THE UNIVERSITY *of* EDINBURGH
informatics

EPSRC Centre for Doctoral Training in
Pervasive Parallelism

www.lift-project.org

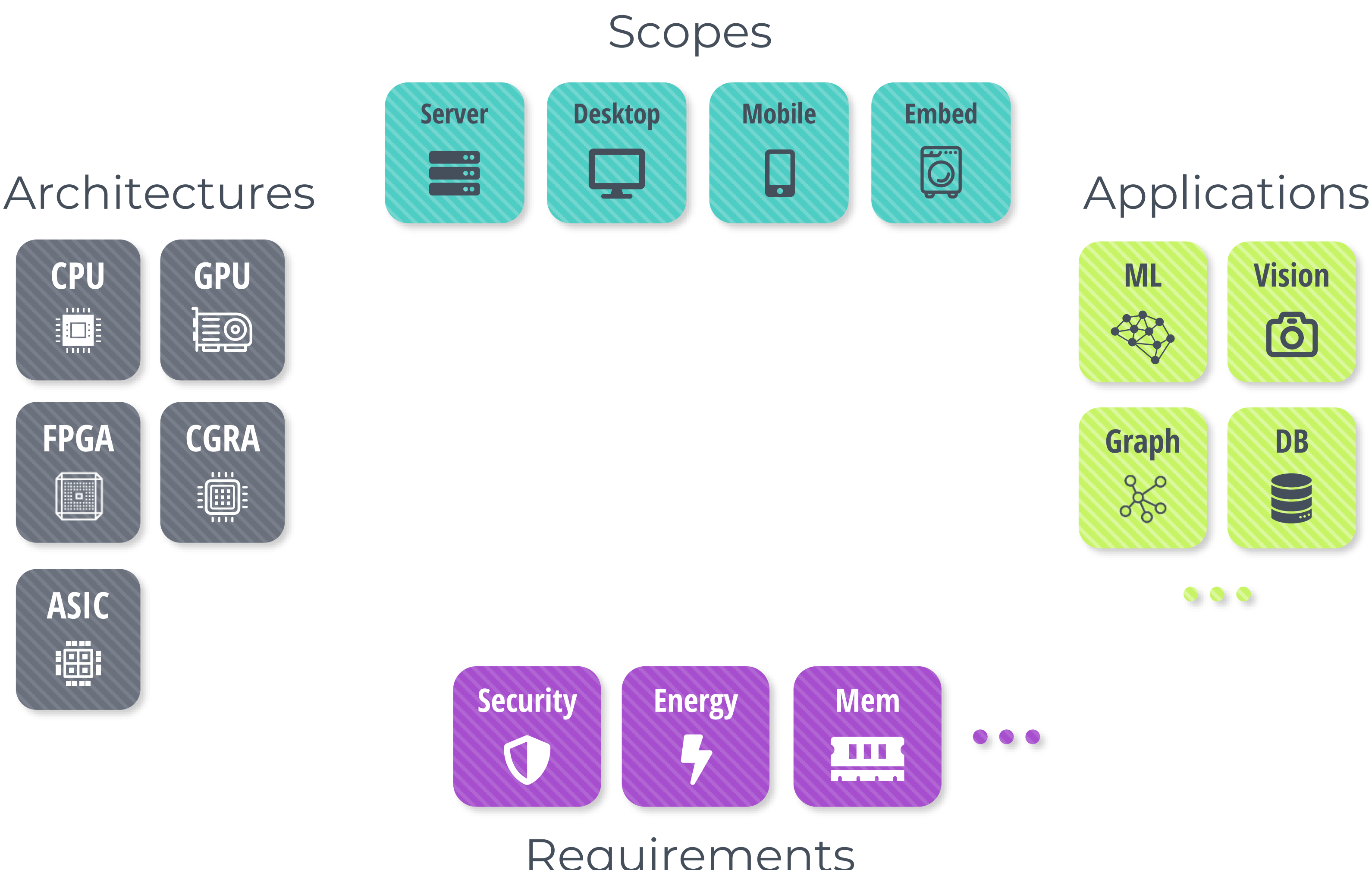
Functional Interface for Performance Portability on Parallel Accelerators

Naums Mogers

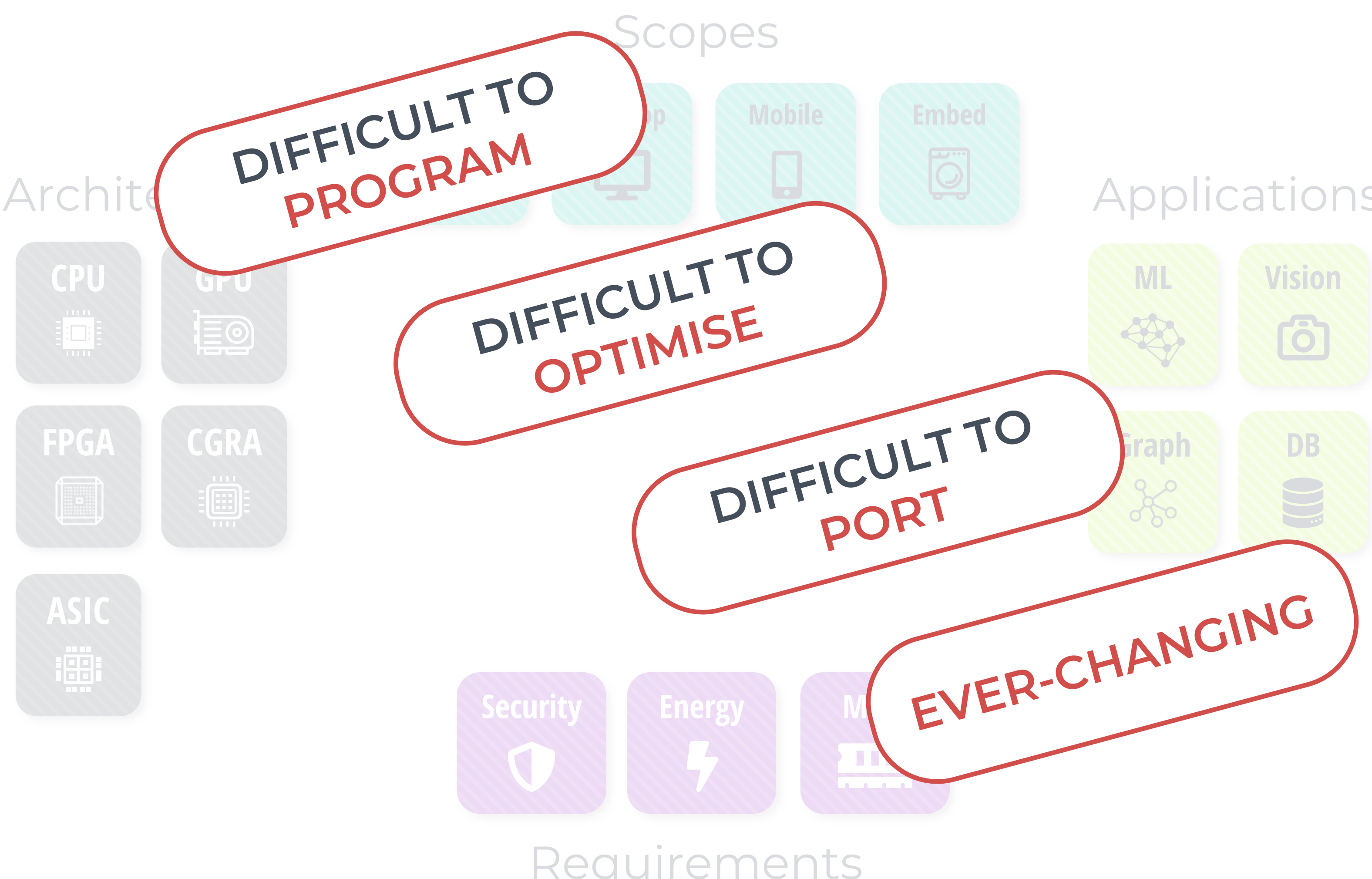
Christophe Dubach

ARM Research Summit, September 2019

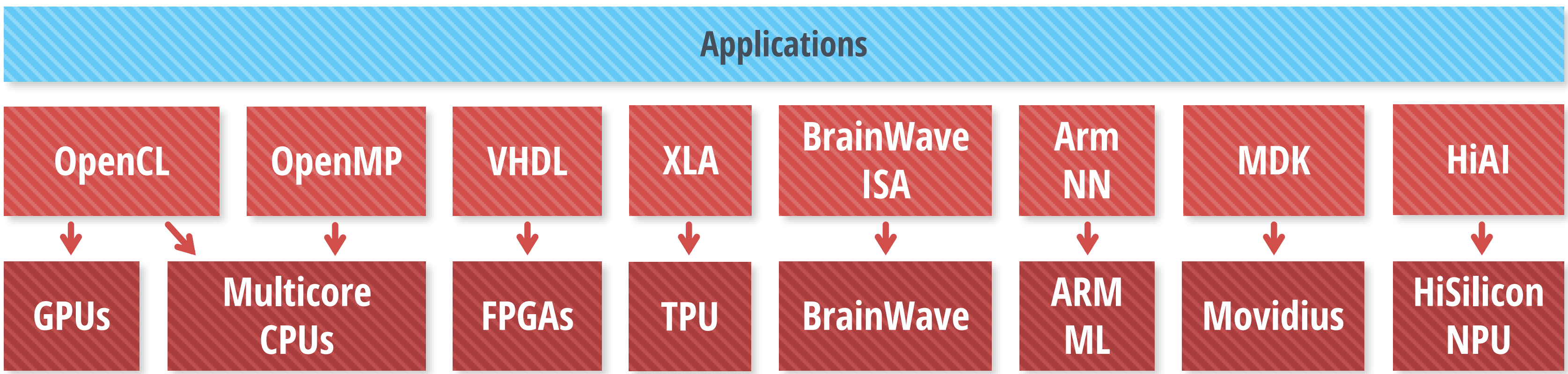
Hardware accelerators



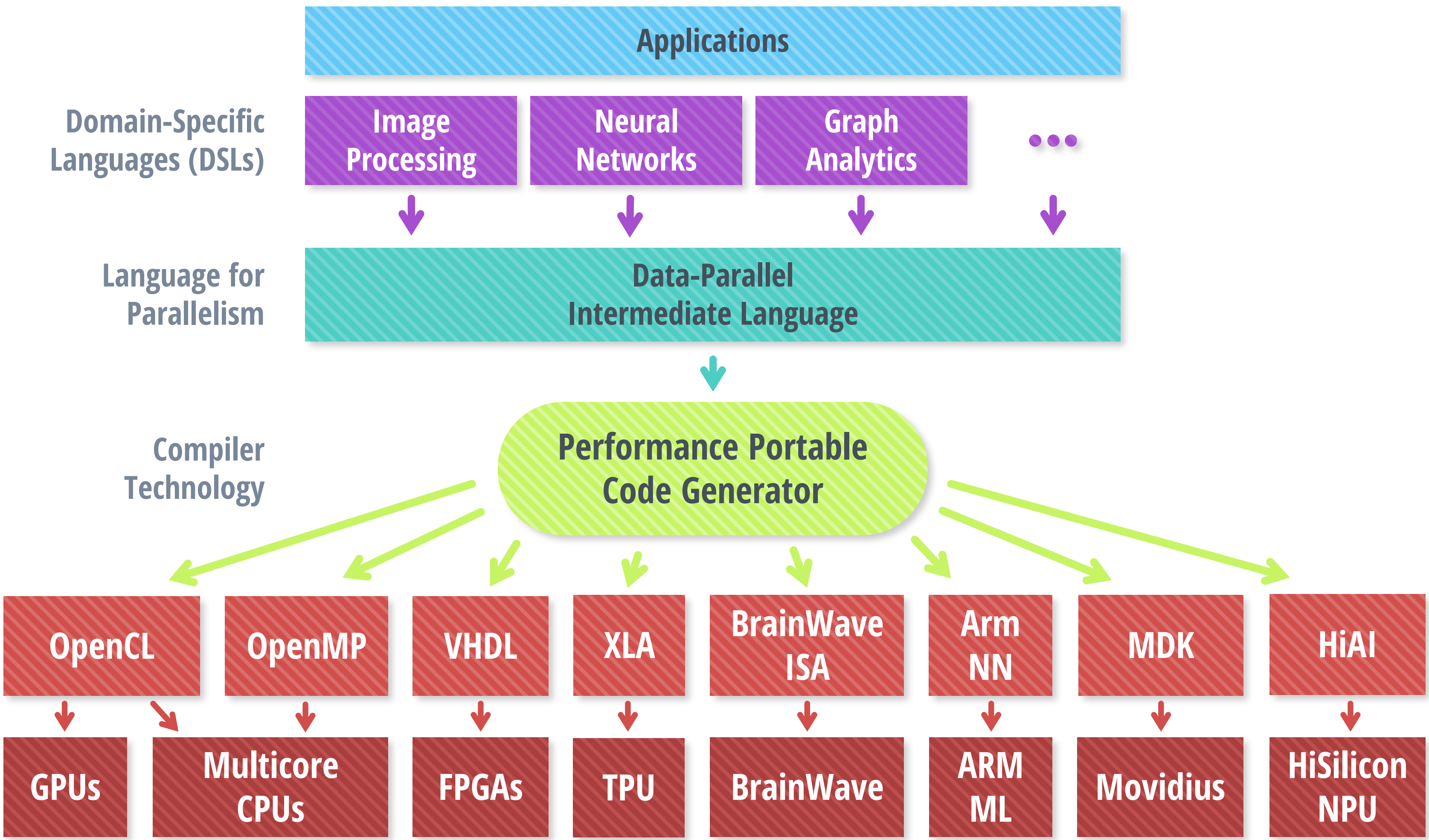
Hardware accelerators



Current landscape

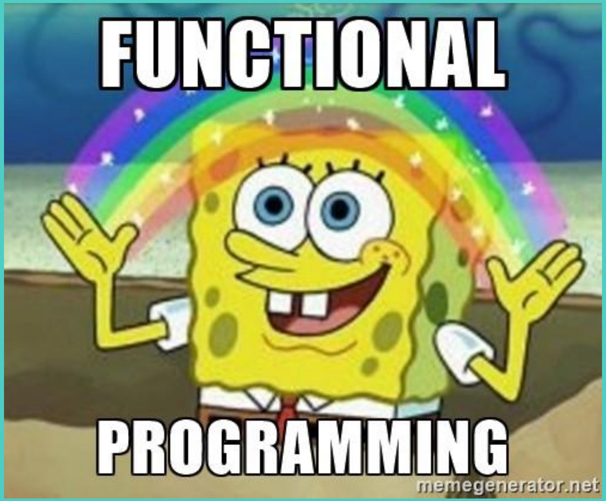


What we need


Applications
Domain-Specific Languages (DSLs)
Image Processing
Neural Networks
Graph Analytics
...
Language for Parallelism
Data-Parallel Intermediate Language
Compiler Technology
Performance Portable Code Generator
OpenCL
OpenMP
VHDL
XLA
BrainWave ISA
Arm NN
MDK
HiAI
GPUs
Multicore CPUs
FPGAs
TPU
BrainWave
ARM ML
Movidius
HiSilicon NPU

What is the right interface for HW accelerators?

What is the right interface for HW accelerators?



Functional approach

Abstract

- Expresses algorithm (WHAT), not implementation (HOW)

High-level

- Captures plenty of algorithmic meta-info for analysis

Pure

- Easy to transform

Safe

- Easier to use and parallelise

Expressive

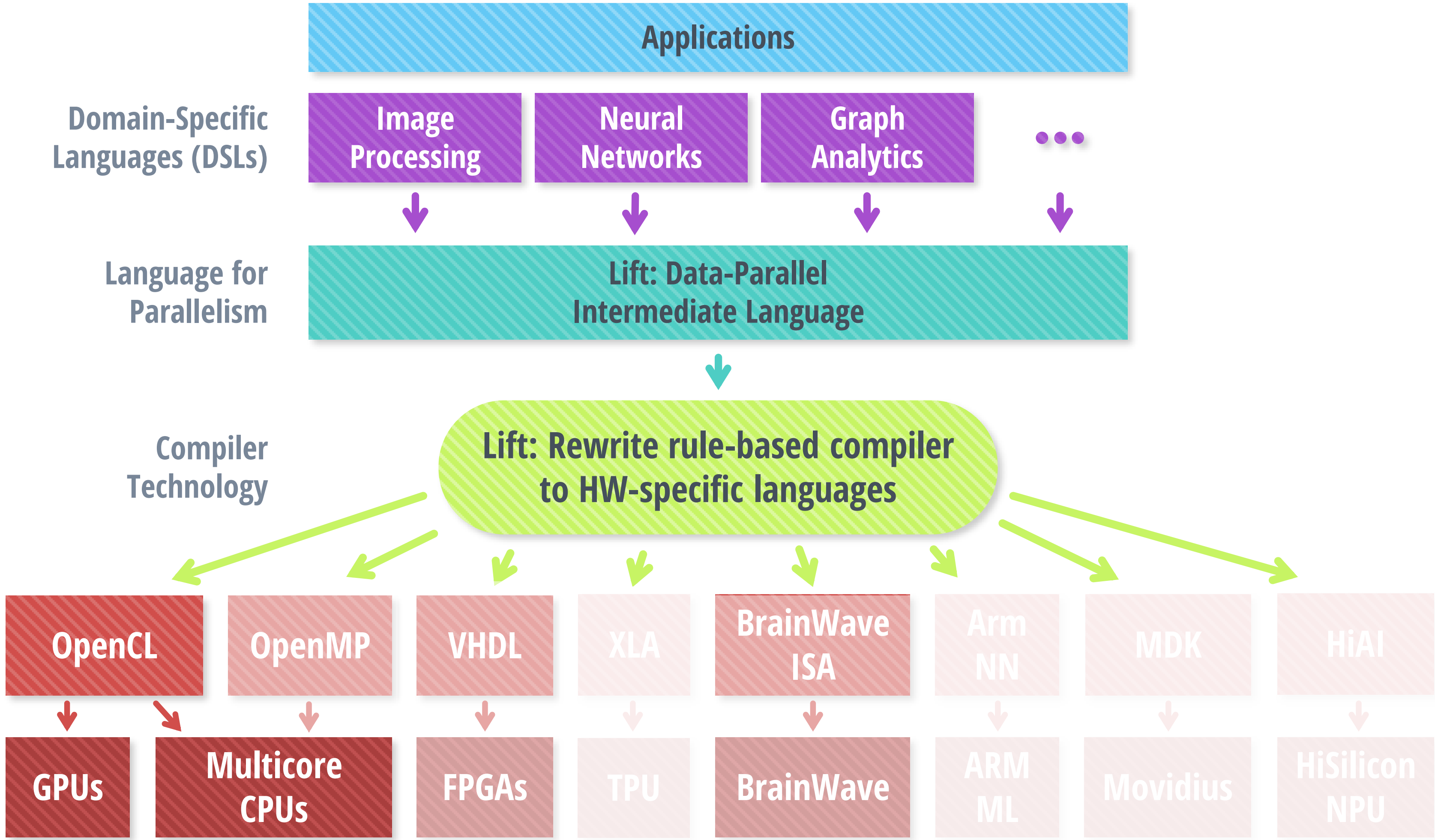
- Control flow
- Memory management

Composable

- Easier to maintain, code-reuse

```
gemv(mat, x, y, α, β) =
  map(+, zip(
    map(λ row ↦ scal(α, dotProduct(row, x)), mat),
    scal(β, y) ) )
```

Lift


Applications
Domain-Specific Languages (DSLs)
Image Processing
Neural Networks
Graph Analytics
...
Language for Parallelism
Lift: Data-Parallel Intermediate Language
Compiler Technology
Lift: Rewrite rule-based compiler to HW-specific languages
OpenCL
OpenMP
VHDL
XLA
BrainWave ISA
Arm NN
MDK
HiAI
GPUs
Multicore CPUs
FPGAs
TPU
BrainWave
ARM ML
Movidius
HiSilicon NPU

Lift

www.lift-project.org

- Functional data-parallel language and compiler

```
A >> map(λ(rowA) =>
  B >> map(λ(colB) =>
    zip(rowA, colB) >>
    map(*) >> reduce(0, +)))
```

→

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

Lift IR

www.lift-project.org

Algorithmic patterns	Data types	Address space operators	Casters	Data operators
`Map, Reduce`	`Int, Float`	`toGlobal`	`toVector`	`add, mul`
`Zip, Split`	`Vector`	`toLocal`	`toScalar`	`dot`
`Scatter, Gather`	`Array`	`toPrivate`		`tanh`
`Slide`				

Lift IR: Views

www.lift-project.org

```
Reorder(stride(s)) >> Map(f)
```

NO WRITES TO MEMORY

TRANSFORMED READS

- Virtual composable data layout transformations
  - `Reorder, Transpose, Slide, Slice, etc`
- Expressed with Views
- Help avoid extra memory writes

Lift IR

www.lift-project.org

Algorithmic patterns	Data types	Address space operators	Casters	Data operators
`Map, Reduce`	`Int, Float`	`toGlobal`	`toVector`	`add, mul`
`Zip, Split`	`Vector`	`toLocal`	`toScalar`	`dot`
`Scatter, Gather`	`Array`	`toPrivate`		`tanh`
`Slide`				

Lift IR

www.lift-project.org

IR level	Algorithmic patterns	Data types	Address space operators	Casters	Data operators

DSL	`conv, lstm`
	`blur, sharpen`
	`select`
		`Vector`
		`Matrix`
		`Tensor`			

Generic	`Map, Reduce`
	`Zip, Split`
	`Scatter, Gather`
	`Slide`
		`Int, Float`
		`Vector`
		`Array`
			`toGlobal`
			`toLocal`
			`toPrivate`
				`toVector`
				`toScalar`
					`add, mul`
					`dot`
					`tanh`

Platform-specific	`MapGlobal`
	`MapLocal`
	`ReduceSeq`
		`Int8`
		`Float16`
		`Float32`
			`toDRAM`
			`toSRAM`
			`toRegistor`
				`toInt8`
				`toFloat16`
					`VVMul`
					`MVMul`
					`VTanh`

How do we achieve performance portability?

Lift: 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)
```

- Express algorithmic implementation choices
- Preserve semantic correctness
- Leverage algorithmic info

- Decouples optimisation from code generation

Lift: Rewrite Rules

IR level	Rewrite rules	

DSL	▣ Algorithm choices for high-level primitives
	▣ Precision level
	▣ ...
		`conv(..) ↦ stencilConv(..)`
		`conv(..) ↦ gemmConv(..)`
		`conv(..) ↦ winogradConv(..)`

Generic	▣ Split-join rule
	▣ Map fusion rule
	▣ Reduce rules
	▣ ...
		`reducePart(z,f) ↦ reorder(..) >> reducePart(z,f)`
		`reducePart(z,f) ↦ split(n) >> map(reducePart(z,f)) >> join()`
		`reducePart(z,f) ↦ iterate(reducePart(z,f))`
		`reducePart(z,f) ↦ reduceSeq(z,f)`

Platform-specific	▣ Using built-ins
	▣ Lowering to the platform programming model
	▣ ...
		`map(*) >> reduce(0, +) ↦ dot()`
		`map(map(f)) ↦ mapWrg(mapLcl(f))`
		`map(f) ↦ asVector() >> map(vectorise(f)) >> asScalar()`

Lift: rewriting

HOW TO
OPTIMISE?

```
for (int n = 0; n < N; n++) {
    out[n] = B[n];
    for (int d = 0; d < D; d++) {
        out[n] += X[d] * W[d + D*n];
    }
}
```

Lift: rewriting

```
layer(W: float[N][D], B: float[N],
      X: float[D]):    float[N] =
  zip(W, B) >> map(λ(Wn, Bn) =>
    zip(Wn, X) >> map(*) >> reduce(Bn, +))
```

HARD STARTING POINT

```
for (int n = 0; n < N; n++) {
    out[n] = B[n];
    for (int d = 0; d < D; d++) {
        out[n] += X[d] * W[d + D*n];
    }
}
```

Lift: rewriting

Search

```
layer(W: float[N][D], B: float[N],
      X: float[D]):    float[N] =
  zip(W, B) >> map(λ(Wn, Bn) =>
    zip(Wn, X) >> map(*) >> reduce(Bn, +))
```

```
layer(W: float[N][D], B: float[N],
      X: float[D]):    float[N] =
  zip(W, B) >> map(λ(Wn, Bn) =>
    zip(Wn, X) >>
    concat(
        slice(0, (D/64)*64) >> split(64) >>
        map(map(*) >> reduce(0, +)) >>
        reduce(0, +),

        slice((D/64)*64, D) >>
        map(*) >> reduce(0, +)) >>

      reduce(Bn, +))
```

```
for (int n = 0; n < N; n++) {
    out[n] = B[n];
    for (int d = 0; d < D; d++) {
        out[n] += X[d] * W[d + D*n];
    }
}
```

Lift: rewriting

```
layer(W: float[N][D], B: float[N],
      X: float[D]):    float[N] =
  zip(W, B) >> map(λ(Wn, Bn) =>
    zip(Wn, X) >> map(*) >> reduce(Bn, +))
```

Search

```
layer(W: float[N][D], B: float[N],
      X: float[D]):    float[N] =
  zip(W, B) >> map(λ(Wn, Bn) =>
    zip(Wn, X) >>
    concat(
        slice(0, (D/64)*64) >> split(64) >>
        map(map(*) >> reduce(0, +)) >>
        reduce(0, +),

        slice((D/64)*64, D) >>
        map(*) >> reduce(0, +)) >>

    reduce(Bn, +))
```

Built-in primitive

```
map(*) >> reduce(0, +)
  ↧      ↧       ↧
 dot_product_accel()
```

Exploitation

```
layer(W: float[N][D], B: float[N],
      X: float[D]):    float[N] =
  zip(W, B) >> mapWrg(λ(Wn, Bn) =>
    zip(Wn, X) >>
    concat(
        slice(0, (D/64)*64) >> split(64) >>
        mapLcl(dot_product_accel()) >>
        reduce(0, +),

        slice((D/64)*64, D) >>
        mapSeq(*) >> reduce(0, +)) >>

    reduce(Bn, +))
```

```
for (int n = 0; n < N; n++) {
    out[n] = B[n];
    for (int d = 0; d < D; d++) {
        out[n] += X[d] * W[d + D*n];
    }
}
```

Lift: rewriting

```
layer(W: float[N][D], B: float[N],
      X: float[D]):    float[N] =
  zip(W, B) >> map(λ(Wn, Bn) =>
    zip(Wn, X) >> map(*) >> reduce(Bn, +))
```

Search

```
layer(W: float[N][D], B: float[N],
      X: float[D]):    float[N] =
  zip(W, B) >> map(λ(Wn, Bn) =>
    zip(Wn, X) >>
    concat(
        slice(0, (D/64)*64) >> split(64) >>
        map(map(*) >> reduce(0, +)) >>
        reduce(0, +),

        slice((D/64)*64, D) >>
        map(*) >> reduce(0, +)) >>

    reduce(Bn, +))
```

Built-in primitive

```
map(*) >> reduce(0, +)
   ↧      ↧      ↧
 dot_product_accel()
```

Parallelisation choice

```
    map(.. >> map(..) >> ..)
           ↧     ↧     ↧
mapWrg(.. >> mapLcl(..) >> ..)
```

Exploitation

```
layer(W: float[N][D], B: float[N],
      X: float[D]):    float[N] =
  zip(W, B) >> mapWrg(λ(Wn, Bn) =>
    zip(Wn, X) >>
    concat(
        slice(0, (D/64)*64) >> split(64) >>
        mapLcl(dot_product_accel()) >>
        reduce(0, +),

        slice((D/64)*64, D) >>
        mapSeq(*) >> reduce(0, +)) >>

    reduce(Bn, +))
```

```
for (int n = 0; n < N; n++) {
  out[n] = B[n];
  for (int d = 0; d < D; d++) {
      out[n] += X[d] * W[d + D*n];
  }
}
```

Lift: rewriting

```
layer(W: float[N][D], B: float[N],
      X: float[D]):    float[N] =
  zip(W, B) >> map(λ(Wn, Bn) =>
    zip(Wn, X) >> map(*) >> reduce(Bn, +))
```

Search

```
layer(W: float[N][D], B: float[N],
      X: float[D]):    float[N] =
  zip(W, B) >> map(λ(Wn, Bn) =>
    zip(Wn, X) >>
    concat(
        slice(0, (D/64)*64) >> split(64) >>
        map(map(*) >> reduce(0, +)) >>
        reduce(0, +),

        slice((D/64)*64, D) >>
        map(*) >> reduce(0, +)) >>

    reduce(Bn, +))
```

Built-in primitive

```
map(*) >> reduce(0, +)
   ↧       ↧      ↧
 dot_product_accel()
```

Parallelisation choice

```
   map(.. >> map(..) >> ..)
           ↧      ↧      ↧
mapWrg(.. >> mapLcl(..) >> ..)
```

Exploitation

```
layer(W: float[N][D], B: float[N],
      X: float[D]):    float[N] =
  zip(W, B) >> mapWrg(λ(Wn, Bn) =>
    zip(Wn, X) >>
    concat(
        slice(0, (D/64)*64) >> split(64) >>
        mapLcl(dot_product_accel()) >>
        reduce(0, +),

        slice((D/64)*64, D) >>
        mapSeq(*) >> reduce(0, +)) >>

    reduce(Bn, +))
```

```
for (int n = get_group_id(0); n < N; n += get_num_groups(0)) {
    out[n] = B[n];
    for (int t = get_local_id(0); t < (D/64); t += get_local_size(0)) {
        out[n] += dot_product_accel(W + (t*64 + n*D), X + (t*64));
    }
    barrier(CLK_LOCAL_MEM_FENCE);

    if (get_local_id(0) < 1) {
        for (int d = D/64*64; d < D; d += 1) {
            out[n] += W[d + n*D] * X[d];
        }
    }
}
```

Code generation

Lift: Rewrite Rules

- Domain-specific and generic
- Reusable
- Provably correct
- Self-contained, extensible

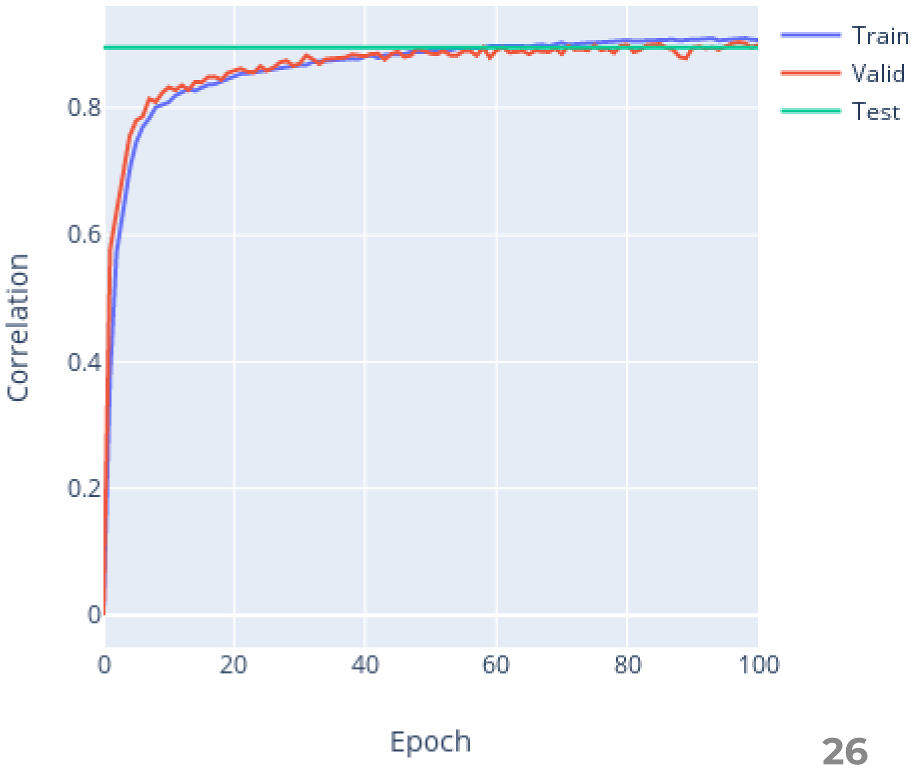
Lift: Constraint Inference

- Required for valid search space generation when using tuning parameters
- Leverages algorithmic meta-info
- Can express heuristics and HW restrictions

Split(*s*) \$ $[T]_N$	=>	***N % s == 0***
asVector(*v*) \$ $[T]_N$	=>	***N % v == 0***
Slide(*len*, *step*) o $[T]_N$	=>	***N >= len***
SlideStrict(*len*, *step*) o $[T]_N$	=>	***N >= len &&*** ***N % ((N - (len - step)) / step) == 0***

Lift: Search Space Exploration

- Uniform random sampling
- Predictor models
- Genetic algorithms
- ...



Lift: Research Directions

- Linear algebra
- Sparse data parallelism
- Optimising reductions
- Stencil computations
- 3D wave modelling
- High-level synthesis for FPGAs
- Machine Learning

Lift for Machine Learning

- Machine Learning
  - Convolution inference optimisation
  - Platforms: Mali GPUs, BrainWave

```
def convLayer(kernelsWeights : [[[[float]_inputChannels]_kernelWidth]_kernelHeight]_numKernel,
              kernelsBiases  : [float]_numKernel,
              inputData      : [[[float]_inputChannels]_inputWidth]_inputHeight,
              padSize        : (int,int,int,int),          kernelStride   : (int,int))
              : [[[float]_outWidth]_outHeight]_numKernel = {

  val paddedInput = pad2D(padSize, value = 0 ,inputData)
  val slidingWindows = slide2D(kernelHeight, kernelWidth, kernelStride._1, kernelStride._2, paddedInput)
   map2D(slidingWindow ->
     map((singleKernelWeights, singleKernelBias) ->
       reduce(init = singleKernelBias, f = (acc, (x, w)) -> {acc + x * w},
         zip(join(join(slidingWindow)), join(join(singleKernelWeights)))),
       zip(kernelsWeights, kernelsBiases)),
     slidingWindows)}
```

Lift for Machine Learning

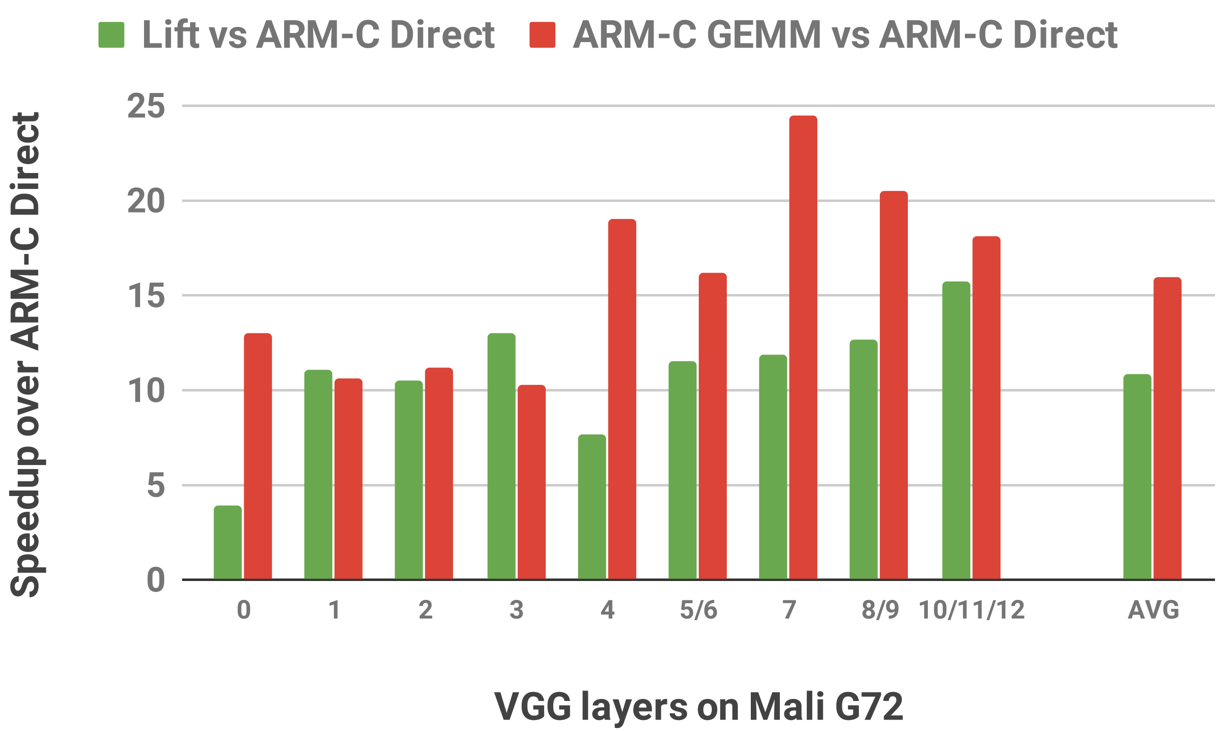
- Machine Learning
  - Convolution inference optimisation
  - Platforms: Mali GPUs, BrainWave

```
def partialConv(kernelsWeights : [[[[float]inputChannels]kernelWidth]kernelHeight]numKernels,
                paddedInput    : [[[float]inputChannels]paddedInputWidth]paddedInputHeight,
                kernelStride   : (int, int))
                : [[[[[float]windowSize/ω]nWindowsInTile]κ]numKernels/κ]nTilesInInput = {

  val tiledInput4D = join(slide2D(θ, tilingStride, paddedInput))

  val tiledSlidedInput5D = map(join(slide2D((kernelHeight, kernelWidth), kernelStride)), tiledInput4D)

  val windowSize = inputChannels * kernelWidth * kernelHeight

  def coalesceChunkVectorizeWindow(window : [[[float]inputChannels]kernelWidth]kernelHeight])
                                   : [[float_v]ω]windowSize/ω = {

    val flatWindow1D = join(join(window))
    val flatCoalescedWindow1D = reorder(striddenIndex(windowSize/ω), flatWindow1D)
    val flatCoalescedChunkedWindow1D = split(ω, flatCoalescedWindow1D)
    asVector(v, flatCoalescedChunkedWindow1D)
  }

  val tiledSlidedCoalescedChunkedVectorizedInput4D = map(tile4D -> map(window3D ->
    coalesceChunkVectorizeWindow(window3D), tile4D), tiledSlidedInput5D)

  val groupedCoalescedChunkedVectorizedKernelsWeights4D = split(κ, map(singleKernelWeights ->
    coalesceChunkVectorizeWindow(singleKernelWeights), kernelsWeights))

  mapWrg(1, inputTile3D ->
    mapWrg(0, kernelsGroupWeights3D -> transpose(
      mapLcl(1, inputWindow2D -> transpose(
        mapLcl(0, (inputWindowChunk1D, kernelsGroupChunk2D) ->
          mapSeq(singleKernelReducedChunk -> toGlobal(singleKernelReducedChunk),
            join(
              reduceSeq(
                init = mapSeq(toPrivate(id(Value(0, [float]κ)))),
                f = (acc, (inputValue, kernelsGroupValue1D)) ->
                  let(inputValuePrivate ->
                    mapSeq((accValue, singleKernelValue) ->
                      accValue + vectorize(v, dot(inputValuePrivate, singleKernelValue)),
                      zip(acc, kernelsGroupValue1D)),
                    toPrivate(vectorize(v, id(inputValue)))),
                zip(inputWindowChunk1D, transpose(kernelsGroupChunk2D))))),
          zip(inputWindow2D, transpose(kernelsGroupWeights3D)))),
        inputTile3D)),
      groupedCoalescedChunkedVectorizedKernelsWeights4D),
    tiledSlidedCoalescedChunkedVectorizedInput4D)
```

```
float4 dotAndSumUp(float acc, float4 l, float4 r){ return acc + dot(l, r); }
void partialConv(const global float* restrict kernels, const global float* restrict input, global float* out){

    int wg_id_1 = get_group_id(1); int wg_id_0 = get_group_id(0);
    int l_id_0 = get_local_id(0);  int l_id_1 = get_local_id(1);

    private float acc_0  = 0.0f;  // start map_seq_unrolled
    private float acc_1  = 0.0f;
    ...
    private float acc_7 = 0.0f; // end map_seq_unrolled

    for (int i = 0; i < 36; i+) { // start reduce_seq
        private float4 inputElem = vload4(((l_id_0+(32*i))%128)/4 + (i ... wg_id_1 ... l_id_0 ...  l_id_1),input)

        // start map_seq_unrolled
        acc_0 = dotAndSumUp(acc_0, inputElem, vload4(0    + l_id_0 % 4 + (l_id_0/4 + (8*i) + (2304*wg_id_0)), kernels);
        acc_1 = dotAndSumUp(acc_1, inputElem, vload4(288  + l_id_0 % 4 + (l_id_0/4 + (8*i) + (2304*wg_id_0)), kernels);
        ...
        acc_7 = dotAndSumUp(acc_7, inputElem, vload4(2304 + l_id_0 % 4 + (l_id_0/4 + (8*i) + (2304*wg_id_0)), kernels);
        // end map_seq_unrolled
        } // end reduce_seq

    // map_seq_unrolled
    out[(0   + l_id_0 + (8 * l_id_1) + (576 * wg_id_0) + (9216 * wg_id_1))] = acc_0;
    out[(72  + l_id_0 + (8 * l_id_1) + (576 * wg_id_0) + (9216 * wg_id_1))] = acc_1;
    out[(144 + l_id_0 + (8 * l_id_1) + (576 * wg_id_0) + (9216 * wg_id_1))] = acc_2;
    ...
    out[(504 + l_id_0 + (8 * l_id_1) + (576 * wg_id_0) + (9216 * wg_id_1))] = acc_8;
}   // end map_seq_unrolled
```

Lift for Machine Learning

- Machine Learning
  - Convolution inference optimisation
  - Platforms: Mali GPUs, BrainWave



Lift for Machine Learning

Naums Mogers, PhD student, Edinburgh

How to best exploit HW accelerators?

Christof Schlaak, PhD student, Edinburgh

How to generate accelerator architectures?

Lu Li, Postdoctoral Researcher, Edinburgh

How to optimise the host code?
How to drive the rewriting process?

Christophe Dubach, Reader, Edinburgh

All of the above

Lift source code is published

https://github.com/lift-project/lift

http://www.lift-project.org

References

(icons) Noun Project, https://thenounproject.com
(icons) Font Awesome, https://fontawesome.com