1 -j 1 -F 1 -V 0 -O 1 "}:
    !llvm<float***>, !llvm<float***>, !llvm<float***>
}
}
```
MLIR MIOPEN DIALECT: RETRIEVE COMPOSED KERNELS

MLIR opt pass to invoke MIOpen and retrieve composed kernels in attributes

```
module attributes {gpu.kernel_module} {
    func @miopen_conv2dex_f32(%arg0: !llvm<{ float*, i64, [4 x i64], [4 x i64] }*},
        %arg1: !llvm<{ float*, i64, [4 x i64], [4 x i64] }*},
        %arg2: !llvm<{ float*, i64, [4 x i64], [4 x i64] }*}) attributes {gpu.kernel} {

        %0 = llvm.load %arg0 : !llvm<{ float*, i64, [4 x i64], [4 x i64] }*> %1 = llvm.load %arg1 : !llvm<{ float*, i64, [4 x i64], [4 x i64] }*> %2 = llvm.load %arg2 : !llvm<{ float*, i64, [4 x i64], [4 x i64] }*> %3 = llvm.extractvalue %0[0 : index] : !llvm<{ float*, i64, [4 x i64], [4 x i64] }*> %4 = llvm.extractvalue %1[0 : index] : !llvm<{ float*, i64, [4 x i64], [4 x i64] }*> %5 = llvm.extractvalue %2[0 : index] : !llvm<{ float*, i64, [4 x i64], [4 x i64] }*>

        miopen.conv2d.kernelex.f32 %3, %4, %5 {kernel_name = "gridwise_convolution_implicit_gemm_v4_nchw_kcyx_chnkhw_lds_double_buffer_n_128_c_128_H_17_W_17_k_128_y_3_x_3_u_3_v_3_p_0_q_0_1_1_l_1_j_1", kernel_path = "/root/.cache/miopen/2.1.0/c024ea5bfa85d4aaf3c526bc183708c2/gridwise_convolution_implicit_gemm_v4_nchw_kcyx_chnkhw_lds_double_buffer.cpp.bc", miopen_driver_command = "conv -n 128 -c 128 -H 17 -W 17 -k 128 -y 3 -x 3 -u 3 -v 3 -p 0 -q 0 -l 1 -j 1 -F 1 -V 0 -O 1 "} : !llvm<float*>, !llvm<float*>, !llvm<float*>}

}
Additional MLIR translation passes will initialize LLVM backend and link with bitcodes comprising convolution kernel

```mlir
define void @miopen_conv2dex_f32({ float*, i64, [4 x i64], [4 x i64] }* %0,
{ float*, i64, [4 x i64], [4 x i64] }* %1,
{ float*, i64, [4 x i64], [4 x i64] }* %2)
{
  %4 = load { float*, i64, [4 x i64], [4 x i64] }, { float*, i64, [4 x i64], [4 x i64] }* %0
  %5 = load { float*, i64, [4 x i64], [4 x i64] }, { float*, i64, [4 x i64], [4 x i64] }* %1
  %6 = load { float*, i64, [4 x i64], [4 x i64] }, { float*, i64, [4 x i64], [4 x i64] }* %2
  %7 = extractvalue { float*, i64, [4 x i64], [4 x i64] } %4, 0
  %8 = extractvalue { float*, i64, [4 x i64], [4 x i64] } %5, 0
  %9 = extractvalue { float*, i64, [4 x i64], [4 x i64] } %6, 0
  call void @gridwise_convolution_implicit_gemm_v4_nchw_kcyx_nkhw_lds_double_buffer_n_128_c_128_H_17_W_17_k_128_y_3_x_3_u_3_v_3_p_0_q_0_l_1_j_1(float* %7, float* %8, float* %9), !kernel_path !0
  ret void
}
```

```mlir
declare void @gridwise_convolution_implicit_gemm_v4_nchw_kcyx_nkhw_lds_double_buffer_n_128_c_128_H_17_W_17_k_128_y_3_x_3_u_3_v_3_p_0_q_0_l_1_j_1(float*, float*, float*)
```
ONWARD TO MLIR

- MIOpen low-level dialect will be extended to support all primitive operators aforementioned
  - Leverage transformations available in MLIR Affine and Loops dialect
- Make it one of the tensor languages for high performance kernel generation
- Any MLIR-based applications can take advantage of MIOpen composable kernels

MIOpen + Affine + Loops dialect
Convolution composed by MLIR
GPU + LLVM dialect
Disclaimer

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions, and typographical errors. The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. Any computer system has risks of security vulnerabilities that cannot be completely prevented or mitigated. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

THIS INFORMATION IS PROVIDED ‘AS IS.’ AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS, OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION. AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY RELIANCE, DIRECT, INDIRECT, SPECIAL, OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

ATTRIBUTION

© 2019 Advanced Micro Devices, Inc. All rights reserved. AMD, the AMD Arrow logo and combinations thereof are trademarks of Advanced Micro Devices, Inc. in the United States and/or other jurisdictions. Other names are for informational purposes only and may be trademarks of their respective owners.