Coarse-Grained Resource Sharing for Entire Neural Networks

Tzung-Han Juang$^1$ Christof Schlaak$^2$ Christophe Dubach$^1$

$^1$McGill University, Canada
$^2$University of Edinburgh, United Kingdom

LATHC Workshop, Feb. 2023
Deep Neural Networks

- Different kinds of layers
- Different layer sizes

![Diagram of Deep Neural Networks]

- **Input**: 224x224x64
- **Conv1**: 112x112x128
- **Conv2**: 56x56x256
- **Conv3**: 28x28x512
- **Conv4**: 14x14x512
- **Conv5**: 7x7x512
- **Fully-Connected**: 4096
- **Output**: 1000
Deep Neural Networks

- Different kinds of layers
- Different layer sizes
- Mostly Multiplications and Additions (Good for sharing)
Hardware Accelerators

- **CPU**
- **GPU**
- **FPGA**
- **ASIC**

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Programming Difficulty</th>
<th>Performance</th>
<th>Flexibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Easy</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td>GPU</td>
<td>Hard</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>FPGA</td>
<td>Easy</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>ASIC</td>
<td>Hard</td>
<td>High</td>
<td>Low</td>
</tr>
</tbody>
</table>
Hardware Accelerators

<table>
<thead>
<tr>
<th>CPU¹</th>
<th>GPU²</th>
<th>FPGA³</th>
<th>ASIC⁴</th>
</tr>
</thead>
<tbody>
<tr>
<td>Easy to program</td>
<td></td>
<td></td>
<td>Hard to program</td>
</tr>
</tbody>
</table>

¹ Intel CORE i9
² NVIDIA
³ FPGA
⁴ Custom ASIC
Hardware Accelerators

<table>
<thead>
<tr>
<th>Hardware Type</th>
<th>Programming Difficulty</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU(^1)</td>
<td>Easy to program</td>
<td>Low</td>
</tr>
<tr>
<td>GPU(^2)</td>
<td>Easy to program</td>
<td>High</td>
</tr>
<tr>
<td>FPGA(^3)</td>
<td>Hard to program</td>
<td>High</td>
</tr>
<tr>
<td>ASIC(^4)</td>
<td>Hard to program</td>
<td>High</td>
</tr>
</tbody>
</table>

CPU: Intel
GPU: NVIDIA
FPGA: Xilinx
ASIC: EnSilica
Hardware Accelerators

- **CPU**
  - Easy to program
  - Low performance
  - High Flexibility

- **GPU**
  - Hard to program
  - High Performance
  - Low Flexibility

- **FPGA**
  - Easy to program
  - Low performance
  - High Flexibility

- **ASIC**
  - Hard to program
  - High Performance
  - Low Flexibility
Fitting Entire Neural Networks into FPGA

**Target:** Fit layers into fpga but still keep the performance
**Target:** Fit layers into FPGA but still keep the performance
Fitting Entire Neural Networks into FPGA

**Target:** Fit layers into fpga but still keep the performance
Approach from Existing Tools (Fine-Grained)
Approach from Existing Tools (Fine-Grained)
Approach from Existing Tools (Fine-Grained)
**Intel OpenCL FPGA SDK Example**

Matrix multiplications

---

```
1 void mxm(int* A, int*B, int* C, int sz) {
2     for(i=0, j=0; i<sz, j<sz; i++, j++) {
3         #pragma unroll
4         for(int k=0; k<size; k++)
5             C[i][j] = A[i][k] * B[j][k]; }
6 }
7
8 kernel mm(int* A, int*B, int* C, int sz) {
9 }
```

Arria 10 FPGA usage

<table>
<thead>
<tr>
<th>Resources</th>
<th># of calls</th>
<th>Logic(%)</th>
<th>RAM(%)</th>
<th>DSP(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

LATHC 2023 6 / 20
Matrix multiplications

```c
void mxm(int * A, int *B, int * C, int sz) {
    for (i=0, j =0; i<sz , j<sz; i++ , j ++) {
        #pragma unroll
        for (int k =0; k< size ; k ++)
            C[i][j] = A[i][k] * B[j][k]; }
}

kernel mm(int * A, int *B, int * C, int sz) {
    mxm(A, B, C, sz);}
```
Matrix multiplications

void mxm(int* A, int* B, int* C, int sz) {
    for (i=0, j=0; i<sz, j<sz; i++, j++) {
        #pragma unroll
        for (int k=0; k<sz; k++)
            C[i][j] = A[i][k] * B[j][k];
    }
}

kernel mm(int* A, int* B, int* C, int sz) {
    mxm(A, B, C, sz);
    mxm(A, B, C, sz);
}

<table>
<thead>
<tr>
<th># of calls</th>
<th>Logic(%)</th>
<th>RAM(%)</th>
<th>DSP(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>23</td>
<td>19</td>
<td>34</td>
</tr>
<tr>
<td>2</td>
<td>32</td>
<td>36</td>
<td>68</td>
</tr>
</tbody>
</table>

Intel OpenCL FPGA SDK Example

OpenCL Code → HDL Code → FPGA Bit File
Matrix multiplications

`void mxm(int* A, int*B, int* C, int sz) {
    for(i=0, j=0; i<sz, j<sz; i++, j++) {
        #pragma unroll
        for (int k =0; k< size; k++)
            C[i][j] = A[i][k] * B[j][k];
    }
}

kernel mm(int* A, int*B, int* C, int sz) {
    mxm(A, B, C, sz);
    mxm(A, B, C, sz);
    mxm(A, B, C, sz);
}

Arria 10 FPGA usage

<table>
<thead>
<tr>
<th># of calls</th>
<th>Logic(%)</th>
<th>RAM(%)</th>
<th>DSP(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>23</td>
<td>19</td>
<td>34</td>
</tr>
<tr>
<td>2</td>
<td>32</td>
<td>36</td>
<td>68</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>Out of DSPs</td>
</tr>
</tbody>
</table>

McGill University

Coarse-Grained Sharing

LATHC 2023

6 / 20
SHIR Functional Framework

```
1 Reduce (+, 0, input: Array(N)) // Array Sum
```

Shared Function (Coarse-Grained Sharing)

• Let defines a value under a scope.

```
1 Let param = ... in scopeBody
```
Shared Function (Coarse-Grained Sharing)

- **Let** defines a value under a scope.
- **Lambda** defines a anonymous function.

```plaintext
Let matMulFun = \( \lambda \) MatX, MatY ->

\[
\text{Map}(\lambda\ rowX,\ colY ->
\text{Reduce}(+,\ \text{Map}(\lambda\ m =>\ Mul(m),\ Zip(rowX,\ colY))),\ MatX,\ MatY)
\]

in

scopeBody
```
Shared Function (Coarse-Grained Sharing)

- **Let** defines a value under a scope.
- **Lambda** defines a anonymous function.
- **FunCall** calls a lambda function.

```
Let matMulFun = λ MatX, MatY ->
  Map (λ rowX, colY ->
    Reduce (+,
      Map (λ m => Mul(m), Zip(rowX, colY))), MatX, MatY)
in
C = FunCall(matMulFun, A, B)
Out = FunCall(matMulFun, C, D)
```
Let \( \text{matMulFun} = \lambda \text{MatX, MatY} \rightarrow \)

\[
\text{Map}(\lambda \text{rowX, colY} \rightarrow \\
\text{Reduce}(+, \\
\text{Map}(\lambda \text{m} \rightarrow \text{Mul(m)}, \text{Zip(rowX, colY)})), \text{MatX}, \text{MatY})
\]

in

\[
C = \text{FunCall(matMulFun, A, B)} \quad \text{// FunCall1} \\
\text{Out} = \text{FunCall(matMulFun, C, D)} \quad \text{// FunCall2}
\]
Let \( \text{matMulFun} = \lambda \text{MatX}, \text{MatY} \rightarrow \)

\[
\text{Map}(\lambda \text{rowX}, \text{colY} \rightarrow \\
\text{Reduce}(+, \\
\text{Map}(\lambda \text{m} \rightarrow \text{Mul}(\text{m}), \text{Zip}(<\text{rowX}, \text{colY}>))), \text{MatX}, \text{MatY})
\]

in

\( C = \text{FunCall(matMulFun, A, B)} \) // FunCall1

\( \text{Out} = \text{FunCall(matMulFun, C, D)} \) // FunCall2
**Shared Function (Coarse-Grained Sharing)**

Let \( \text{matMulFun} = \lambda \text{MatX}, \text{MatY} \rightarrow \) 
\[
\text{Map}(\lambda \text{rowX}, \text{colY} \rightarrow \\
\text{Reduce}(+, \\
\text{Map}(\lambda \text{m} \rightarrow \text{Mul}(\text{m}), \text{Zip}(\text{rowX}, \text{colY}))), \text{MatX}, \text{MatY})
\]
in 
\[
\text{C} = \text{FunCall}(\text{matMulFun}, \text{A}, \text{B}) \quad // \quad \text{FunCall1} \\
\text{Out} = \text{FunCall}(\text{matMulFun}, \text{C}, \text{D}) \quad // \quad \text{FunCall2}
\]
Let $\text{matMulFun} = \lambda \text{MatX}, \text{MatY} ->$

1. $\text{Map}(\lambda \text{rowX}, \text{colY} ->$
2. $\text{Reduce}(+,$
3. $\text{Map}(\lambda \text{m} => \text{Mul}(\text{m}), \text{Zip}(\text{rowX}, \text{colY}))), \text{MatX}, \text{MatY})$

in

6. $\text{C} = \text{FunCall}(\text{matMulFun}, \text{A}, \text{B})$ // FunCall1
7. $\text{Out} = \text{FunCall}(\text{matMulFun}, \text{C}, \text{D})$ // FunCall2
**Shared Function (Coarse-Grained Sharing)**

The problem is similar to **Register Allocation**.

---

**Interference Graph**

---

```plaintext
Let matMulFun = \( \lambda \text{MatX}, \text{MatY} \rightarrow \)
 
\[
\text{Map}(\lambda \text{rowX}, \text{colY} \rightarrow 
\text{Reduce}(+, 
\text{Map}(\lambda m \rightarrow \text{Mul}(m), \text{Zip}(\text{rowX}, \text{colY}))), \text{MatX}, \text{MatY})
\]

in

\[
C = \text{FunCall}(\text{matMulFun}, A, B) // \text{FunCall1}
\]

Out = \text{FunCall}(\text{matMulFun}, C, D) // \text{FunCall2}
```
Let matMulFun = \( \lambda \) MatX, MatY \(
\begin{array}{l}
\text{Map}(\lambda \text{rowX, colY} \rightarrow \\
\text{Reduce}(+,
\text{Map}(\lambda m \rightarrow \text{Mul}(m), \text{Zip(rowX, colY)))), \text{MatX, MatY})
\end{array}
\)
in
C = FunCall(matMulFun, A, B) \quad // \text{FunCall1}
Out = FunCall(matMulFun, Buffer(C), D) \quad // \text{FunCall2}
**Shared Function (Coarse-Grained Sharing)**

Let \( \text{matMulFun} = \lambda \text{MatX}, \text{MatY} \rightarrow \)

\[
\text{Map}(\lambda \text{rowX}, \text{colY} \rightarrow \\
\text{Reduce}(+, \\
\text{Map}(\lambda \text{m} \rightarrow \text{Mul}(\text{m}), \text{Zip}(\text{rowX}, \text{colY}))), \text{MatX}, \text{MatY})
\]

in

\( \text{C} = \text{FunCall} (\text{matMulFun}, \text{A}, \text{B}) \quad // \text{FunCall1} \)

\( \text{Out} = \text{FunCall} (\text{matMulFun}, \text{Buffer(C)}, \text{D}) \quad // \text{FunCall2} \)

Buffering is similar to Spilling.

Control:

- FunCall1
- FunCall2

Ctrls:

- FunCall1
- FunCall2

Buffer

\( \text{C} \)
Buffering is similar to **Spilling**.

---

### Interference Graph

1. \( \text{Ctrl}_1 \)
2. \( \text{Ctrl}_2 \)

---

Let \( \text{matMulFun} = \lambda \text{MatX, MatY} \rightarrow \)

\[
\text{Map}(\lambda \text{rowX, colY} \rightarrow \\
\text{Reduce}(+, \\
\text{Map}(\lambda \text{m} \rightarrow \text{Mul}(m), \text{Zip}(\text{rowX, colY}))), \text{MatX, MatY})
\]

in

\( \text{C} = \text{FunCall(matMulFun, A, B)} \quad \text{// FunCall1} \)

\( \text{Out} = \text{FunCall(matMulFun, Buffer(C), D)} \quad \text{// FunCall2} \)
Low-Level Synthesizer

Synthesizer does not well support detection of duplicated components.

Sharing multiplication only.

Logic Usage: 33% DSP Usage: 34%

Sharing dot product.
Low-Level Synthesizer

Synthesizer does not well support detection of duplicated components.

Sharing multiplication only.

Logic Usage: 33%  DSP Usage: 34%

Sharing multiplication only.
Mux Propagation

Multiple function calls also lead to redundant components and wiring.

```plaintext
1. Let fun1 = \( \lambda x \rightarrow \text{BlockRAMBuffer}(x) \) in
2. Let fun2 = \( \lambda x \rightarrow \text{Compute}(x) \) in
3. output1 = FunCall(fun2, Transpose( FunCall(fun1, input1) ))
4. output2 = FunCall(fun2, Transpose( FunCall(fun1, input2) ))
```

Before optimizations.

![Diagram showing the process before optimizations.](image-url)
Solution: Propagate Muxes based on rewrite rules.

1. Let \( \text{fun1} = \lambda x \rightarrow \text{Transpose}(\text{BlockRAMBuffer}(x)) \) in
2. Let \( \text{fun2} = \lambda x \rightarrow \text{Compute}(x) \) in
3. \( \text{output1} = \text{FunCall} (\text{fun2}, \text{FunCall}(\text{fun1}, \text{input1})) \)
4. \( \text{output2} = \text{FunCall}(\text{fun2}, \text{FunCall}(\text{fun1}, \text{input2})) \)

After optimizations.
Solution: Merge nearby functions together.

1. Let \( \text{fun3} = \lambda x \rightarrow \text{Compute}(\text{Transpose}(\text{BlockRAMBuffer}(x))) \) in
2. \( \text{output1} = \text{FunCall} (\text{fun3}, \text{input1}) \)
3. \( \text{output2} = \text{FunCall} (\text{fun3}, \text{input2}) \)

After Further merging.
**Experiment Setup**

![Diagram of experiment setup]

1. **Init.**
   - Host RAM
   - Weights
   - Input

2. **Trigger**
   - PCI-E
   - Arria10 FPGA

3. **Transfer**
   - BitFile
VGG16 (Each Layer Independently)
**VGG16 (Entire Network)**

![Diagram of VGG16 network](image)

<table>
<thead>
<tr>
<th>Setup</th>
<th>Perf.</th>
<th>Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DSP eff. (%)</td>
<td>Logic (%)</td>
</tr>
<tr>
<td>One Layer Per FPGA (Max usage)</td>
<td>73</td>
<td>33</td>
</tr>
</tbody>
</table>

- **FPGA**
- **RAM**
- **Conv1**
- **Conv2**
- **Fully Cnct1**
VGG16 (Entire Network)

### Setup

<table>
<thead>
<tr>
<th>Setup</th>
<th>Perf. DSP eff. (%)</th>
<th>Resources Logic (%)</th>
<th>Resources RAM (%)</th>
<th>Resources DSP (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>One Layer Per FPGA (Max usage)</td>
<td>73</td>
<td>33</td>
<td>22</td>
<td>76</td>
</tr>
<tr>
<td>All together. Function Sharing Opt.</td>
<td>73</td>
<td>51</td>
<td>41</td>
<td>76</td>
</tr>
</tbody>
</table>
Future Work

- Evaluation on different neural networks.
- Automatic shared functions detection.
Thank You!

Contact: Tzung-Han Juang (tzung-han.juang@mail.mcgill.ca)
References