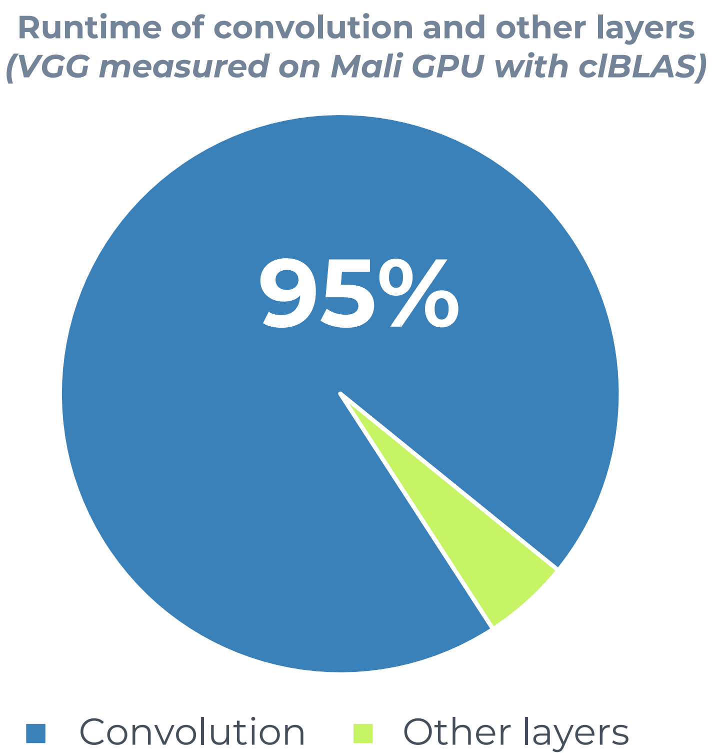
Naums Mogers
Aaron Smith, Dimitrios Vytiniotis
Michel Steuwer, Christophe Dubach
Ryota Tomioka

Towards Mapping Lift To Deep Neural Networks

THE UNIVERSITY of EDINBURGH informatics
EPSRC Centre for Doctoral Training in Pervasive Parallelism
Microsoft Research

Convolution

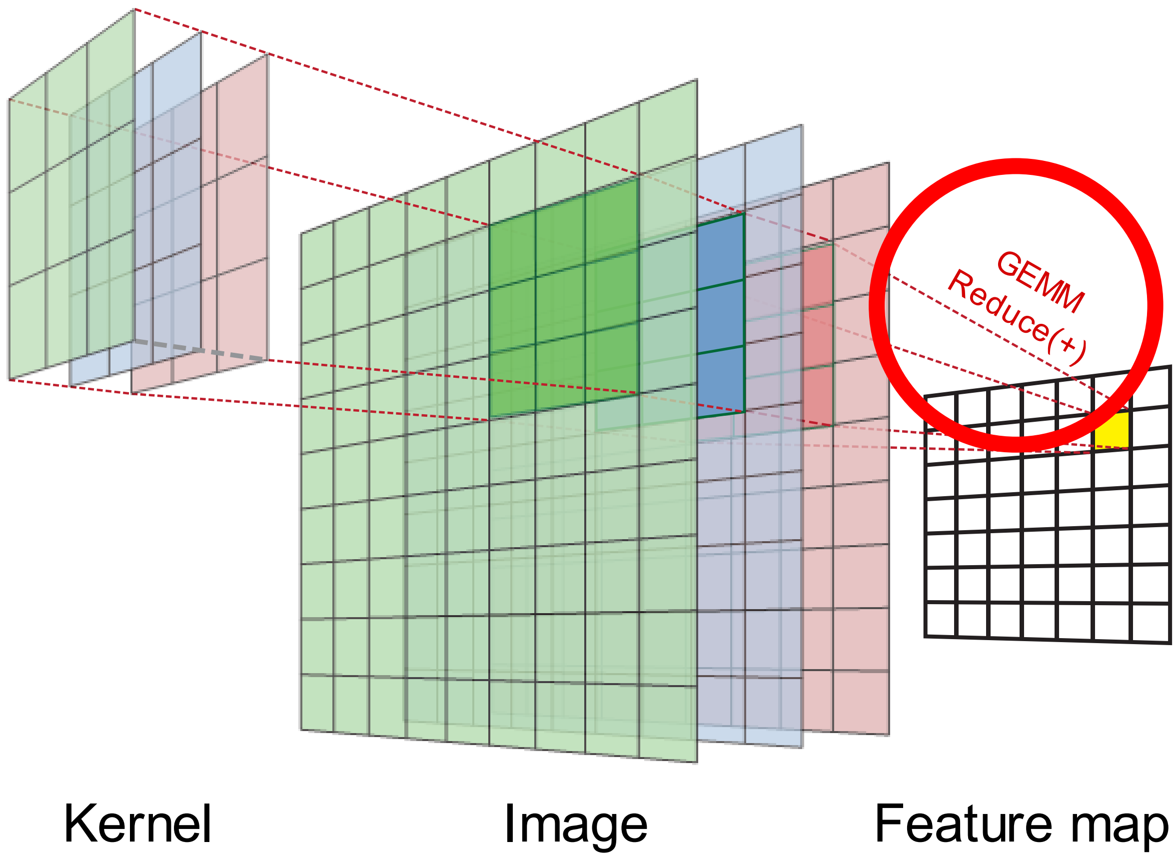
- Many popular NN architectures depend on convolution
  - *AlexNet (2012)*
  - *ZFNet(2013)*
  - *GoogleNet(2014)*
  - *VGGNet(2014)*
  - *ResNet(2015)*



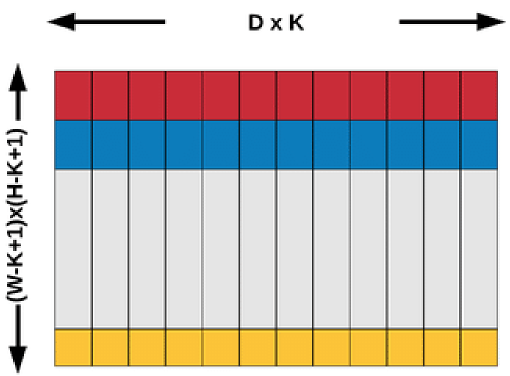
Convolution: algorithms

- Stencils
- *Im2col*
- *Fast Fourier transform*
- *Winograd*

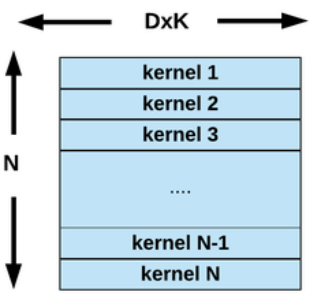
Convolution: stencils



Convolution: im2col



X

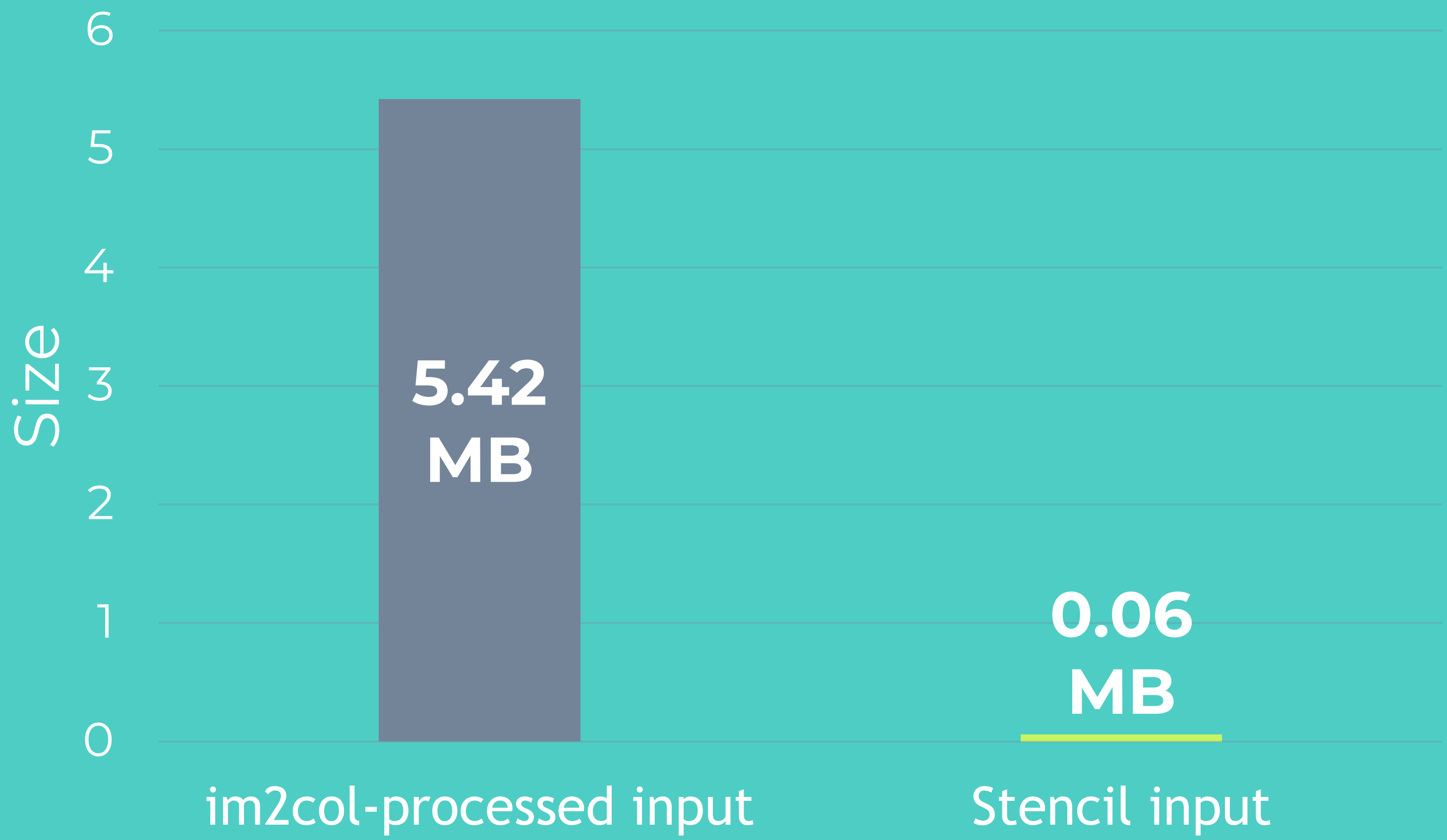


= Convolution result

GEMM
Reduce(+)

Image source: Loukadakis et al (2018)

Input image size in the two methods (1^{st} layer of VGG)



GEMM in neural networks

- Both stencil and im2col methods for convolution rely heavily on GEMM
- GEMM is also the basis of fully connected layers
  - MLPs make for a large portion of server workloads

Hardware accelerators

GPUs

Image source: https://tomshardware.co.uk

FPGAs

Image source: https://eetimes.com

ASICs

Image source: https://blogs.microsoft.com

Examples: TPU, BrainWave, DianNao, Huawei Da Vinci, Movidius Myriad

All accelerators feature:

- Large memory
- High bandwidth
- Multidimensional computational units

Compiling for hardware accelerators

- Multidimensional computational units are exposed in instruction sets:
  - `VVAdd32, VVAdd64`
  - `VVMul32, VVMul64, VVMul128`
  - `MVMul32, MVMul64, MVMul128`

- Finding opportunities to use these primitives is challenging

Compiling for hardware accelerators

- Using built-in primitives is made harder by:
  - **Other optimisations**
    - Parallelisation
    - Tiling *(for limited shared memory)*
    - Memory coalescing
    - Prefetching
    - *(etc)*
  - **Individual device characteristics**

Compiling for hardware accelerators

GEMM

```
for (int i = 0; i < M; i++) {
  for (int j = 0; j < N; j++) {
    for (int k = 0; k < K; k++) {
      temp[k + K*N*i + K*j] =
        A[k + K*i] * B[k + K*j];
    }
    for (int k = 0; k < K; k++) {
      C[j + N*i] +=
        temp[k + K*N*i + K*j];
    }}}
```

Compiling for hardware accelerators

GEMM

```
for (int i = 0; i < M; i++) {
  for (int j = 0; j < N; j++) {
    for (int k = 0; k < K; k++) {
      temp[k + K*N*i + K*j] =
        A[k + K*i] * B[k + K*j];
    }
    for (int k = 0; k < K; k++) {
      C[j + N*i] +=
        temp[k + K*N*i + K*j];
    }}}
```

GEMMtiled

```
for (int i = 0; i < M; i++) {
  for (int j = 0; j < N; j++) {
    for (int k = 0; k < K/4; k++) {
      for (int l = 0; l < 4; l++) {
        temp[l + k*4 + K*N*i + K*j] =
          A[l + k*4 + K*i] * B[l + k*4 + K*j];
      }
    }
    for (int k = 0; k < K; k++) {
      C[j + N*i] +=
        temp[k + K*N*i + K*j];
    }}}
```

Questions:

Data contiguity

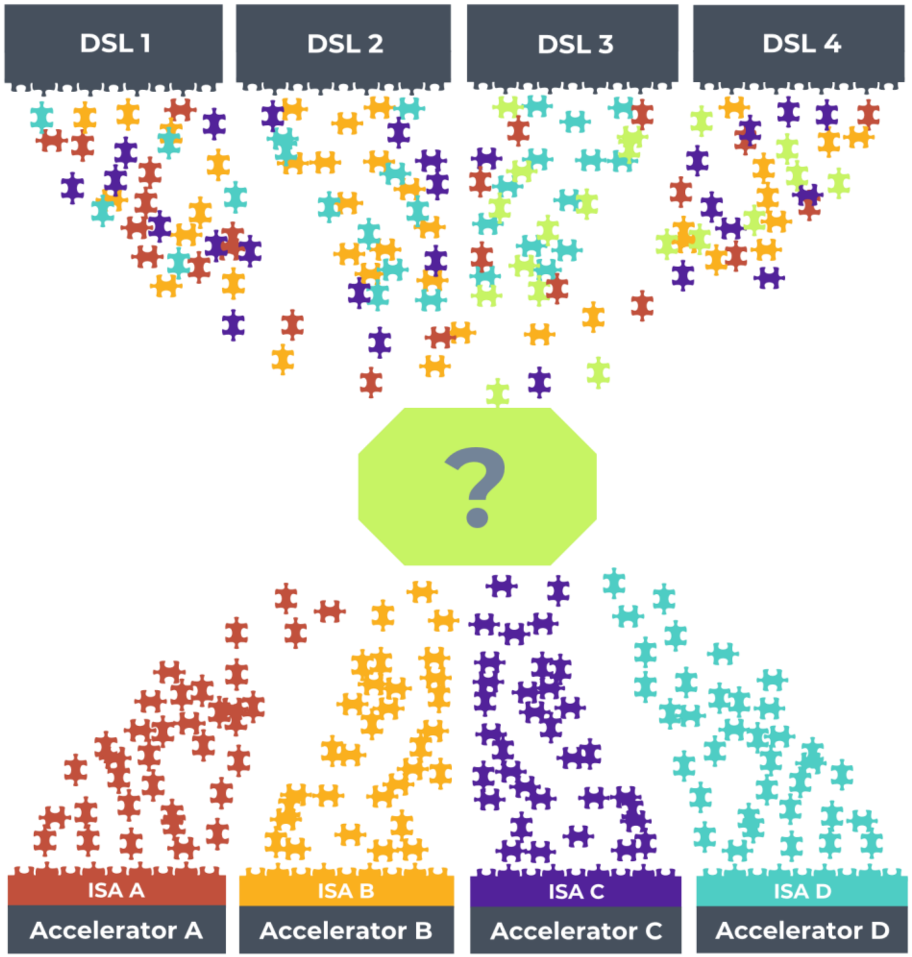
Data size

Efficient memory accesses

Data type

The problem

- Combine device-specific operators optimally
- Design a performance portable approach
- Automate and abstract the process from the user



Lift: the approach

1. Separate algorithm (WHAT) from implementation (HOW)

GEMM imperative

```
for (int i = 0; i < M; i++) {
  for (int j = 0; j < N; j++) {
    for (int k = 0; k < K; k++) {
      temp[k + K*N*i + K*j] =
        A[k + K*i] * B[k + K*j];
    }
    for (int k = 0; k < K; k++) {
      C[j + N*i] +=
        temp[k + K*N*i + K*j];
    }}}
```

→

GEMM functional

```
A >> Map(Arow =>
  B >> Map(Bcol =>
    Zip(Arow, Brow) >>
    Map(ScalarMul) >>
    Reduce(0, Add)
  )
)
```

Lift: the approach

2. Detect and rewrite patterns

GEMM$^{\text{functional}}$

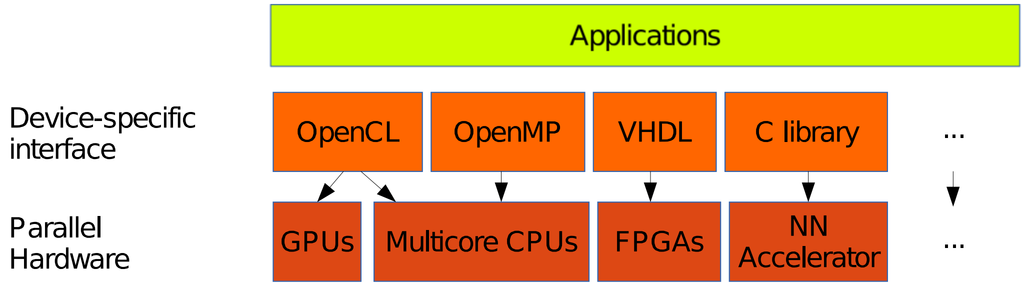
```
A >> Map(Arow =>
  B >> Map(Bcol =>
    Zip(Arow, Brow) >>
    Map(ScalarMul) >>
    Reduce(0, Add)
  )
)
```

GEMM$^{\text{optimised}}$

```
MMul(A, B)
```

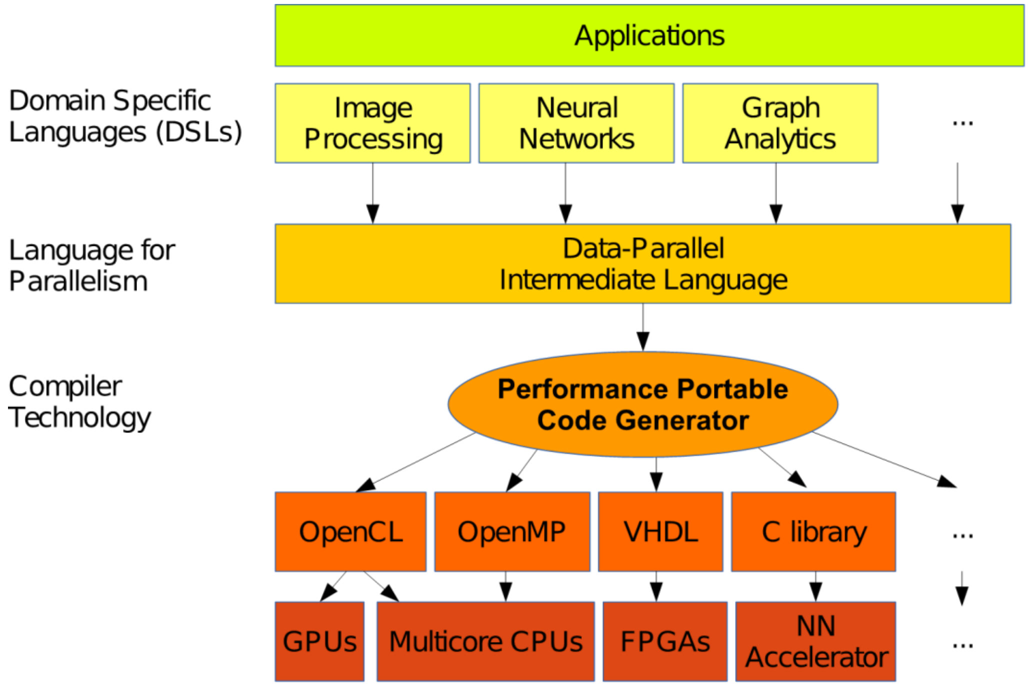
Lift in context

Current landscape



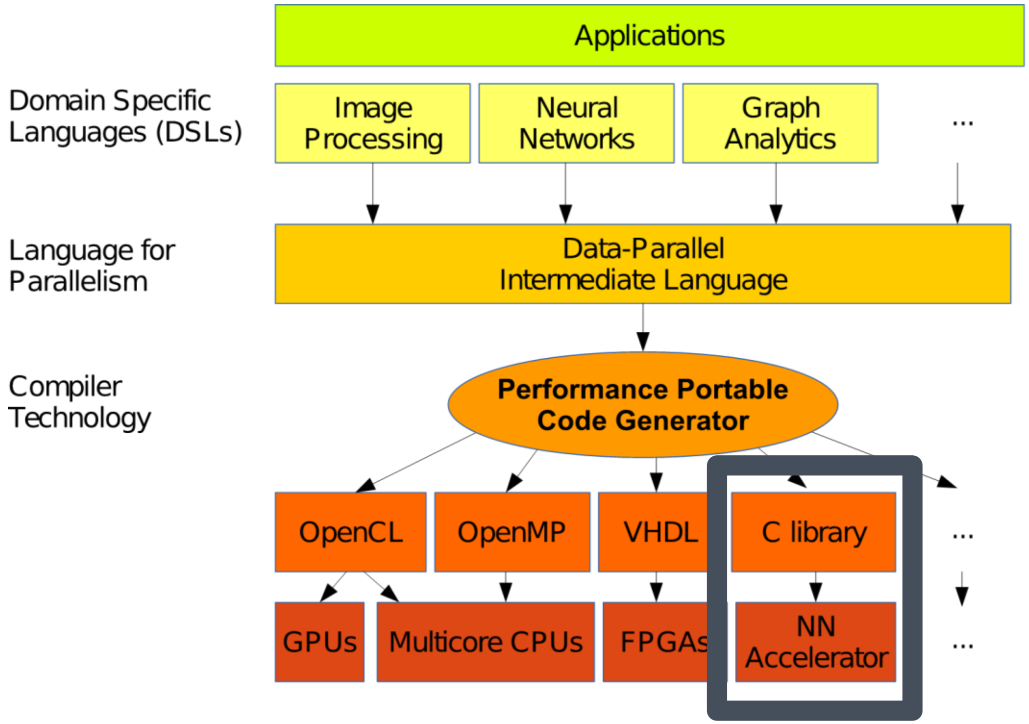
Lift in context

The goal



Lift in context

The focus of this talk



Lift: IR

- Data types
  - `Int, Float8 / Float16 / Float32, Arrays`
- Algorithmic patterns
  - `Map, Slide, Reduce, Zip, Join, Split`
- Address space operators
  - `toChip, toDram, toOutput`
- Arithmetic operators
  - `ScalarAdd, VVAdd, MVAdd, MMAdd`
  - `ScalarMul, VVMul, MVMul, MMMul`
  - `VVRelu, VVTanh`

Lift: example rewrite rules

Split-join rule

```
Map(f)
   ↧   ↧   ↧
Split(n) >>
Map(Map(f)) >>
Join
```

Map fusion rule

```
Map(f) >> Map(g)
   ↧   ↧   ↧
Map(f >> g)
```

GEMV rule

```
matrix >> Map(row =>
  VVMul(row, vector) >>
  Reduce(ScalarAdd, 0))
      ↧   ↧   ↧
MVMul(matrix, vector)
```

Rewrite rule system

- Generic and customisable
- 3 levels: DSL, algorithmic, hardware
- Extensible

Lift: rewriting

A fully connected layer

```
for (int i = 0; i < n_neurons; i++) {
  for (int j = 0; j < x_length; j++) {
    temp[j + x_length*i] =
      X[j] * W[j + x_length*i];
  }

  OUT[i] = B[i];
  for (int j = 0; j < x_length; j++) {
    OUT[j] += temp[j + x_length*i];
  }
}
```

?

```
temp = MVMul(W, X);
OUT = VVAdd(temp, B);
```

Lift: rewriting

```
Zip(
  W,
  B) >>
Map((neuronW, neuronB) =>
  VVMul(neuronW, toChip(X)) >>
  Reduce(ScalarAdd, neuronB)) >>
VVRelu() >> toOutput()
```

↑

```
for (int i = 0; i < n_neurons; i++) {
  for (int j = 0; j < x_length; j++) {
    temp[j + x_length*i] =
      X[j] * W[j + x_length*i];
  }

  OUT[i] = B[i];
  for (int j = 0; j < x_length; j++) {
    OUT[j] += temp[j + x_length*i];
  }
}
```

Lift: rewriting

```
Zip(
  W,
  B) >>
Map((neuronW, neuronB) =>
  VVMul(neuronW, toChip(X)) >>
  Reduce(ScalarAdd, neuronB)) >>
VVRelu() >> toOutput()
```

```
// GEMV rewrite
matrix >> Map(row =>
  VVMul(row, vector) >>
  Reduce(ScalarAdd, 0))
       ↧    ↧    ↧
MVMul(matrix, vector)
```

```
for (int i = 0; i < n_neurons; i++) {
  for (int j = 0; j < x_length; j++) {
    temp[j + x_length*i] =
      X[j] * W[j + x_length*i];
  }

  OUT[i] = B[i];
  for (int j = 0; j < x_length; j++) {
    OUT[j] += temp[j + x_length*i];
  }
}
```

Lift: rewriting

```
Zip(
  W,
  B) >>
Map((neuronW, neuronB) =>
  VVMul(neuronW, toChip(X)) >>
  Reduce(ScalarAdd, neuronB)) >>
VVRelu() >> toOutput()
```

`<Extract Initializer From Reduce>`

```
for (int i = 0; i < n_neurons; i++) {
  for (int j = 0; j < x_length; j++) {
    temp[j + x_length*i] =
      X[j] * W[j + x_length*i];
  }

  OUT[i] = B[i];
  for (int j = 0; j < x_length; j++) {
    OUT[j] += temp[j + x_length*i];
  }
}
```

Lift: rewriting

```
Zip(
  W,
  B) >>
Map((neuronW, neuronB) =>
  VVMul(neuronW, toChip(X)) >>
  Reduce(ScalarAdd, neuronB)) >>
VVRelu() >> toOutput()
```

→

```
Zip(
  W,
  B) >>
Map((neuronW, neuronB) =>
  VVMul(neuronW, toChip(X)) >>
  Reduce(ScalarAdd, 0) >>
  ScalarAdd(neuronB)) >>
VVRelu() >> toOutput()
```

```
for (int i = 0; i < n_neurons; i++) {
  for (int j = 0; j < x_length; j++) {
    temp[j + x_length*i] =
      X[j] * W[j + x_length*i];
  }

  OUT[i] = B[i];
  for (int j = 0; j < x_length; j++) {
    OUT[j] += temp[j + x_length*i];
  }
}
```

```
for (int i = 0; i < n_neurons; i++) {
  for (int j = 0; j < x_length; j++) {
    temp[j + x_length*i] =
      X[j] * W[j + x_length*i];
  }

  OUT[i] = 0;
  for (int j = 0; j < x_length; j++) {
    OUT[j] += temp[j + x_length*i];
  }
  OUT[i] += B[i];
}
```

Lift: rewriting

```
Zip(
  W,
  B) >>
Map((neuronW, neuronB) =>
  VVMul(neuronW, toChip(X)) >>
  Reduce(ScalarAdd, 0) >>
  ScalarAdd(neuronB)) >>
VVRelu() >> toOutput()
```

```
for (int i = 0; i < n_neurons; i++) {
  for (int j = 0; j < x_length; j++) {
    temp[j + x_length*i] =
      X[j] * W[j + x_length*i];
  }

  OUT[i] = 0;
  for (int j = 0; j < x_length; j++) {
    OUT[j] += temp[j + x_length*i];
  }
  OUT[i] += B[i];
}
```

Lift: rewriting

```
Zip(
  W,
  B) >>
Map((neuronW, neuronB) =>
  VVMul(neuronW, toChip(X)) >>
  Reduce(ScalarAdd, 0) >>
  ScalarAdd(neuronB)) >>
VVRelu() >> toOutput()
```

<Map fission>

```
for (int i = 0; i < n_neurons; i++) {
  for (int j = 0; j < x_length; j++) {
    temp[j + x_length*i] =
      X[j] * W[j + x_length*i];
  }

  OUT[i] = 0;
  for (int j = 0; j < x_length; j++) {
    OUT[j] += temp[j + x_length*i];
  }
  OUT[i] += B[i];
}
```

Lift: rewriting

```
Zip(
  W,
  B) >>
Map((neuronW, neuronB) =>
  VVMul(neuronW, toChip(X)) >>
  Reduce(ScalarAdd, 0) >>
  ScalarAdd(neuronB)) >>
VVRelu() >> toOutput()
```

```
Zip(
  W >> Map(neuronW =>
    VVMul(neuronW, toChip(X))),
  B) >>
Map((neuronWX, neuronB) =>
  neuronWX >>
  Reduce(ScalarAdd, 0) >>
  ScalarAdd(neuronB)) >>
VVRelu() >> toOutput()
```

```
for (int i = 0; i < n_neurons; i++) {
  for (int j = 0; j < x_length; j++) {
    temp[j + x_length*i] =
      X[j] * W[j + x_length*i];
  }

  OUT[i] = 0;
  for (int j = 0; j < x_length; j++) {
    OUT[j] += temp[j + x_length*i];
  }
  OUT[i] += B[i];
}
```

```
for (int i = 0; i < n_neurons; i++) {
  for (int j = 0; j < x_length; j++) {
    temp[j + x_length*i] =
      X[j] * W[j + x_length*i];
  }
}
for (int i = 0; i < n_neurons; i++) {
  OUT[i] = 0;
  for (int j = 0; j < x_length; j++) {
    OUT[j] += temp[j + x_length*i];
  }
  OUT[i] += B[i];
}
```

Lift: rewriting

```
Zip(
  W >> Map(neuronW =>
    VVMul(neuronW, toChip(X))),
  B) >>
Map((neuronWX, neuronB) =>
  neuronWX >>
  Reduce(ScalarAdd, 0) >>
  ScalarAdd(neuronB)) >>
VVRelu() >> toOutput()
```

```
for (int i = 0; i < n_neurons; i++) {
  for (int j = 0; j < x_length; j++) {
    temp[j + x_length*i] =
      X[j] * W[j + x_length*i];
  }
}
for (int i = 0; i < n_neurons; i++) {
  OUT[i] = 0;
  for (int j = 0; j < x_length; j++) {
    OUT[j] += temp[j + x_length*i];
  }
  OUT[i] += B[i];
}
```

Lift: rewriting

```
Zip(
  W >> Map(neuronW =>
    VVMul(neuronW, toChip(X))),
  B) >>
Map((neuronWX, neuronB) =>
  neuronWX >>
  Reduce(ScalarAdd, 0) >>
  ScalarAdd(neuronB)) >>
VVRelu() >> toOutput()
```

<A couple rewrites later...>

```
for (int i = 0; i < n_neurons; i++) {
  for (int j = 0; j < x_length; j++) {
    temp[j + x_length*i] =
      X[j] * W[j + x_length*i];
  }
}
for (int i = 0; i < n_neurons; i++) {
  OUT[i] = 0;
  for (int j = 0; j < x_length; j++) {
    OUT[j] += temp[j + x_length*i];
  }
  OUT[i] += B[i];
}
```

Lift: rewriting

```
Zip(
  W >> Map(neuronW =>
    VVMul(neuronW, toChip(X))),
  B) >>
Map((neuronWX, neuronB) =>
  neuronWX >>
  Reduce(ScalarAdd, 0) >>
  ScalarAdd(neuronB)) >>
VVRelu() >> toOutput()
```

→

```
Zip(
  W >> Map(neuronW =>
    VVMul(neuronW, toChip(X)) >>
    Reduce(ScalarAdd, 0)),
  B) >>
Map((neuronWXreduced, neuronB) =>
  neuronWXreduced >>
  ScalarAdd(neuronB)) >>
VVRelu() >> toOutput()
```

```
for (int i = 0; i < n_neurons; i++) {
  for (int j = 0; j < x_length; j++) {
    temp[j + x_length*i] =
      X[j] * W[j + x_length*i];
  }
}
for (int i = 0; i < n_neurons; i++) {
  OUT[i] = 0;
  for (int j = 0; j < x_length; j++) {
    OUT[j] += temp[j + x_length*i];
  }
  OUT[i] += B[i];
}
```

```
for (int i = 0; i < n_neurons; i++) {
  for (int j = 0; j < x_length; j++) {
    temp[j + x_length*i] =
      X[j] * W[j + x_length*i];
  }
  OUT[i] = 0;
  for (int j = 0; j < x_length; j++) {
    OUT[j] += temp[j + x_length*i];
  }
}
for (int i = 0; i < n_neurons; i++) {
  OUT[i] += B[i];
}
```

Lift: rewriting

```
Zip(
  W >> Map(neuronW =>
    VVMul(neuronW, toChip(X)) >>
    Reduce(ScalarAdd, 0)),
  B) >>
Map((neuronWXreduced, neuronB) =>
  neuronWXreduced >>
  ScalarAdd(neuronB)) >>
VVRelu() >> toOutput()
```

```
for (int i = 0; i < n_neurons; i++) {
  for (int j = 0; j < x_length; j++) {
    temp[j + x_length*i] =
      X[j] * W[j + x_length*i];
  }
  OUT[i] = 0;
  for (int j = 0; j < x_length; j++) {
    OUT[j] += temp[j + x_length*i];
  }
}
for (int i = 0; i < n_neurons; i++) {
  OUT[i] += B[i];
}
```

Lift: rewriting

```
Zip(
  W >> Map(neuronW =>
    VVMul(neuronW, toChip(X)) >>
    Reduce(ScalarAdd, 0)),
  B) >>
Map((neuronWXreduced, neuronB) =>
  neuronWXreduced >>
  ScalarAdd(neuronB)) >>
VVRelu() >> toOutput()
```

```
// GEMV rewrite
matrix >> Map(row =>
  VVMul(row, vector) >>
  Reduce(ScalarAdd, 0))
      ↧   ↧   ↧
MVMul(matrix, vector)
```

```
for (int i = 0; i < n_neurons; i++) {
  for (int j = 0; j < x_length; j++) {
    temp[j + x_length*i] =
      X[j] * W[j + x_length*i];
  }
  OUT[i] = 0;
  for (int j = 0; j < x_length; j++) {
    OUT[j] += temp[j + x_length*i];
  }
}
for (int i = 0; i < n_neurons; i++) {
  OUT[i] += B[i];
}
```

Lift: rewriting

```
Zip(
  W >> Map(neuronW =>
    VVMul(neuronW, toChip(X)) >>
    Reduce(ScalarAdd, 0)),
  B) >>
Map((neuronWXreduced, neuronB) =>
  neuronWXreduced >>
  ScalarAdd(neuronB)) >>
VVRelu() >> toOutput()
```

```
Zip(
  MVMul(W, X >> toChip)
  B) >>
Map((neuronWXreduced, neuronB) =>
  neuronWXreduced >>
  ScalarAdd(neuronB)) >>
VVRelu() >> toOutput()
```

```
for (int i = 0; i < n_neurons; i++) {
  for (int j = 0; j < x_length; j++) {
    temp[j + x_length*i] =
      X[j] * W[j + x_length*i];
  }
  OUT[i] = 0;
  for (int j = 0; j < x_length; j++) {
    OUT[j] += temp[j + x_length*i];
  }
}
for (int i = 0; i < n_neurons; i++) {
  OUT[i] += B[i];
}
```

```
temp = MVMul(W, X);
for (int i = 0; i < n_neurons; i++) {
  OUT[i] += B[i];
}
```

Lift: rewriting

```
Zip(
  MVMul(W, X >> toChip)
  B) >>
Map((neuronWXreduced, neuronB) =>
  neuronWXreduced >>
  ScalarAdd(neuronB)) >>
VVRelu() >> toOutput()
```

```
temp = MVMul(W, X);
for (int i = 0; i < n_neurons; i++) {
  OUT[i] += B[i];
}
```

Lift: rewriting

```
Zip(
  MVMul(W, X >> toChip)
  B) >>
Map((neuronWXreduced, neuronB) =>
  neuronWXreduced >>
  ScalarAdd(neuronB)) >>
VVRelu() >> toOutput()
```

```
< Vectorise Map-Zip >
```

```
temp = MVMul(W, X);
for (int i = 0; i < n_neurons; i++) {
  OUT[i] += B[i];
}
```

Lift: rewriting

```
Zip(
  MVMul(W, X >> toChip)
  B) >>
Map((neuronWXreduced, neuronB) =>
  neuronWXreduced >>
  ScalarAdd(neuronB)) >>
VVRelu() >> toOutput()
```

```
VVAdd(
  MVMul(W, X >> toChip)
  B) >>
VVRelu() >> toOutput()
```

```
temp = MVMul(W, X);
for (int i = 0; i < n_neurons; i++) {
  OUT[i] += B[i];
}
```

```
temp = MVMul(W, X);
OUT = VVAdd(temp, B);
```

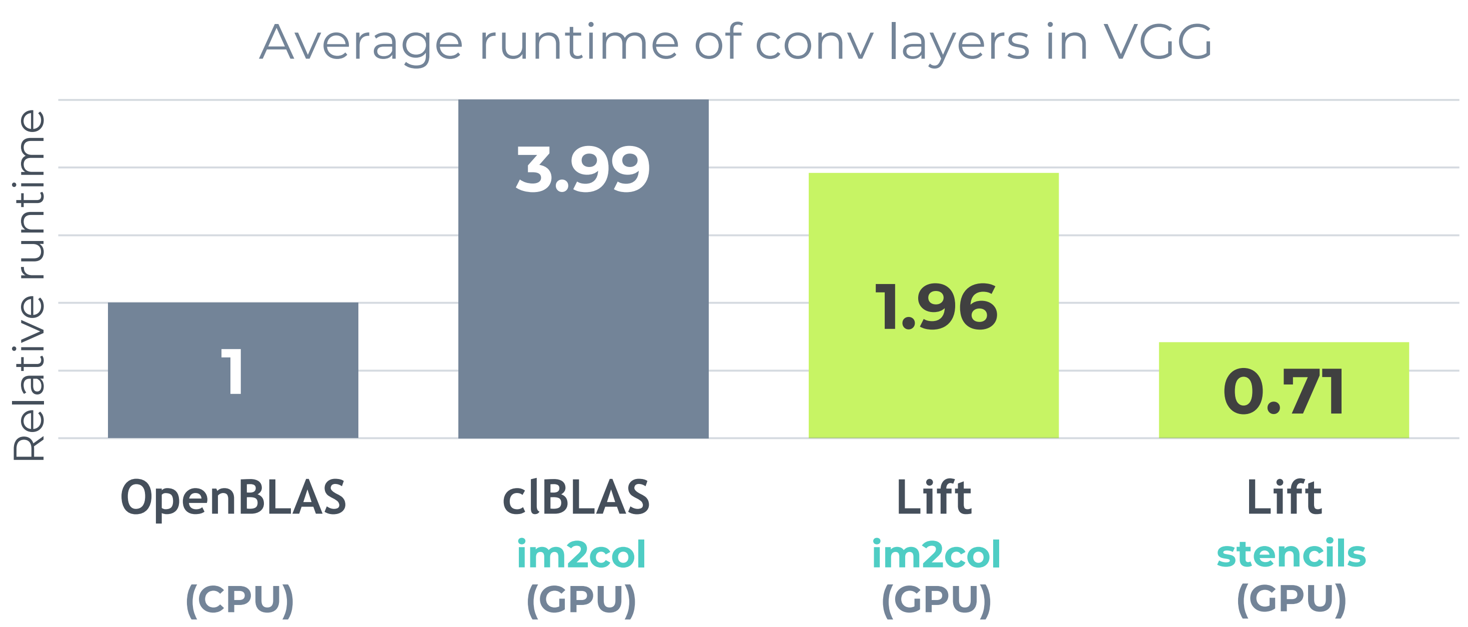
Lift: rewriting

```
VVAdd(
  MVMul(W, X >> toChip)
  B) >>
VVRelu() >> toOutput()
```

```
temp = MVMul(W, X);
OUT = VVAdd(temp, B);
```

Preliminary results

- Functional correctness on the BrainWave accelerator
- Performance measurements on Mali GPU



Future work

- Generation of both OpenCL kernel and host runtime
- Performance evaluation on popular DNN architectures
- Support for more hardware accelerators