Let Coarse-Grained Resources Be Shared:
Mapping Entire Neural Networks on FPGAs

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

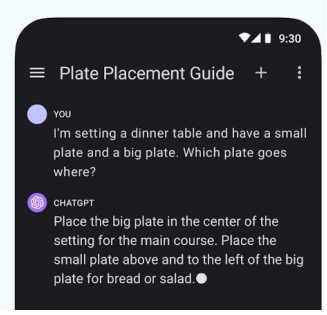
[1]McGill University, Canada

[2]University of Edinburgh, United Kingdom

CASES, Sep. 2023

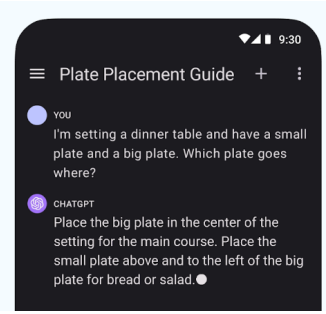
Machine Learning now dominates our lives.

Machine Learning now dominates our lives.

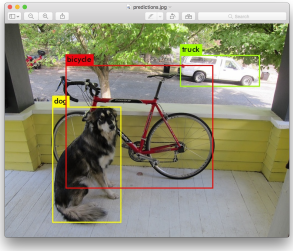


ChatGPT.

Machine Learning now dominates our lives.

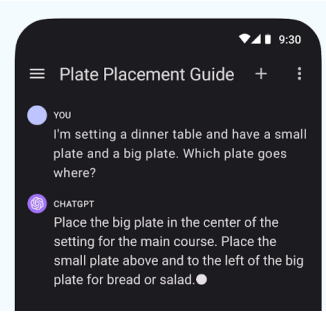


ChatGPT.

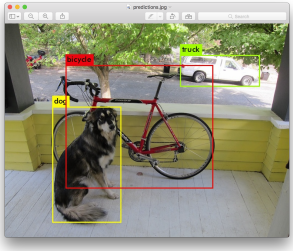


Object Detection.

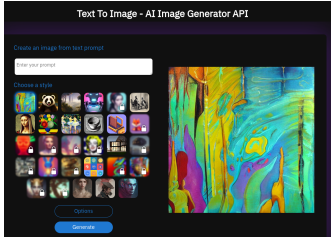
Machine Learning now dominates our lives.



ChatGPT.

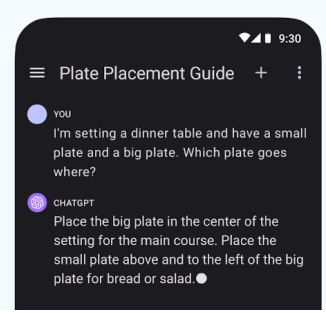


Object Detection.

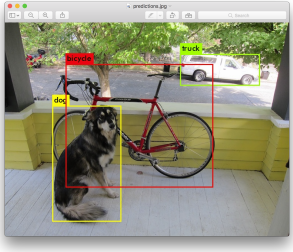


Text To Image.

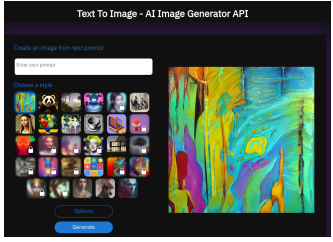
Machine Learning now dominates our lives.



ChatGPT.



Object Detection.

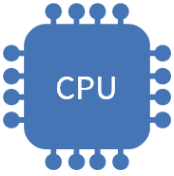


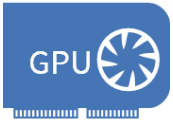
Text To Image.

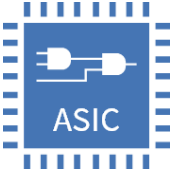
All these use cases come at the cost of heavy computations.

e.g., YoloNet uses billions of multiplications.

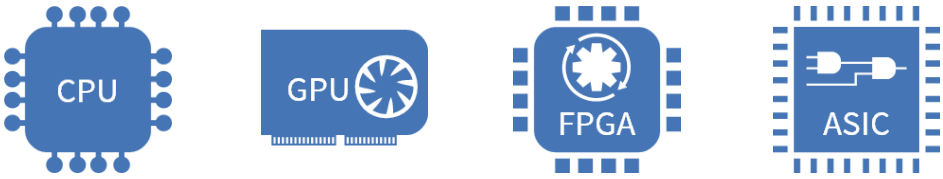
Massive Computation ⇒ Need for Efficient Hardware



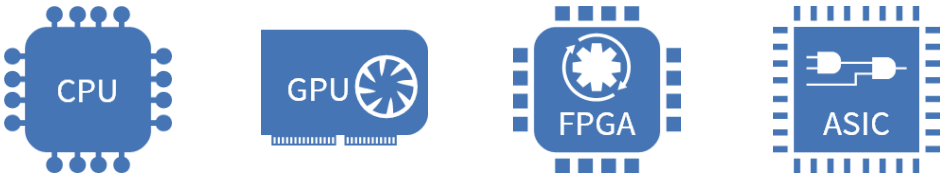




Massive Computation ⇒ Need for Efficient Hardware



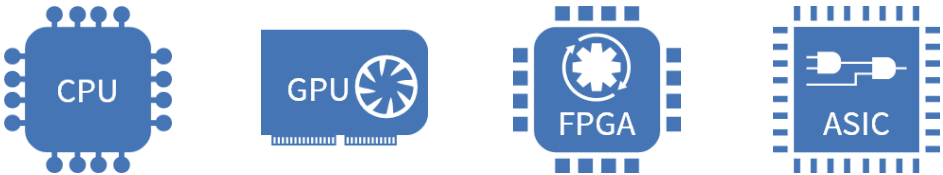
Massive Computation ⇒ Need for Efficient Hardware



Easy ← Coding → Hard

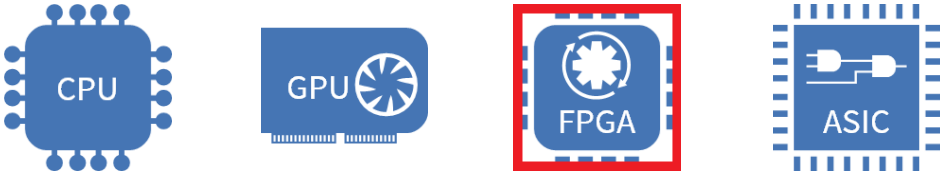
Low ← Efficiency → High

Massive Computation ⇒ Need for Efficient Hardware



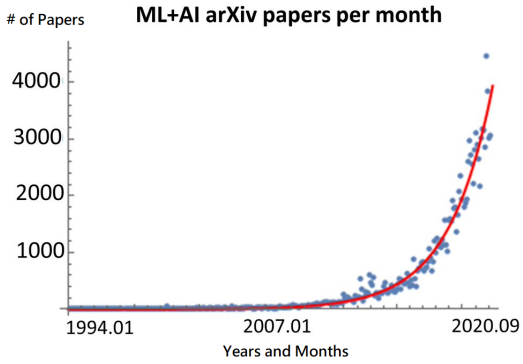
Easy	Coding	Hard
Low	Efficiency	High
High	Re-programmibility	Low

Massive Computation ⇒ Need for Efficient Hardware



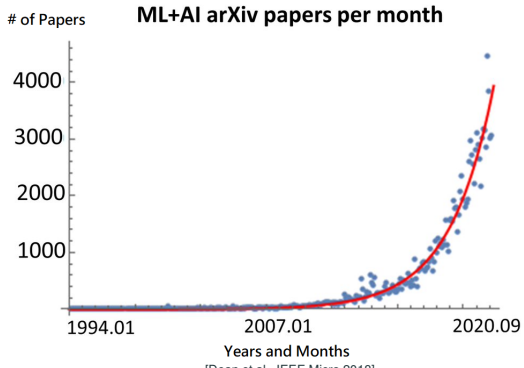
Easy	Coding	Hard
Low	Efficiency	High
High	Re-programmibility	Low

Machine Learning is Fast-Changing



[Dean et al., IEEE Micro 2018]

Machine Learning is Fast-Changing

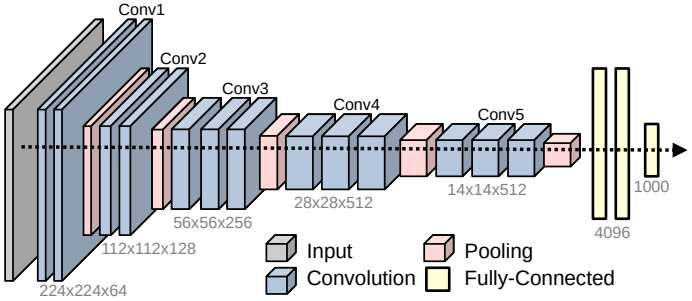


[Dean et al., IEEE Micro 2018]

Problem: Hardware design is still time-consuming. ⇒ Designers are playing catch-up.

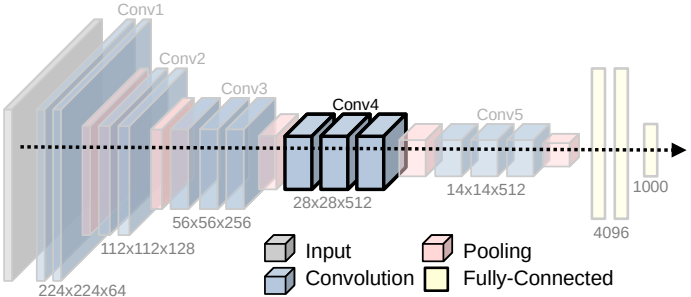
Solution: Automation of accelerator design.

Need for Resource Sharing/Reusing



About 1.7 billion multiplications.

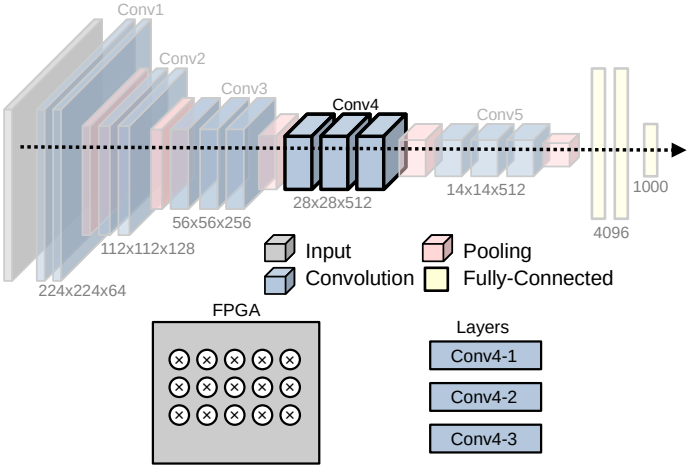
Need for Resource Sharing/Reusing



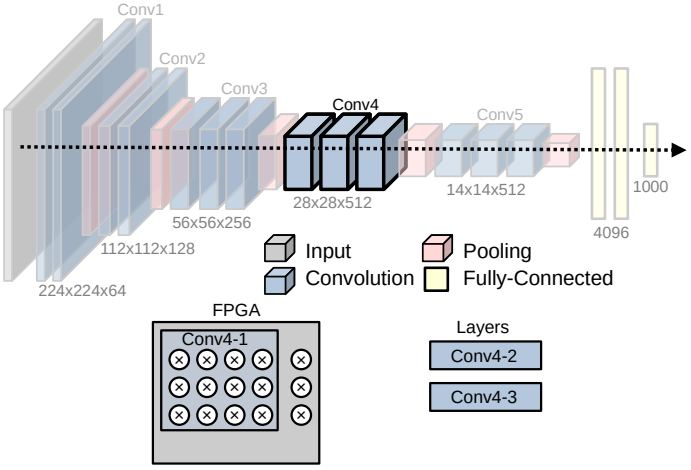
Layers

- Conv4-1
- Conv4-2
- Conv4-3

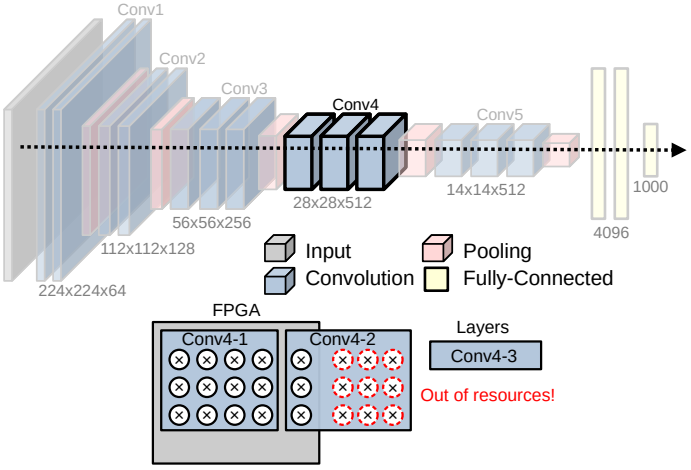
Need for Resource Sharing/Reusing



Need for Resource Sharing/Reusing

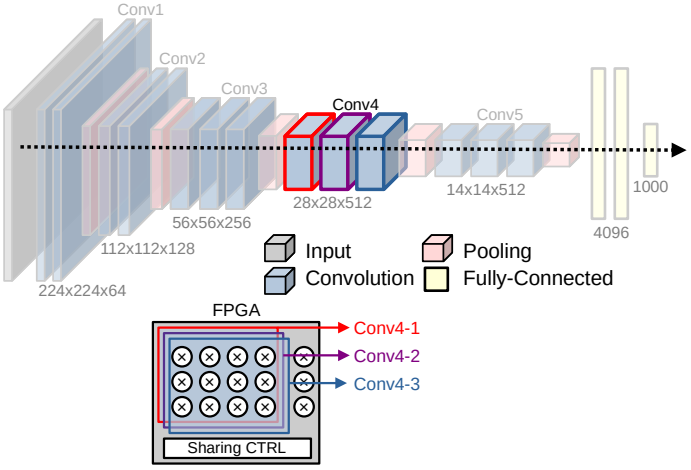


Need for Resource Sharing/Reusing



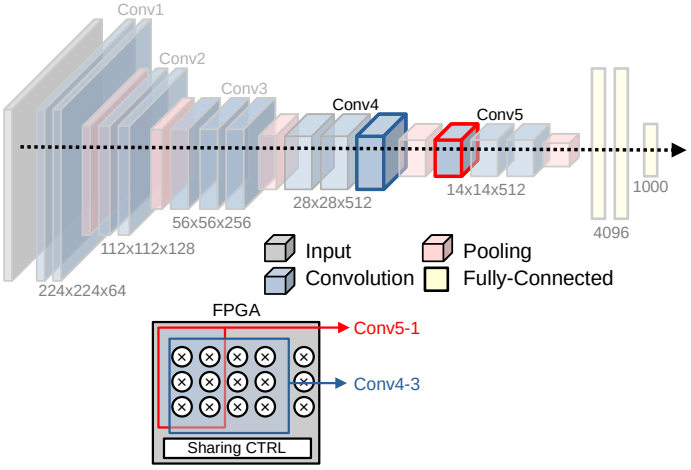
Naive resource allocation does not work.

Need for Resource Sharing/Reusing



Sharing resources is necessary.

Need for Resource Sharing/Reusing



Need extra effort to maintain the same size for different sizes.

How do Existing Tools Fare?

Matrix Multiplications with Intel OpenCL FPGA SDK on Arria 10 FPGA.

```
1  void matMul(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 main(int* A, int* B, int* C, int sz) {
8  }
9
10
```

How do Existing Tools Fare?

Matrix Multiplications with Intel OpenCL FPGA SDK on Arria 10 FPGA.

```
1  void matMul(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 main(int* A, int* B, int* C, int sz) {
8    matMul(A, B, C, sz);}
9
10
```

# of Calls: 1, DSP (Digital Signal Processor) Usage: 512/1518

How do Existing Tools Fare?

Matrix Multiplications with Intel OpenCL FPGA SDK on Arria 10 FPGA.

```
1  void matMul(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 main(int* A, int* B, int* C, int sz) {
8    matMul(A, B, C, sz);
9    matMul(A, B, C, sz);}
10
```

# of Calls: 1, DSP (Digital Signal Processor) Usage: 512/1518

# of Calls: 2, DSP Usage: 1024/1518

How do Existing Tools Fare?

Matrix Multiplications with Intel OpenCL FPGA SDK on Arria 10 FPGA.

```
1  void matMul(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 main(int* A, int* B, int* C, int sz) {
8    matMul(A, B, C, sz);
9    matMul(A, B, C, sz);
10   matMul(A, B, C, sz);}
```

# of Calls: 1, DSP (Digital Signal Processor) Usage: 512/1518

# of Calls: 2, DSP Usage: 1024/1518

# of Calls: 3, DSP Usage: 1536/1518

Functional Approach

Neural Networks are Functional

```
// Fully Connected Layer
FullyConnected(input, weights) =
  ...
```

Neural Networks are Functional

```
// Fully Connected Layer
FullyConnected(input: Array1D[N], weights: Array2D[N, M]) =
  ...
```

Neural Networks are Functional

```
// Fully Connected Layer
FullyConnected(input, weights) =
  Map(λ wgt ->
    // Do dot product of each wgt and input
    weights)
```

Neural Networks are Functional

```
// Fully Connected Layer
FullyConnected(input, weights) =
  Map(λ wgt ->
    Reduce(+, Map(*, Zip(wgt, input))), // Dot Product
    weights)
```

Neural Networks are Functional

```
// Fully Connected Layer
FullyConnected(input, weights) =
  Map(λ wgt ->
    Reduce(+, Map(*, Zip(wgt, input))), // Dot Product
    weights)
// Other Layers
Regression(input, weight) = ...
Act(input) = ...
```

Neural Networks are Functional

```
// Fully Connected Layer
FullyConnected(input, weights) =
  Map(λ wgt ->
    Reduce(+, Map(*, Zip(wgt, input))), // Dot Product
    weights)
// Other Layers
Regression(input, weight) = ...
Act(input) = ...
// Example Network
Regression(Act(FullyConnected(Act(FullyConnected(input, wgt1)), wgt2)), wgt3)
```

Neural Networks are Functional

```
// Fully Connected Layer
FullyConnected(input, weights) =
  Map(λ wgt ->
    Reduce(+, Map(*, Zip(wgt, input))), // Dot Product
    weights)
// Other Layers
Regression(input, weight) = ...
Act(input) = ...
// Example Network
Regression(Act(FullyConnected(Act(FullyConnected(input, wgt1)), wgt2)), wgt3)
```

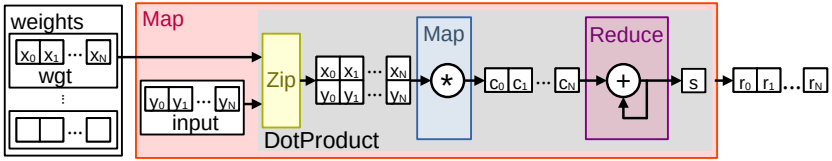
Our backend will transform functional expressions into hardware ...

From Functional Expressions to Hardware

```
1 // Fully Connected Layer
2 FullyConnected(input, weights) =
3   Map(λ wgt ->
4     Reduce(+, Map(*, Zip(wgt, input))), // Dot Product
5     weights)
```

From Functional Expressions to Hardware

```
1 // Fully Connected Layer
2 FullyConnected(input, weights) =
3   Map(λ wgt ->
4     Reduce(+, Map(*, Zip(wgt, input))), // Dot Product
5     weights)
```



Function Sharing with Let, Lambda, and FunCall

- **Let** defines a value under a scope.

```
// Functional
Let x = 3 in
  x + x
```

```
// C-like
int x = 3;
return x + x;
```

Function Sharing with Let, Lambda, and FunCall

- **Let** defines a value under a scope.
- Lambda(λ) defines an anonymous function.

```
// Functional
Let foo = λ x -> x + x in
  ...
```

```
// C-like
int foo(int x) {
  return x + x;
}
...
```

Function Sharing with Let, Lambda, and FunCall

- **Let** defines a value under a scope.
- Lambda(λ) defines an anonymous function.
- FunCall calls a lambda function.

```
// Functional
Let foo = λ x -> x + x in
  FunCall(foo, 3)
```

```
// C-like
int foo(int x) {
  return x + x;
}
foo(3)
```

Function Sharing with Let, Lambda, and FunCall

- **Let** defines a value under a scope.
- Lambda(λ) defines an anonymous function.
- FunCall calls a lambda function.

```
// Functional
Let foo = λ x -> x + x in
  FunCall(foo, 3) +
    FunCall(foo, 2)

```

```
// C-like
int foo(int x) {
  return x + x;
}
foo(3) + foo(2)
```

Function Sharing with Let, Lambda, and FunCall

- **Let** defines a value under a scope.
- Lambda(λ) defines an anonymous function.
- FunCall calls a lambda function.

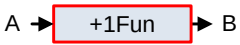
```
// Functional
Let foo = λ x -> x + x in
  FunCall(foo, 3) +
    FunCall(foo, 2)

```

```
// C-like
int foo(int x) {
  return x + x;
}
foo(3) + foo(2)
```

These standard functional primitives will be turned into hardware.

From Functional Sharing to Hardware

```
1 PlusOneFun = λ input -> Map(+1, input))
2
3 B = FunCall(PlusOneFun1, A)
4
5
```

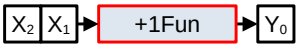


From Functional Sharing to Hardware

```
1 PlusOneFun = λ input -> Map(+1, input))
2
3 B = FunCall(PlusOneFun1, A)
4
5
```

From Functional Sharing to Hardware

```
1 PlusOneFun = λ input -> Map(+1, input))
2
3 B = FunCall(PlusOneFun1, A)
4
5
```



From Functional Sharing to Hardware

```
1 PlusOneFun = λ input -> Map(+1, input))
2
3 B = FunCall(PlusOneFun1, A)
4
5
```

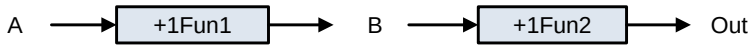
From Functional Sharing to Hardware

```
1 PlusOneFun = λ input -> Map(+1, input))
2
3 B = FunCall(PlusOneFun1, A)
4
5
```

Data comes in a streaming way.

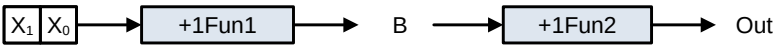
From Functional Sharing to Hardware

```
1 PlusOneFun1 = λ input -> Map(+1, input))
2 PlusOneFun2 = λ input -> Map(+1, input))
3
4 B = FunCall(PlusOneFun1, A)
5 Out = FunCall(PlusOneFun2, B)
```



From Functional Sharing to Hardware

```
1 PlusOneFun1 = λ input -> Map(+1, input))
2 PlusOneFun2 = λ input -> Map(+1, input))
3
4 B = FunCall(PlusOneFun1, A)
5 Out = FunCall(PlusOneFun2, B)
```

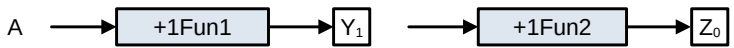


From Functional Sharing to Hardware

```
1 PlusOneFun1 = λ input -> Map(+1, input))
2 PlusOneFun2 = λ input -> Map(+1, input))
3
4 B = FunCall(PlusOneFun1, A)
5 Out = FunCall(PlusOneFun2, B)
```

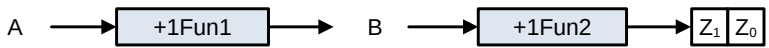
From Functional Sharing to Hardware

```
1 PlusOneFun1 = λ input -> Map(+1, input))
2 PlusOneFun2 = λ input -> Map(+1, input))
3
4 B = FunCall(PlusOneFun1, A)
5 Out = FunCall(PlusOneFun2, B)
```



From Functional Sharing to Hardware

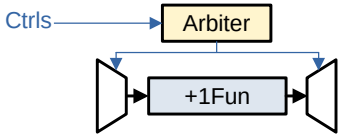
```
1 PlusOneFun1 = λ input -> Map(+1, input))
2 PlusOneFun2 = λ input -> Map(+1, input))
3
4 B = FunCall(PlusOneFun1, A)
5 Out = FunCall(PlusOneFun2, B)
```



Resources are not shared.

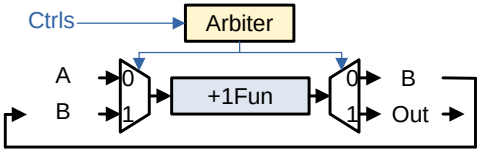
From Functional Sharing to Hardware

```
1 Let PlusOneFun = λ input ->
2   Map(+1, input))
3 in
4   ...
5
```



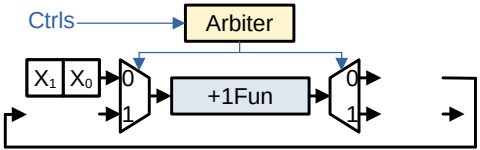
From Functional Sharing to Hardware

```
1 Let PlusOneFun = λ input ->
2   Map(+1, input))
3 in
4   B = FunCall(PlusOneFun, A) // FunCall0
5   Out = FunCall(PlusOneFun, B) // FunCall1
```



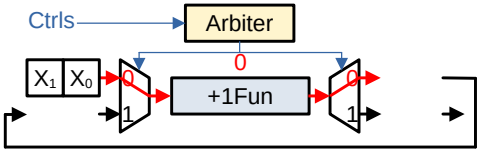
Function Call Conflicts

```
1 Let PlusOneFun = λ input ->
2   Map(+1, input))
3 in
4   B = FunCall(PlusOneFun, A) // FunCall0
5   Out = FunCall(PlusOneFun, B) // FunCall1
```



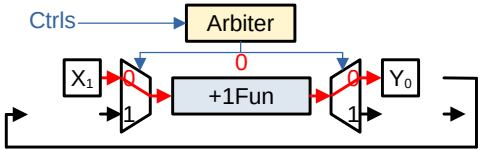
Function Call Conflicts

```
1 Let PlusOneFun = λ input ->
2   Map(+1, input))
3 in
4   B = FunCall(PlusOneFun, A) // FunCall0
5   Out = FunCall(PlusOneFun, B) // FunCall1
```



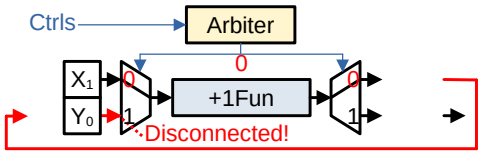
Function Call Conflicts

```
1 Let PlusOneFun = λ input ->
2   Map(+1, input))
3 in
4   B = FunCall(PlusOneFun, A) // FunCall0
5   Out = FunCall(PlusOneFun, B) // FunCall1
```



Function Call Conflicts

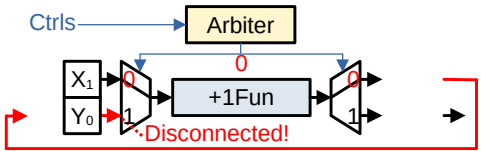
```
Let PlusOneFun = λ input ->
  Map(+1, input))
in
  B = FunCall(PlusOneFun, A) // FunCall0
  Out = FunCall(PlusOneFun, B) // FunCall1
```



The whole stream should be consumed before switching.

Function Call Conflicts

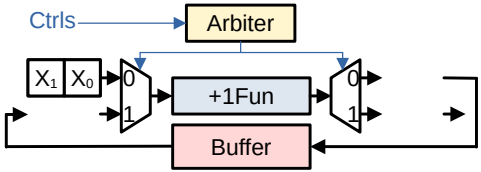
```
1 Let PlusOneFun = λ input ->
2   Map(+1, input))
3 in
4   B = FunCall(PlusOneFun, A) // FunCall0
5   Out = FunCall(PlusOneFun, B) // FunCall1
```



Conflict: Two calls are data-dependant and accessing the same function.

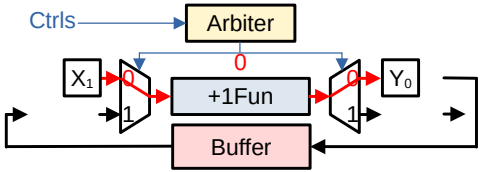
Conflict Resolution

```
1 Let PlusOneFun = λ input ->
2   Map(+1, input))
3 in
4   B = FunCall(PlusOneFun, A) // FunCall0
5   Out = FunCall(PlusOneFun, Buffer(B)) // FunCall1
```



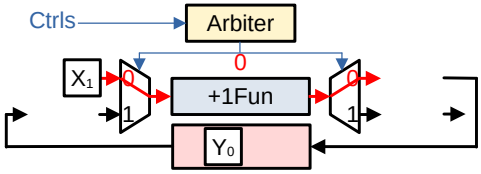
Conflict Resolution

```
1 Let PlusOneFun = λ input ->
2   Map(+1, input))
3 in
4   B = FunCall(PlusOneFun, A) // FunCall0
5   Out = FunCall(PlusOneFun, Buffer(B)) // FunCall1
```



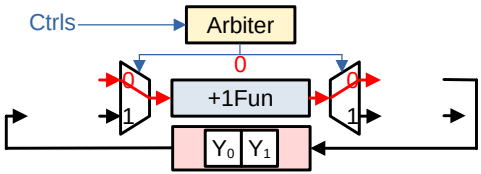
Conflict Resolution

```
1 Let PlusOneFun = λ input ->
2   Map(+1, input))
3 in
4   B = FunCall(PlusOneFun, A) // FunCall0
5   Out = FunCall(PlusOneFun, Buffer(B)) // FunCall1
```



Conflict Resolution

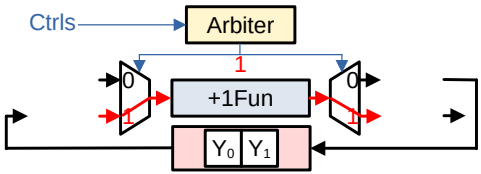
```
1 Let PlusOneFun = λ input ->
2   Map(+1, input))
3 in
4   B = FunCall(PlusOneFun, A) // FunCall0
5   Out = FunCall(PlusOneFun, Buffer(B)) // FunCall1
```



The whole stream is stored in the buffer before switching the arbiter.

Conflict Resolution

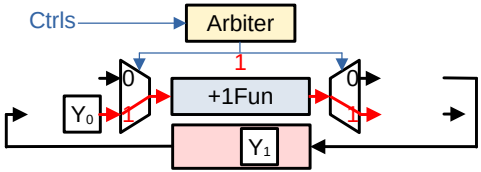
```
1 Let PlusOneFun = λ input ->
2   Map(+1, input))
3 in
4   B = FunCall(PlusOneFun, A) // FunCall0
5   Out = FunCall(PlusOneFun, Buffer(B)) // FunCall1
```



The whole stream is stored in the buffer before switching the arbiter.

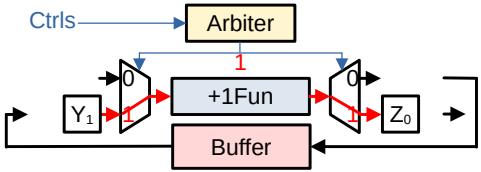
Conflict Resolution

```
1 Let PlusOneFun = λ input ->
2   Map(+1, input))
3 in
4   B = FunCall(PlusOneFun, A) // FunCall0
5   Out = FunCall(PlusOneFun, Buffer(B)) // FunCall1
```



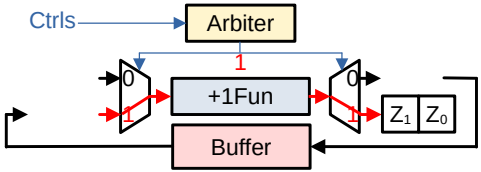
Conflict Resolution

```
1 Let PlusOneFun = λ input ->
2   Map(+1, input))
3 in
4   B = FunCall(PlusOneFun, A) // FunCall0
5   Out = FunCall(PlusOneFun, Buffer(B)) // FunCall1
```



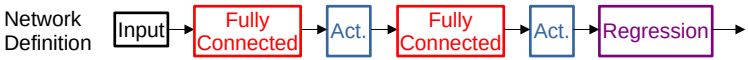
Conflict Resolution

```
1 Let PlusOneFun = λ input ->
2   Map(+1, input))
3 in
4   B = FunCall(PlusOneFun, A) // FunCall0
5   Out = FunCall(PlusOneFun, Buffer(B)) // FunCall1
```

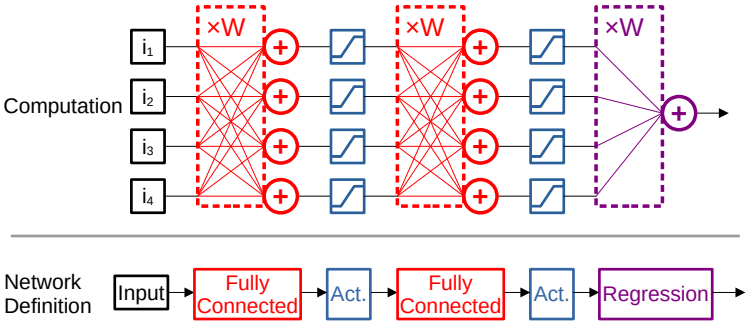


Function Call Conflicts

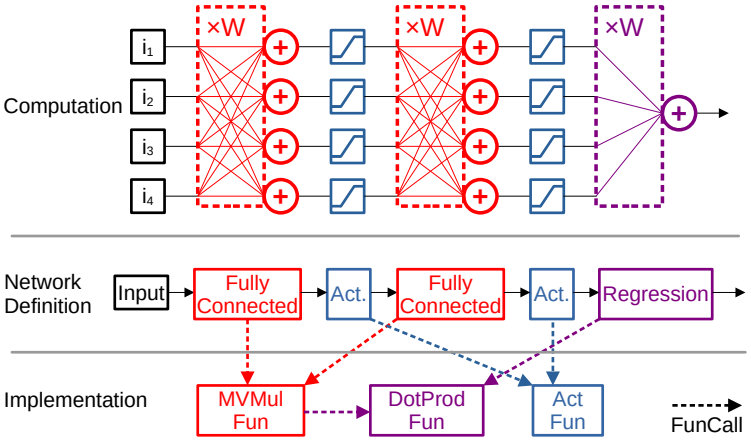
Multiple Functions



Multiple Functions

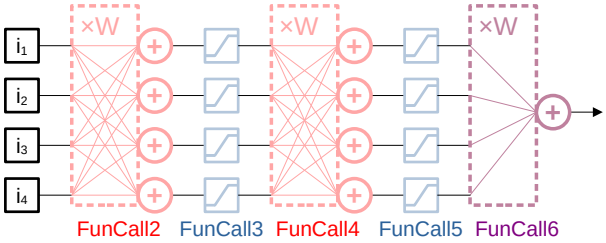


Multiple Functions



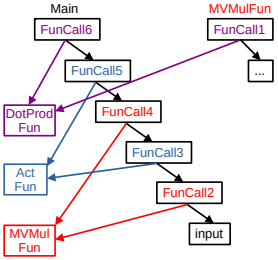
A function can be hierarchical. (MVMul calls DotProd.)

Multiple Functions




```
1 Let DotProdFun = λ  ... in
2   Let MVMulFun = λ ... FunCall¹(DotProdFun, ...) ... in
3     Let ActFun = λ ... in
4       FunCall⁶(DotProdFun,
5         FunCall⁵(ActFun,
6           FunCall⁴(MVMulFun,
7             FunCall³(ActFun,
8               FunCall²(MVMulFun, input, wgt1)), wgt2)),  wgt3)
```

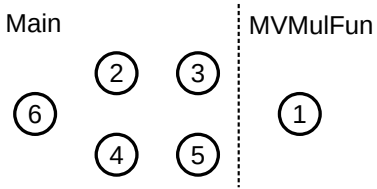
Finding Conflicts

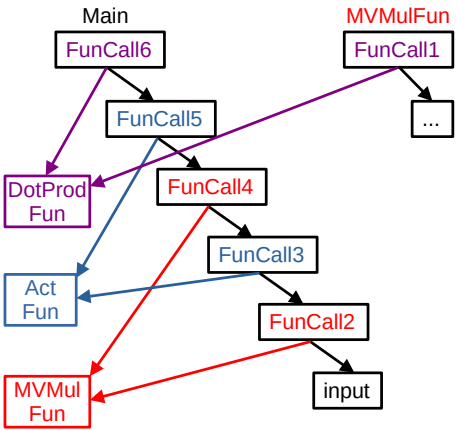



```
1 Let DotProdFun = λ  ... in
2   Let MVMulFun = λ ... FunCall¹(DotProdFun, ...) ... in
3     Let ActFun = λ ... in
4       FunCall⁶(DotProdFun,
5         FunCall⁵(ActFun,
6           FunCall⁴(MVMulFun,
7             FunCall³(ActFun,
8               FunCall²(MVMulFun, input, wgt1)), wgt2)),  wgt3)
```

Finding Conflicts

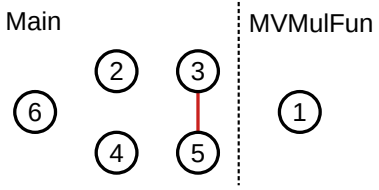
Building one graph per function.

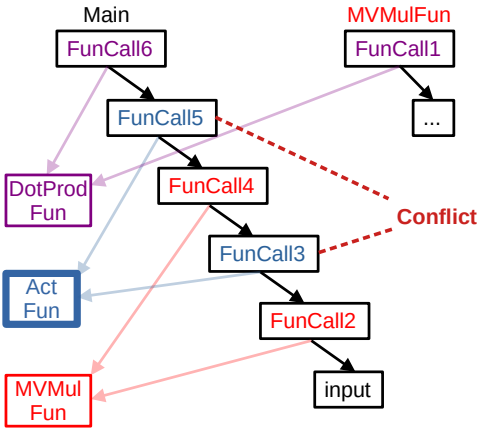




Finding Conflicts

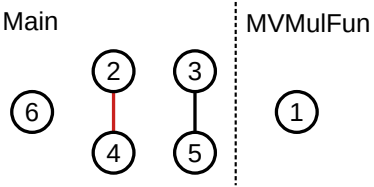
Finding conflicts function by function.

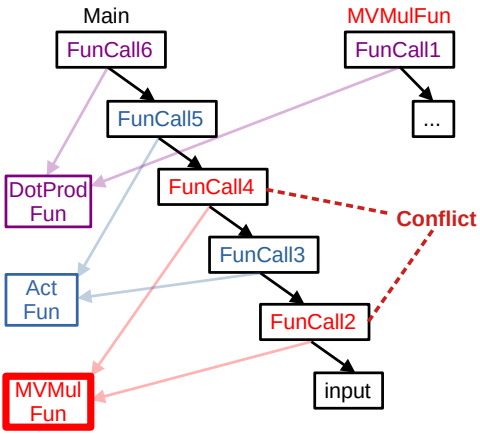




Finding Conflicts

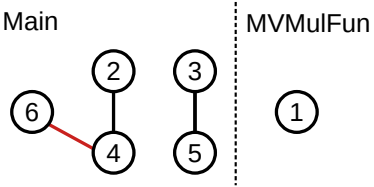
Finding conflicts function by function.

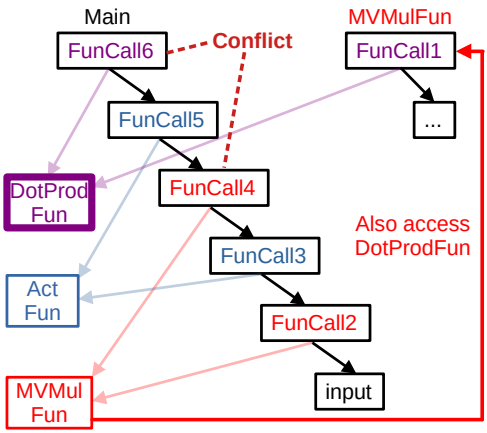




Finding Conflicts

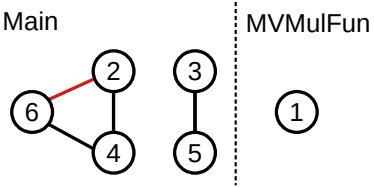
Also considering the indirect calls.

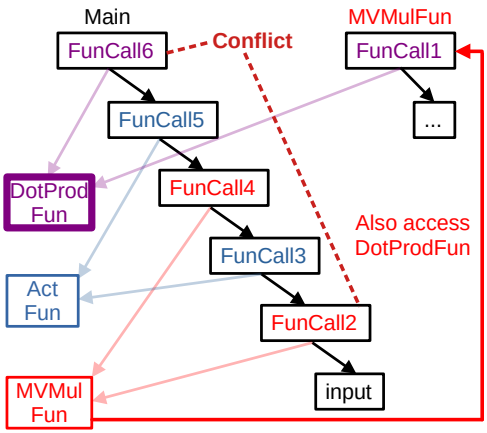




Finding Conflicts

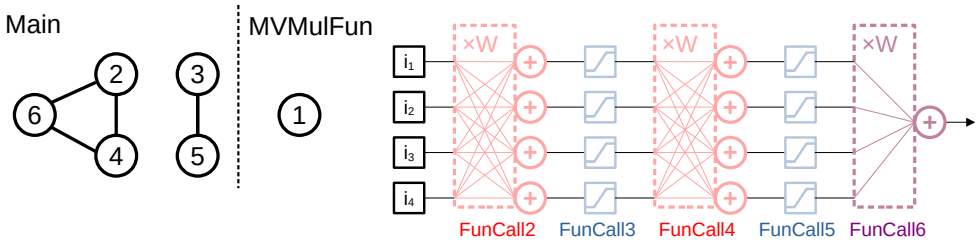
Also considering the indirect calls.





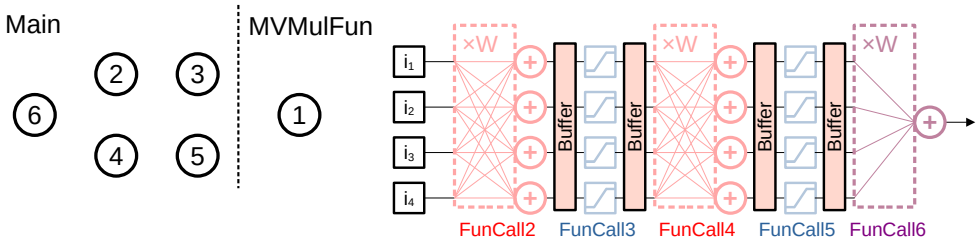
Naive Conflict Resolution

- Call Conflicts = Edges in Interference Graph (IFG).
- Inserting buffers ⇒ Removing conflicts.

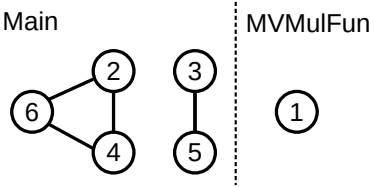


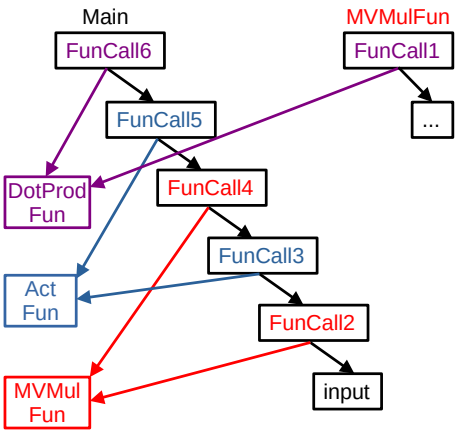
Naive Conflict Resolution

- Call Conflicts = Edges in Interference Graph (IFG).
- Inserting buffers ⇒ Removing conflicts.
- Inserting buffer everywhere (naive way).



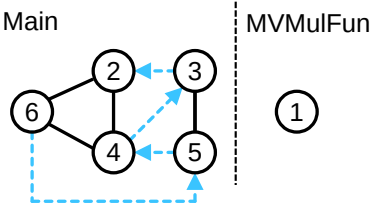
Better Conflict Resolution

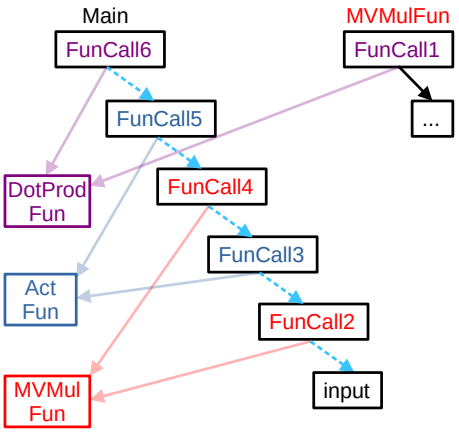




Better Conflict Resolution

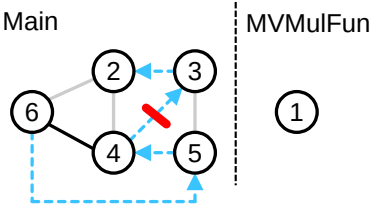
Introduce Call Sequence to IFG.

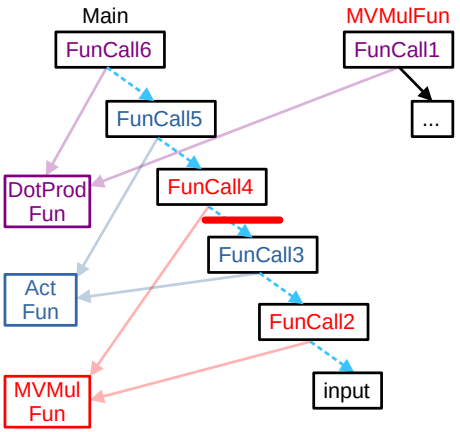




Better Conflict Resolution

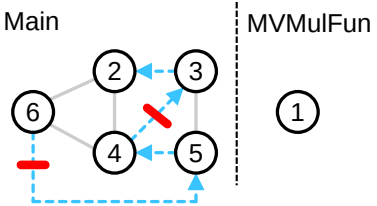
Inserting buffer may remove multiple conflicts. (E.g., removing the edges in a greedy way.)

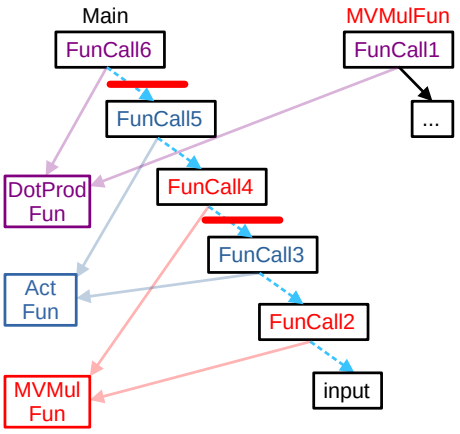




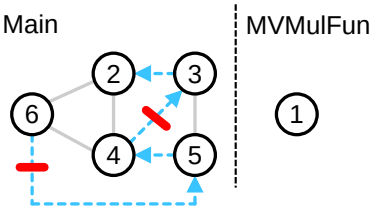
Better Conflict Resolution

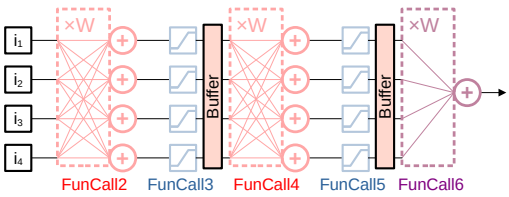
Inserting buffer may remove multiple conflicts. (E.g., removing the edges in a greedy way.)





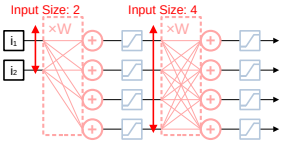
Better Conflict Resolution





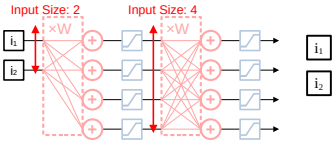
Inserting less buffer ⇒ Less data traffic or resource usage.

Dealing with Different Input Shapes



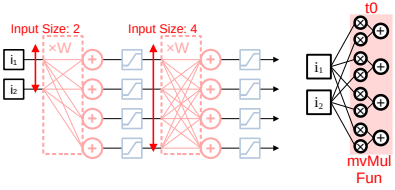
Dealing with Different Input Shapes

Size: 2



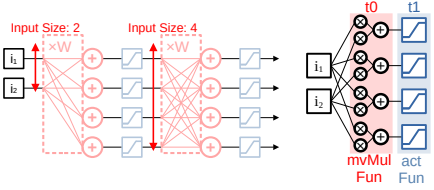
Dealing with Different Input Shapes

Size: 2

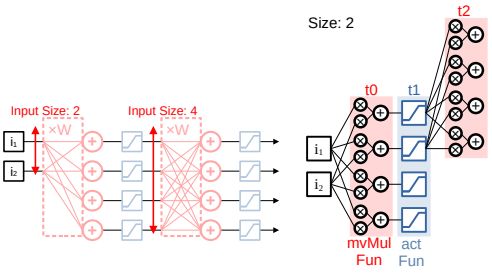


Dealing with Different Input Shapes

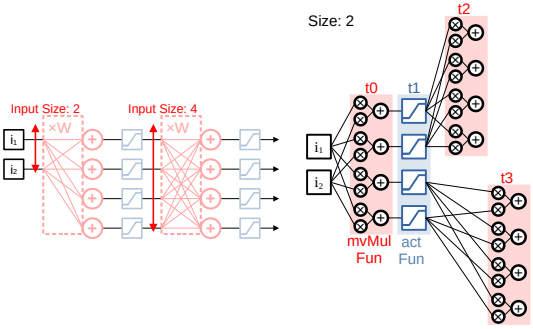
Size: 2



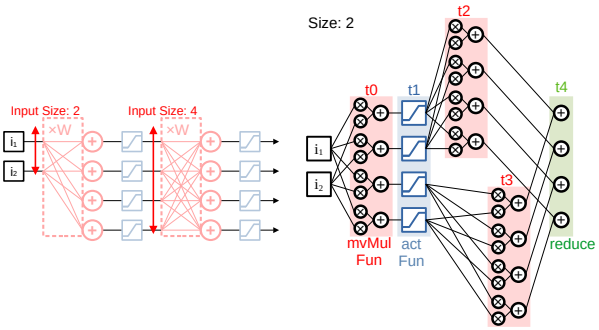
Dealing with Different Input Shapes



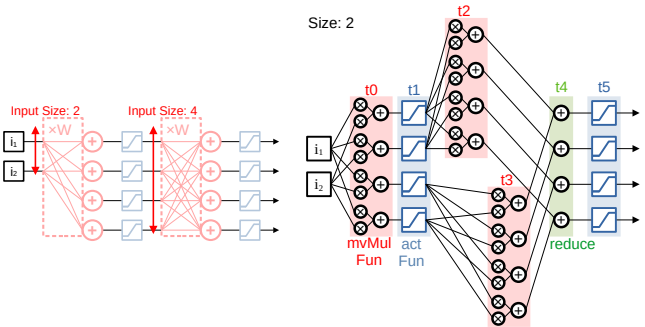
Dealing with Different Input Shapes



Dealing with Different Input Shapes

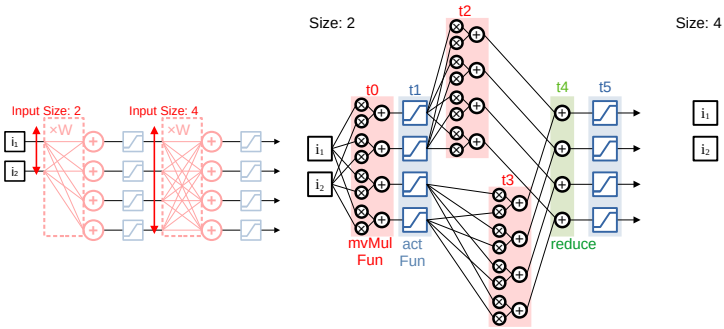


Dealing with Different Input Shapes



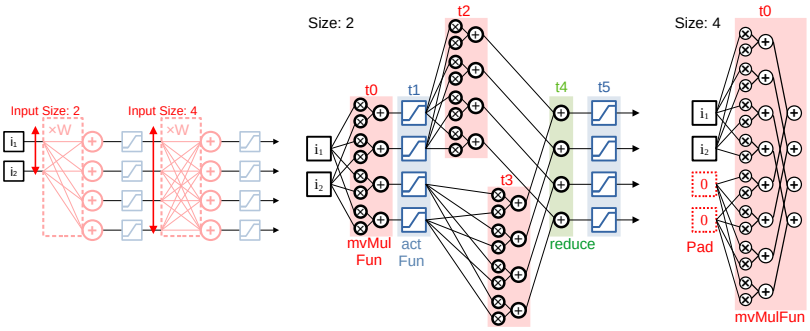
Small tile size ⇒ Extra steps.

Dealing with Different Input Shapes



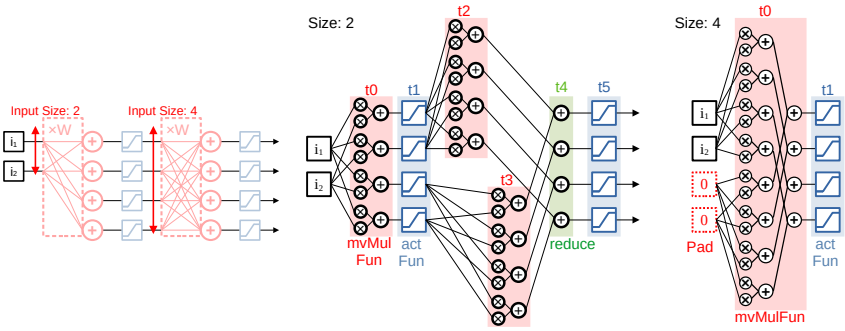
Small tile size ⇒ Extra steps.

Dealing with Different Input Shapes



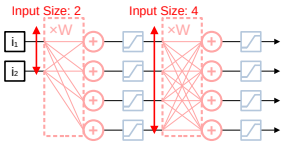
Small tile size ⇒ Extra steps.

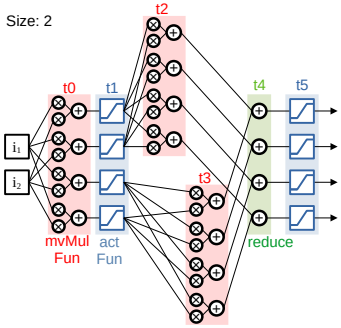
Dealing with Different Input Shapes

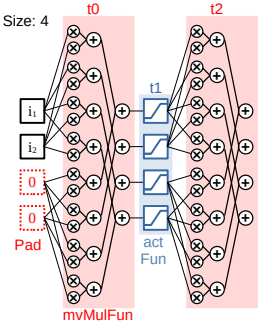


Small tile size ⇒ Extra steps.

Dealing with Different Input Shapes

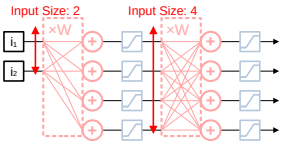


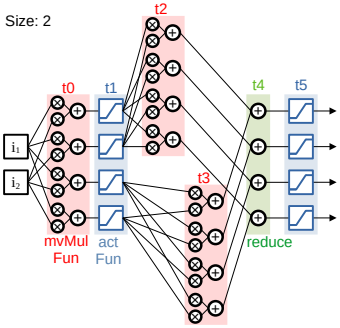




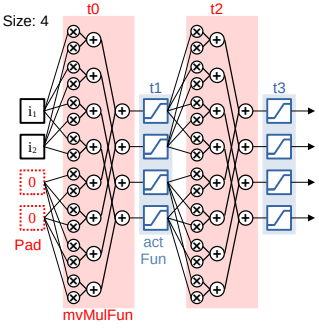
Small tile size ⇒ Extra steps.

Dealing with Different Input Shapes





Small tile size ⇒ Extra steps.



Small tile size ⇒ Extra Redundancy.

Evaluation

Experiment Setup

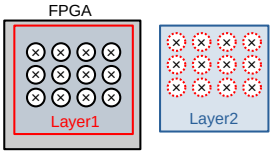
- Including data transfer between host and Arria 10 FPGA.
- Evaluation with neural networks such as VGG and TinyYolo-v2.

Sharing vs Not Sharing

$$\text{DSP eff.} = \frac{\text{Number of Operations}}{\text{Run Cycles} \times \text{Number of DSPs}}$$

Setup (VGG Conv 1 and 2)	DSP eff.	DSP

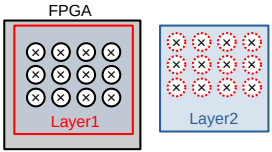
Sharing vs Not Sharing



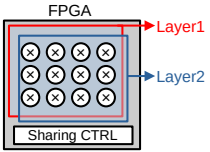
No Sharing.

Setup (VGG Conv 1 and 2)	DSP eff.	DSP
NoSharing	N/A	152%

Sharing vs Not Sharing



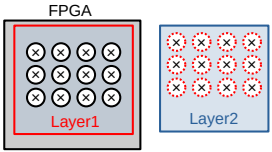
No Sharing.



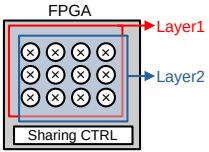
Sharing.

Setup (VGG Conv 1 and 2)	DSP eff.	DSP
NoSharing	N/A	152%
Sharing	86%	76%

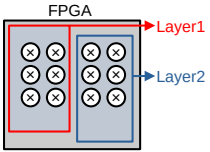
Sharing vs Not Sharing



No Sharing.



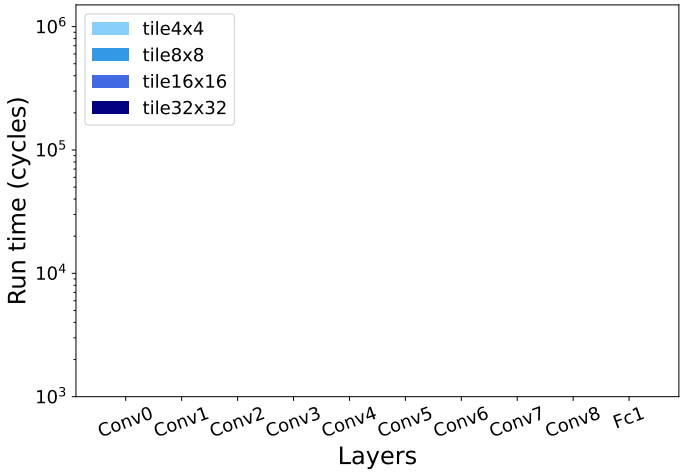
Sharing.



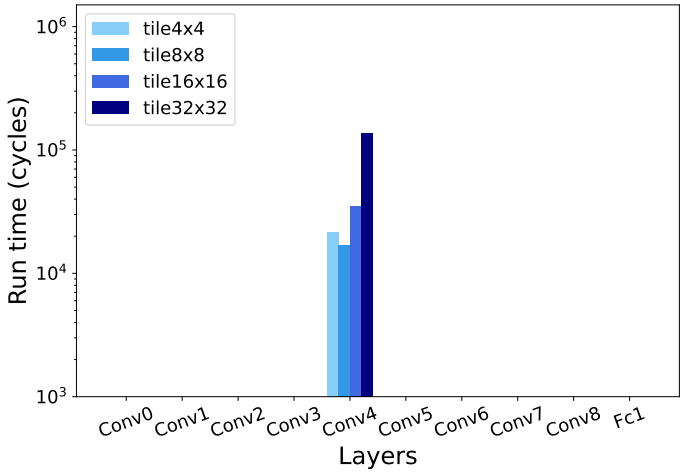
Partitioning.

Setup (VGG Conv 1 and 2)	DSP eff.	DSP
NoSharing	N/A	152%
Sharing	86%	76%
Partitioning	45%	76%

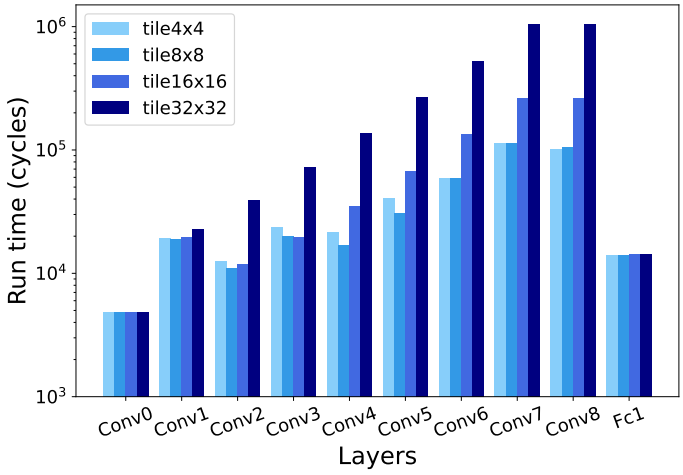
Design Space Exploration with Different Tile Sizes



Design Space Exploration with Different Tile Sizes



Design Space Exploration with Different Tile Sizes



Comparison with State-of-the-Art

	OPs/Cycle	DSPs.
VGG convolutions (224 img. size, int16)		
OpenCL Winograd [1]	1202	544
OpenCL Conv [2]	1678	543
This paper	2225	576

[1] Z. Bai, H. Fan, L. Liu, L. Liu, and D. Wang. An OpenCL-Based FPGA Accelerator with the Winograd's Minimal Filtering Algorithm for Convolution Neuron Networks. IEEE 5th International Conference on Computer and Communications (ICCC). 2019.
[2] H. Li. Acceleration of Deep Learning on FPGA. Ph. D. Dissertation. University of Windsor. 2017.
[3] K. Xu, X. Wang, X. Liu, C. Cao, H. Li, H. Peng, and D. Wang. A dedicated hardware accelerator for real-time acceleration of YOLOv2. Journal of Real-Time Image Processing 18 (2021).

Comparison with State-of-the-Art

	OPs/Cycle	DSPs.
VGG convolutions (224 img. size, int16)		
OpenCL Winograd [1]	1202	544
OpenCL Conv [2]	1678	543
This paper	2225	576

[1] Z. Bai, H. Fan, L. Liu, L. Liu, and D. Wang. An OpenCL-Based FPGA Accelerator with the Winograd's Minimal Filtering Algorithm for Convolution Neuron Networks. IEEE 5th International Conference on Computer and Communications (ICCC). 2019.
[2] H. Li. Acceleration of Deep Learning on FPGA. Ph. D. Dissertation. University of Windsor. 2017.
[3] K. Xu, X. Wang, X. Liu, C. Cao, H. Li, H. Peng, and D. Wang. A dedicated hardware accelerator for real-time acceleration of YOLOv2. Journal of Real-Time Image Processing 18 (2021).

Comparison with State-of-the-Art

	OPs/Cycle	DSPs.
VGG convolutions (224 img. size, int16)		
OpenCL Winograd [1]	1202	544
OpenCL Conv [2]	1678	543
This paper	2225	576
Tiny Yolo v2 full network (416 img. size, int8)		
OpenCL YOLO [3]	2632	884
This paper	3042	1152

[1] Z. Bai, H. Fan, L. Liu, L. Liu, and D. Wang. An OpenCL-Based FPGA Accelerator with the Winograd's Minimal Filtering Algorithm for Convolution Neuron Networks. IEEE 5th International Conference on Computer and Communications (ICCC). 2019.
[2] H. Li. Acceleration of Deep Learning on FPGA. Ph. D. Dissertation. University of Windsor. 2017.
[3] K. Xu, X. Wang, X. Liu, C. Cao, H. Li, H. Peng, and D. Wang. A dedicated hardware accelerator for real-time acceleration of YOLOv2. Journal of Real-Time Image Processing 18 (2021).

Comparison with State-of-the-Art

	OPs/Cycle	DSPs.
VGG convolutions (224 img. size, int16)		
OpenCL Winograd [1]	1202	544
OpenCL Conv [2]	1678	543
This paper	2225	576
Tiny Yolo v2 full network (416 img. size, int8)		
OpenCL YOLO [3]	2632	884
This paper	3042	1152

[1] Z. Bai, H. Fan, L. Liu, L. Liu, and D. Wang. An OpenCL-Based FPGA Accelerator with the Winograd's Minimal Filtering Algorithm for Convolution Neuron Networks. IEEE 5th International Conference on Computer and Communications (ICCC). 2019.
[2] H. Li. Acceleration of Deep Learning on FPGA. Ph. D. Dissertation. University of Windsor. 2017.
[3] K. Xu, X. Wang, X. Liu, C. Cao, H. Li, H. Peng, and D. Wang. A dedicated hardware accelerator for real-time acceleration of YOLOv2. Journal of Real-Time Image Processing 18 (2021).

Conclusion

- Neural Networks ♥ Functional Programming.
- `Let` is all we need for expressing resource sharing.

Conclusion

- Neural Networks ♥ Functional Programming.
- `Let` is all we need for expressing resource sharing.

Future Work

- Automate the process further by identifying shared functions.
- Support PyTorch frontend to get access to more models.