Coarse-Grained Resource Sharing for Entire Neural Networks

Tzung-Han Juang[1] Christof Schlaak[2] Christophe Dubach[1]

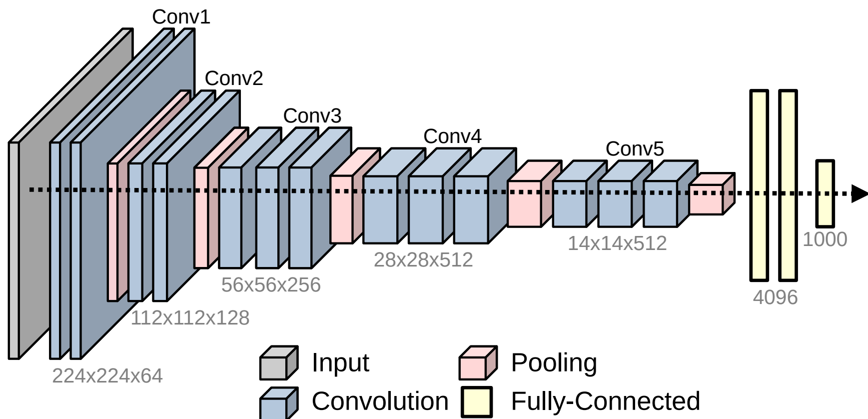
[1]McGill University, Canada

[2]University of Edinburgh, United Kingdom

LATHC Workshop, Feb. 2023

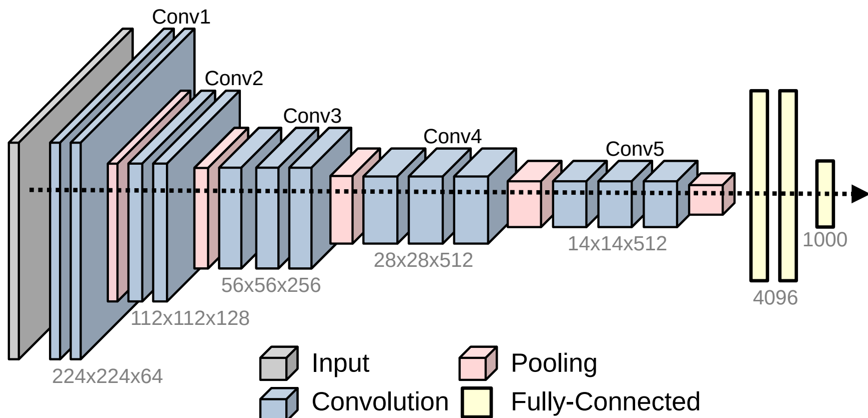
Deep Neural Networks

- Different kinds of layers
- Different layer sizes

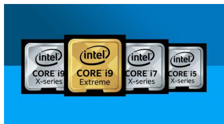


Deep Neural Networks

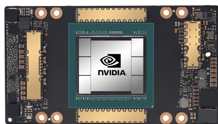
- Different kinds of layers
- Different layer sizes
- Mostly Multiplications and Additions (Good for sharing)



Hardware Accelerators



CPU[1]

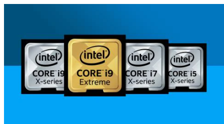


GPU[2]

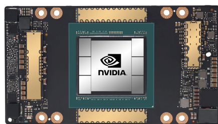
FPGA[3]

ASIC[4]

Hardware Accelerators



CPU[1]



GPU[2]

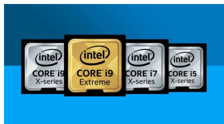
FPGA[3]

ASIC[4]

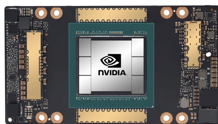
Easy to program

Hard to program

Hardware Accelerators



CPU[1]



GPU[2]

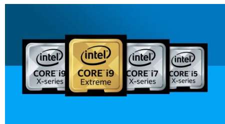
FPGA[3]

ASIC[4]

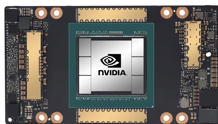
Easy to program ← Hard to program

Low performance → High Performance

Hardware Accelerators



CPU[1]



GPU[2]

FPGA[3]

ASIC[4]

Easy to program ← Hard to program

Low performance → High Performance

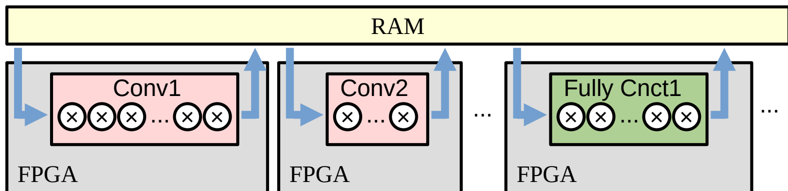
High Flexibility ← Low Flexibility

Fitting Entire Neural Networks into FPGA

Target: Fit layers into fpga but still keep the performance

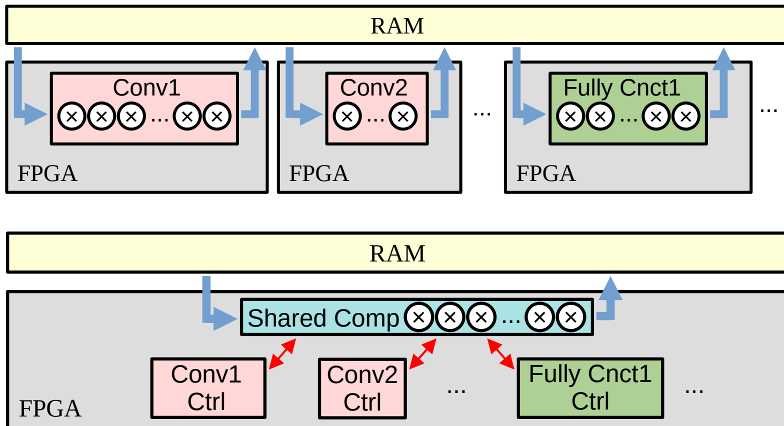
Fitting Entire Neural Networks into FPGA

Target: Fit layers into fpga but still keep the performance

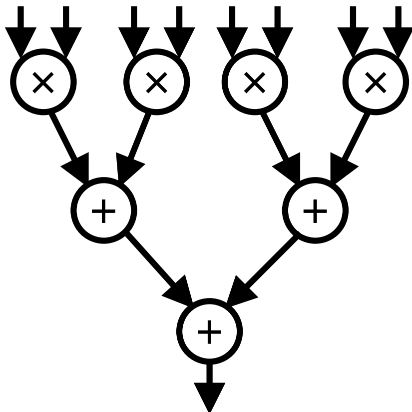


Fitting Entire Neural Networks into FPGA

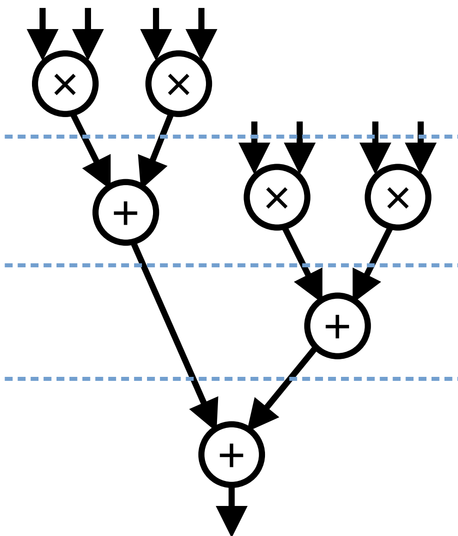
Target: Fit layers into fpga but still keep the performance



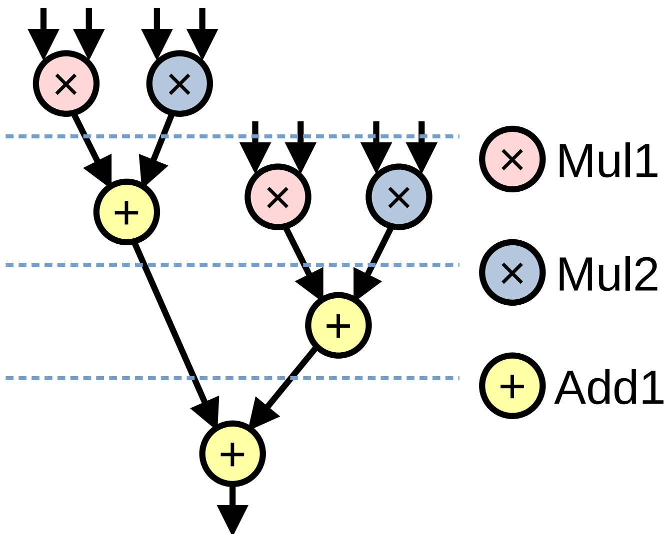
Approach from Existing Tools (Fine-Grained)



Approach from Existing Tools (Fine-Grained)



Approach from Existing Tools (Fine-Grained)



Intel OpenCL FPGA SDK Example

```
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) {
}
```

Matrix multiplications

Arria 10 FPGA usage

# of calls	Resources		
	Logic(%)	RAM(%)	DSP(%)

Intel OpenCL FPGA SDK Example

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

Matrix multiplications

Arria 10 FPGA usage

# of calls	Resources		
	Logic(%)	RAM(%)	DSP(%)
1	23	19	34

Intel OpenCL FPGA SDK Example

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

Matrix multiplications

Arria 10 FPGA usage

# of calls	Resources		
	Logic(%)	RAM(%)	DSP(%)
1	23	19	34
2	32	36	68

Intel OpenCL FPGA SDK Example

```
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  kernel mm(int* A, int*B, int* C, int sz) {
8    mxm(A, B, C, sz);
9    mxm(A, B, C, sz);
10   mxm(A, B, C, sz);}}
```

Matrix multiplications

Arria 10 FPGA usage

# of calls	Resources		
	Logic(%)	RAM(%)	DSP(%)
1	23	19	34
2	32	36	68
3	Out of DSPs		

SHIR Functional Framework

SHIR

Algorithmic Level → Architecture Level → Low-Level IRs → VHDL Files → Synthesizer

Christof Schlaak, Tzung-Han Juang, and Christophe Dubach. Memory-aware functional ir for higher-level synthesis of accelerators. ACM Transactions on Architecture and Code Optimization, 19(2), jan 2022.

SHIR Functional Framework

SHIR

Algorithmic Level → Architecture Level → Low-Level IRs → VHDL Files → Synthesizer

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

Christof Schlaak, Tzung-Han Juang, and Christophe Dubach. Memory-aware functional ir for higher-level synthesis of accelerators. ACM Transactions on Architecture and Code Optimization, 19(2), jan 2022.

SHIR Functional Framework

SHIR

Algorithmic Level → Architecture Level → Low-Level IRs → VHDL Files → Synthesizer

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

```
1 Let mem = MemAlloc(N) in
2   ReduceStm(+, 0, Read(mem, Counter(N))) // Array Sum
```

Christof Schlaak, Tzung-Han Juang, and Christophe Dubach. Memory-aware functional ir for higher-level synthesis of accelerators. ACM Transactions on Architecture and Code Optimization, 19(2), jan 2022.

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.

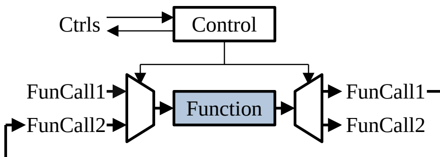
```
1 Let matMulFun = λ MatX, MatY ->
2   Map(λ rowX, colY ->
3     Reduce(+, Map(λ m => Mul(m), Zip(rowX, colY))),
4     MatX, MatY)
5 in
6   scopeBody
```

Shared Function (Coarse-Grained Sharing)

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

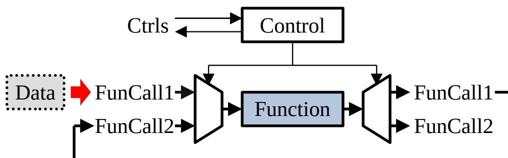
```
1 Let matMulFun = λ MatX, MatY ->
2   Map(λ rowX, colY ->
3     Reduce(+,
4       Map(λ m => Mul(m), Zip(rowX, colY))), MatX, MatY)
5 in
6   C = FunCall(matMulFun, A, B)
7   Out = FunCall(matMulFun, C, D)
```

Shared Function (Coarse-Grained Sharing)



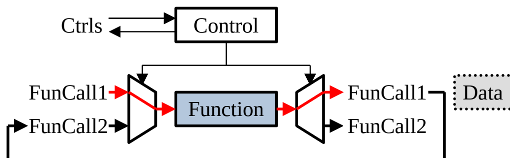

```
1 Let matMulFun = λ MatX, MatY ->
2   Map(λ rowX, colY ->
3     Reduce(+,
4       Map(λ m => Mul(m), Zip(rowX, colY))), MatX, MatY)
5 in
6   C = FunCall(matMulFun, A, B)    // FunCall1
7   Out = FunCall(matMulFun, C, D) // FunCall2
```

Shared Function (Coarse-Grained Sharing)



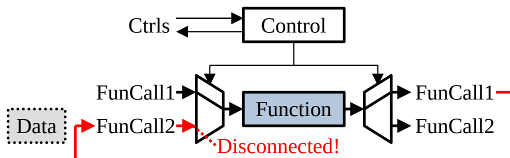

```
Let matMulFun = λ MatX, MatY ->
  Map(λ rowX, colY ->
    Reduce(+,
      Map(λ m => Mul(m), Zip(rowX, colY))), MatX, MatY)
in
  C = FunCall(matMulFun, A, B)    // FunCall1
  Out = FunCall(matMulFun, C, D) // FunCall2
```

Shared Function (Coarse-Grained Sharing)




```
1 Let matMulFun = λ MatX, MatY ->
2   Map(λ rowX, colY ->
3     Reduce(+,
4       Map(λ m => Mul(m), Zip(rowX, colY))), MatX, MatY)
5 in
6   C = FunCall(matMulFun, A, B)    // FunCall1
7   Out = FunCall(matMulFun, C, D) // FunCall2
```

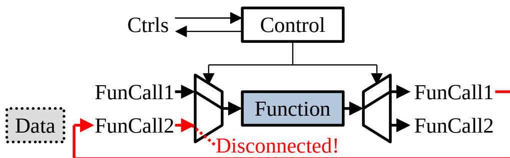
Shared Function (Coarse-Grained Sharing)

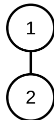



```
1 Let matMulFun = λ MatX, MatY ->
2   Map(λ rowX, colY ->
3     Reduce(+,
4       Map(λ m => Mul(m), Zip(rowX, colY))), MatX, MatY)
5 in
6   C = FunCall(matMulFun, A, B)    // FunCall1
7   Out = FunCall(matMulFun, C, D) // FunCall2
```

Shared Function (Coarse-Grained Sharing)

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





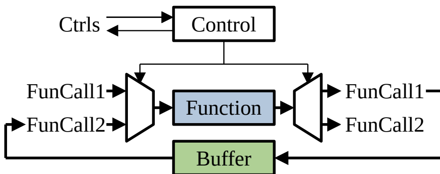
Interference Graph

```
1 Let matMulFun = λ MatX, MatY ->
2   Map(λ rowX, colY ->
3     Reduce(+,
4       Map(λ m => Mul(m), Zip(rowX, colY))), MatX, MatY)
5 in
6   C = FunCall(matMulFun, A, B)    // FunCall1
7   Out = FunCall(matMulFun, C, D) // FunCall2
```

Shared Function (Coarse-Grained Sharing)

```
1 Let matMulFun = λ MatX, MatY ->
2   Map(λ rowX, colY ->
3     Reduce(+,
4       Map(λ m => Mul(m), Zip(rowX, colY))), MatX, MatY)
5 in
6   C = FunCall(matMulFun, A, B)    // FunCall1
7   Out = FunCall(matMulFun, Buffer(C), D) // FunCall2
```

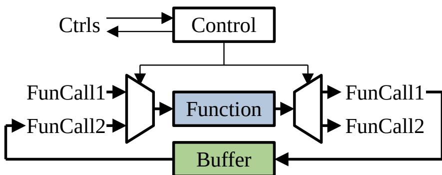
Shared Function (Coarse-Grained Sharing)




```
1 Let matMulFun = λ MatX, MatY ->
2   Map(λ rowX, colY ->
3     Reduce(+,
4       Map(λ m => Mul(m), Zip(rowX, colY))), MatX, MatY)
5 in
6   C = FunCall(matMulFun, A, B)    // FunCall1
7   Out = FunCall(matMulFun, Buffer(C), D) // FunCall2
```

Shared Function (Coarse-Grained Sharing)

Buffering is similar to **Spilling**.



1

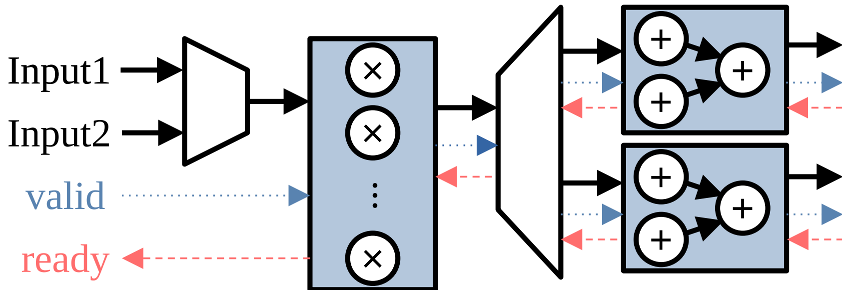
2

Interference Graph

```
1 Let matMulFun = λ MatX, MatY ->
2   Map(λ rowX, colY ->
3     Reduce(+,
4       Map(λ m => Mul(m), Zip(rowX, colY))), MatX, MatY)
5 in
6   C = FunCall(matMulFun, A, B)    // FunCall1
7   Out = FunCall(matMulFun, Buffer(C), D) // FunCall2
```

Low-Level Synthesizer

Synthesizer does not well support detection of duplicated components.

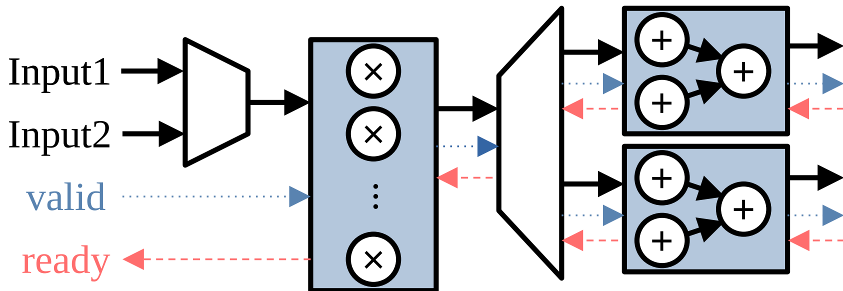


Logic Usage: 33%   DSP Usage: 34%

Sharing multiplication only.

Low-Level Synthesizer

Synthesizer does not well support detection of duplicated components.



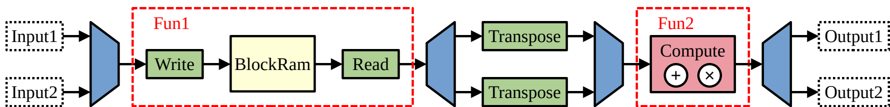
Logic Usage: 33% DSP Usage: 34%

Sharing multiplication only.

Mux Propagation

Multiple function calls also lead to redundant components and wiring.

```
1 Let fun1 = λ x -> BlockRAMBuffer(x) in
2   Let fun2 = λ x -> Compute(x) in
3     output1 = FunCall(fun2, Transpose(FunCall(fun1, input1)))
4     output2 = FunCall(fun2, Transpose(FunCall(fun1, input2)))
```

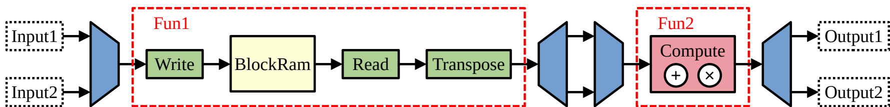


Before optimizations.

Mux Propagation

Solution: Propagate Muxes based on rewrite rules.

```
1 Let fun1 = λ x -> Transpose(BlockRAMBuffer(x)) in
2   Let fun2 = λ x -> Compute(x) in
3     output1 = FunCall(fun2, FunCall(fun1, input1))
4     output2 = FunCall(fun2, FunCall(fun1, input2))
```

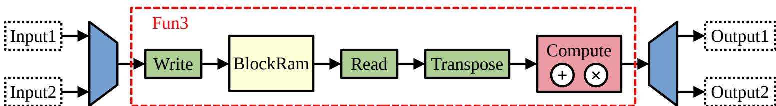


After optimizations.

Mux Propagation

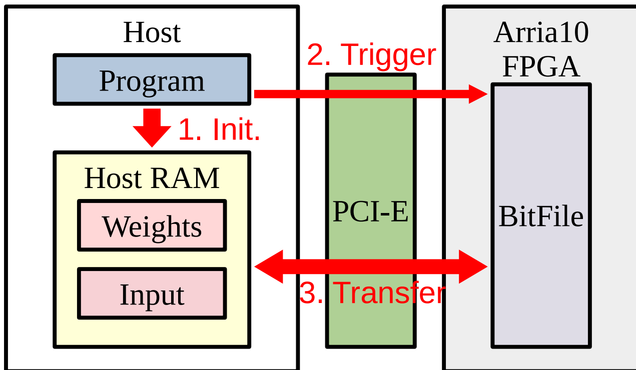
Solution: Merge nearby functions together.

```
1 Let fun3 = λ x -> Compute(Transpose(BlockRAMBuffer(x))) in
2     output1 = FunCall(fun3, input1)
3     output2 = FunCall(un3, input2)
```

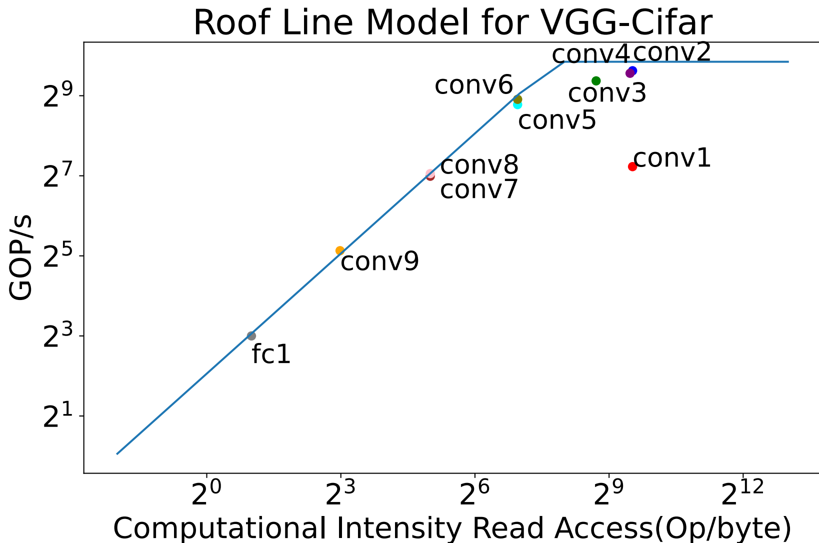


After Further merging.

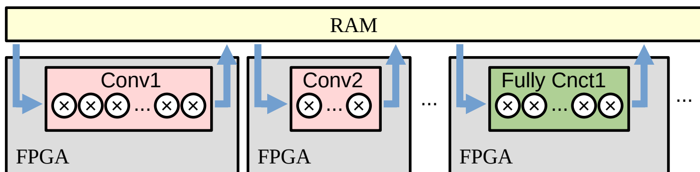
Experiment Setup



VGG16 (Each Layer Independently)

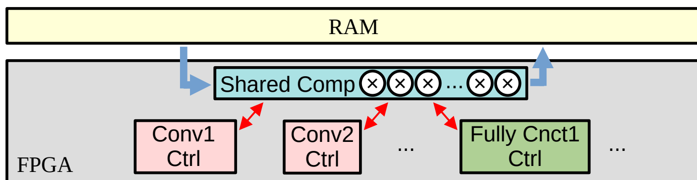


VGG16 (Entire Network)



Setup	Perf.	Resources		
Setup	DSP eff. (%)	Logic (%)	RAM (%)	DSP (%)
One Layer Per FPGA (Max usage)	73	33	22	76

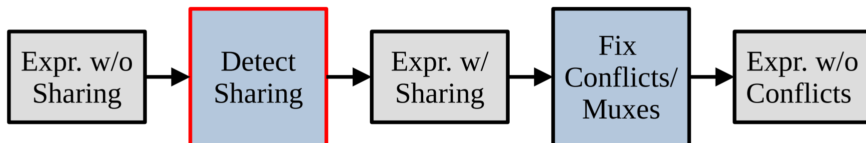
VGG16 (Entire Network)



Setup	Perf.	Resources		
Setup	DSP eff. (%)	Logic (%)	RAM (%)	DSP (%)
One Layer Per FPGA (Max usage)	73	33	22	76
All together. Function Sharing Opt.	73	51	41	76

Future Work

- Evaluation on different neural networks.
- Automatic shared functions detection.



Thank You!

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

References

[1] https://www.intel.com/content/www/us/en/products/details/processors/core/x.html

[2] https://www.nvidia.com/en-gb/graphics-cards/

[3] https://www.xilinx.com/products/boards-and-kits/ek-u1-vcu108-g.html

[4] https://www.ensilica.com/custom-asic/