Lift: a Functional Approach to Generating High Performance GPU Code using Rewrite Rules

Toomas
Remmelg

Michel
Steuwer

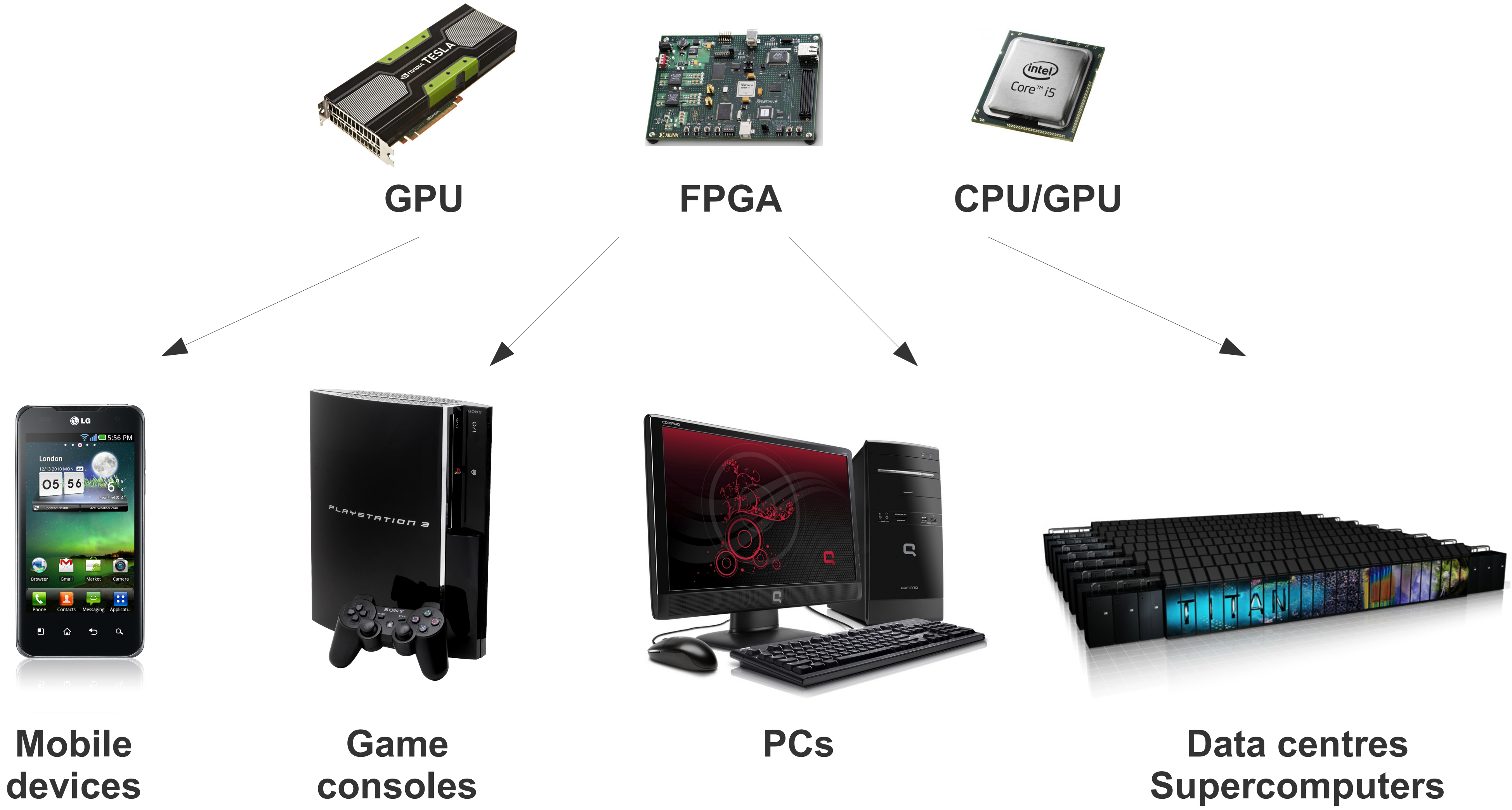
Christophe
Dubach

THE UNIVERSITY *of* EDINBURGH
informatics

The 4th South of England Regional Programming Language Seminar

27th September 2016

Heterogeneity Everywhere



GPU

FPGA

CPU/GPU

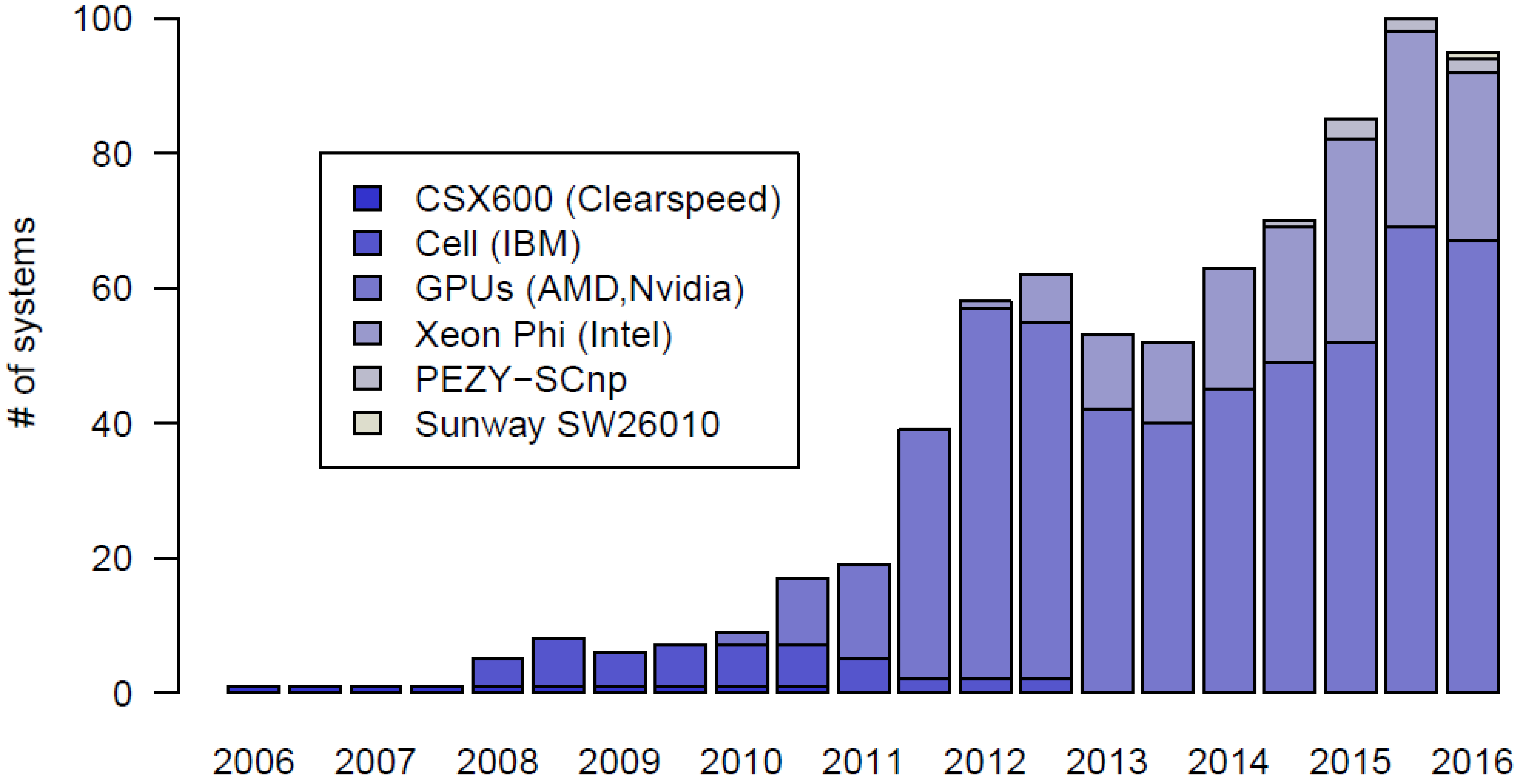
Mobile devices

Game consoles

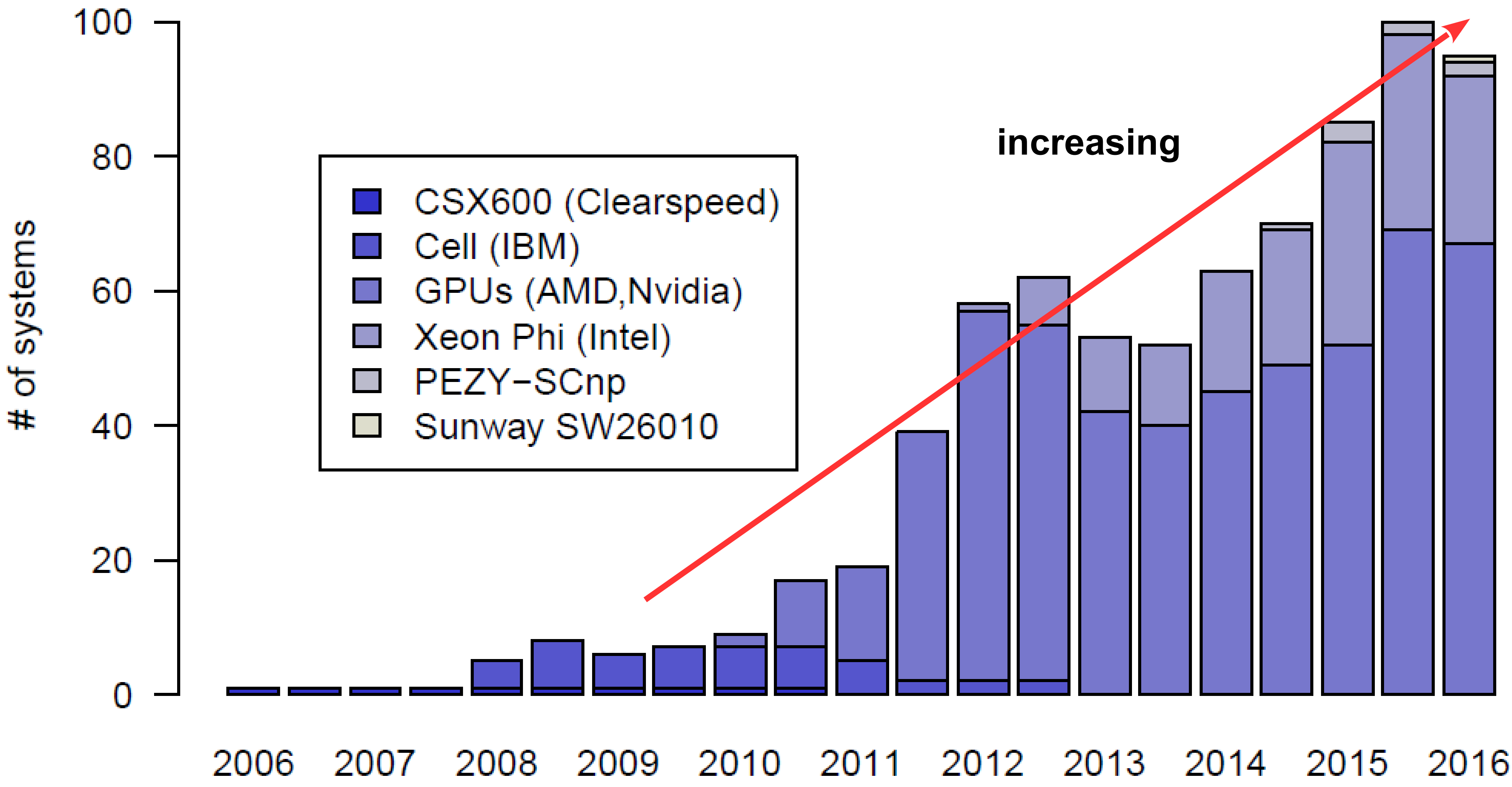
PCs

Data centres
Supercomputers

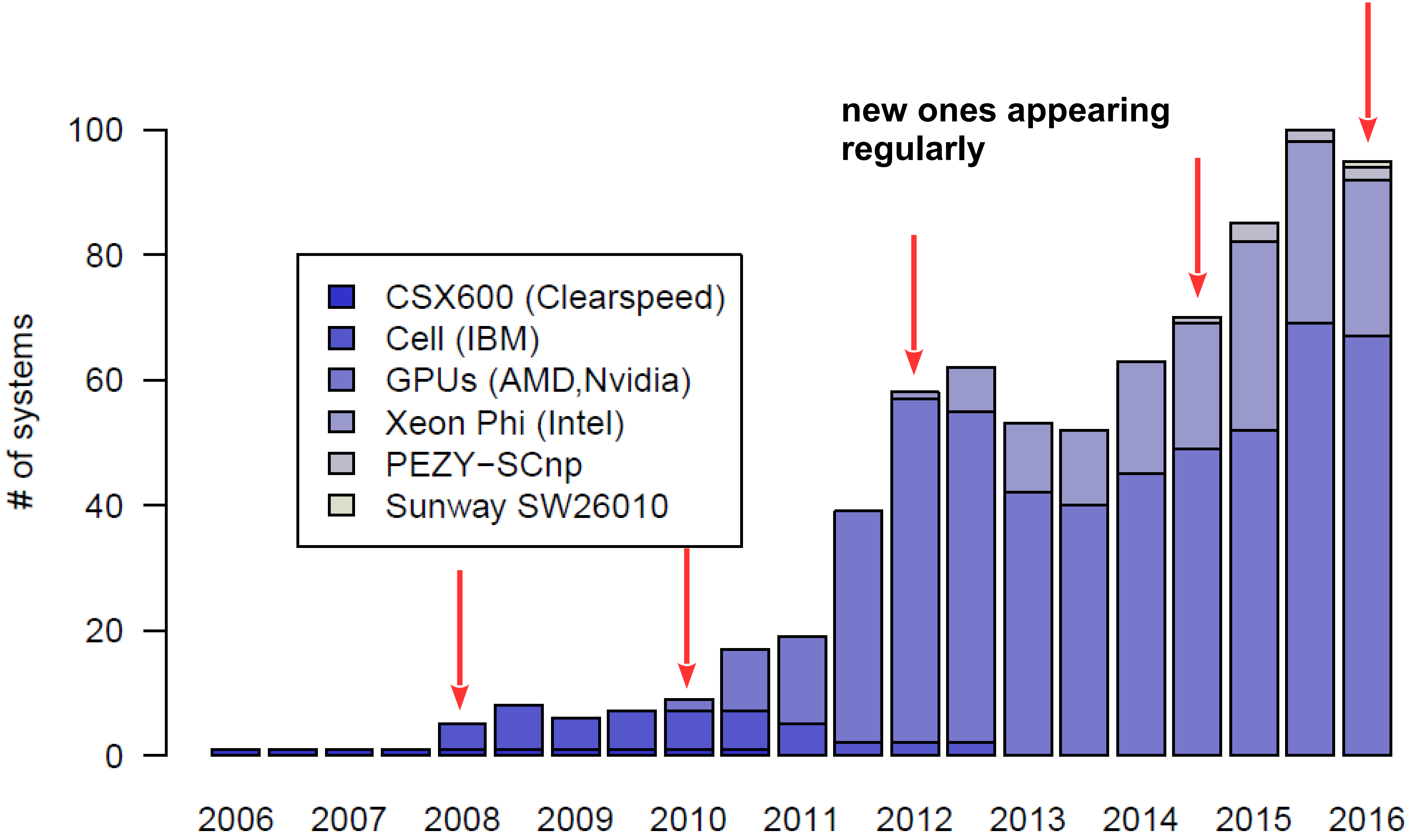
Top 500 with parallel accelerators



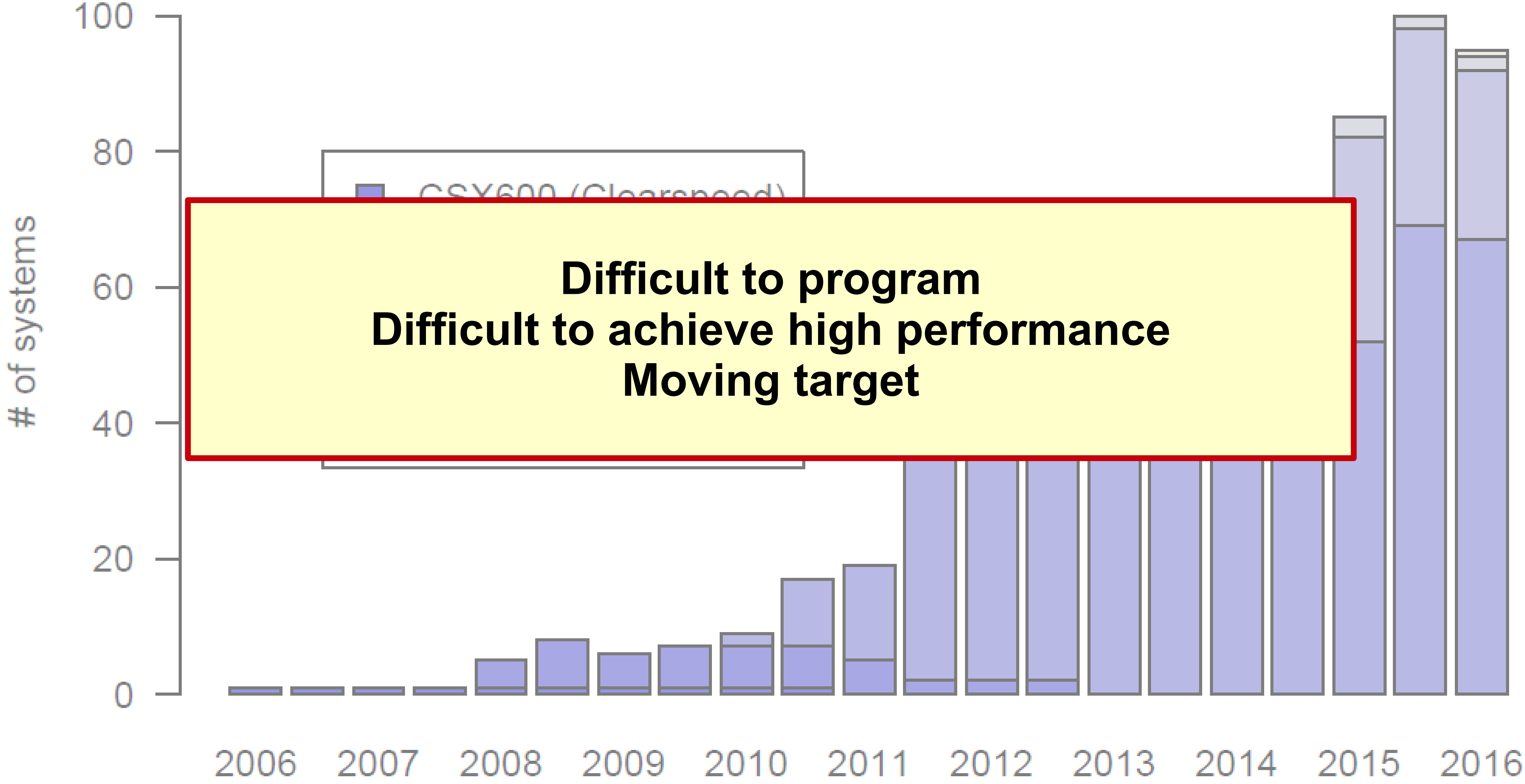
Top 500 with parallel accelerators



Top 500 with parallel accelerators


new ones appearing regularly

Top 500 with parallel accelerators



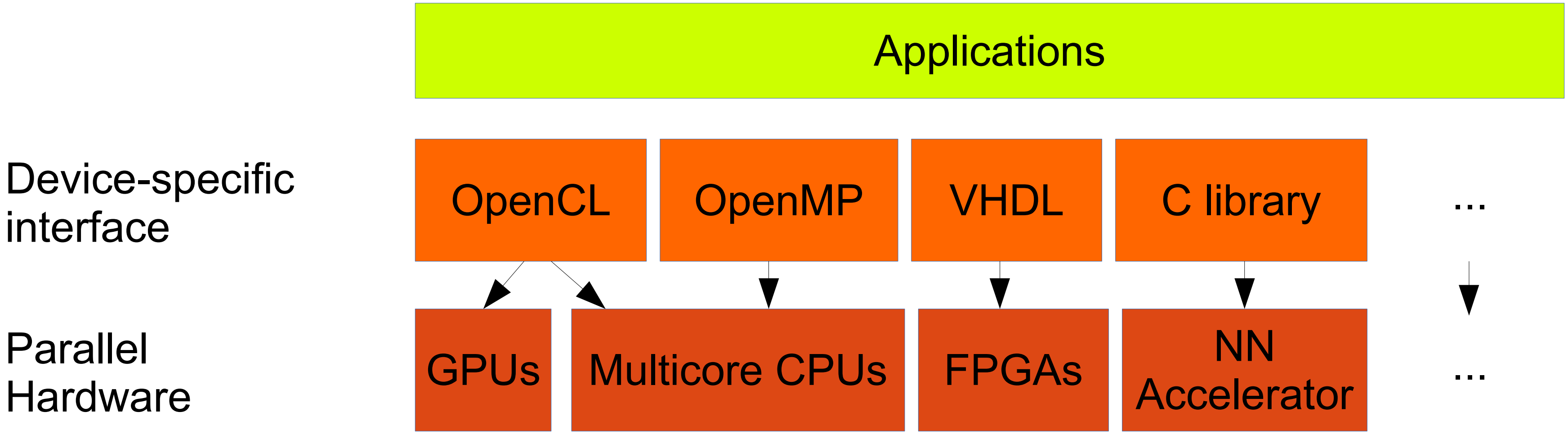
Difficult to program
Difficult to achieve high performance
Moving target

But also domain-specific accelerators

- E.g. Google neural network accelerators
  - TPU (Tensor Processor Units)

- E.g. Movidius vision accelerators
  - VPU (Vision Processing Units)

Current situation



Applications

Device-specific interface

OpenCL

OpenMP

VHDL

C library

...

Parallel Hardware

GPUs

Multicore CPUs

FPGAs

NN Accelerator

...

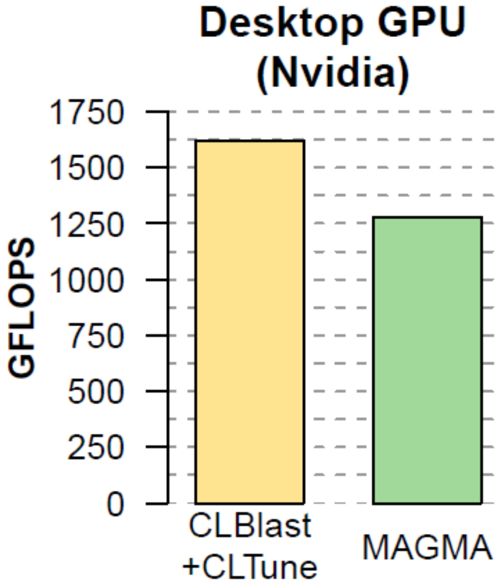
Programmers have to be expert in high performance computing!

+ Performance is not portable!

- hand-written implementations for each device
  - parametric auto-tuner

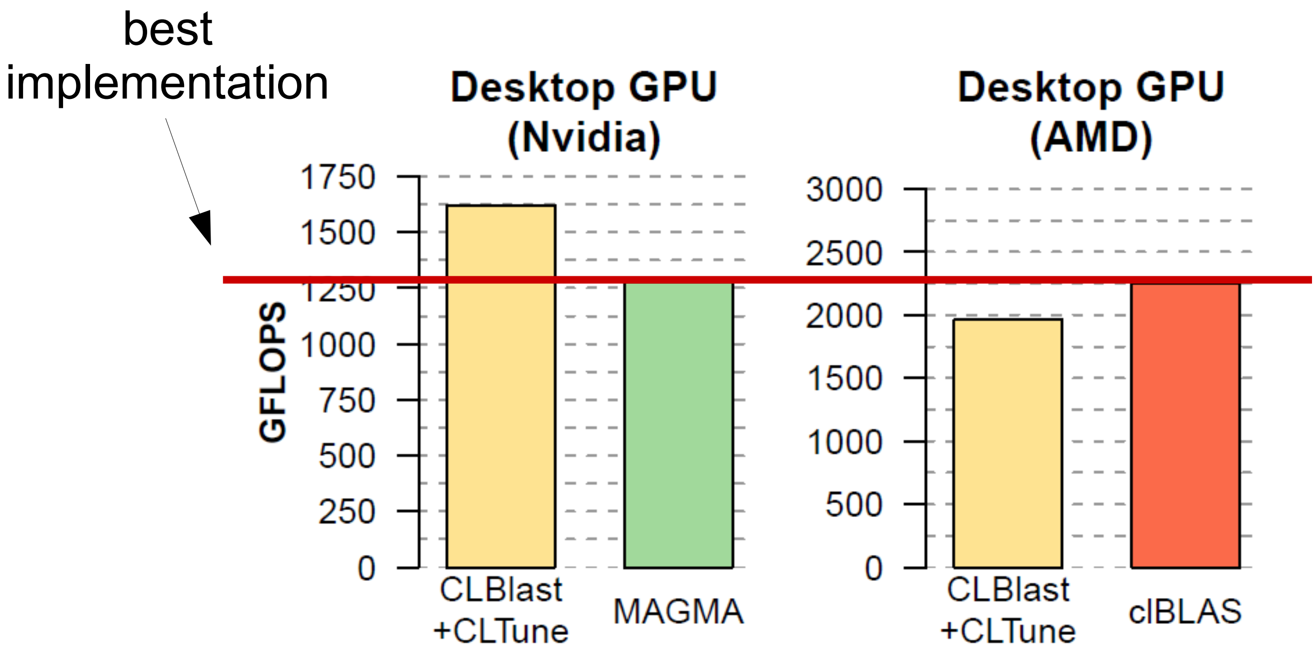
+ Performance is not portable even on same device class

Matrix-matrix multiplication auto-tuner



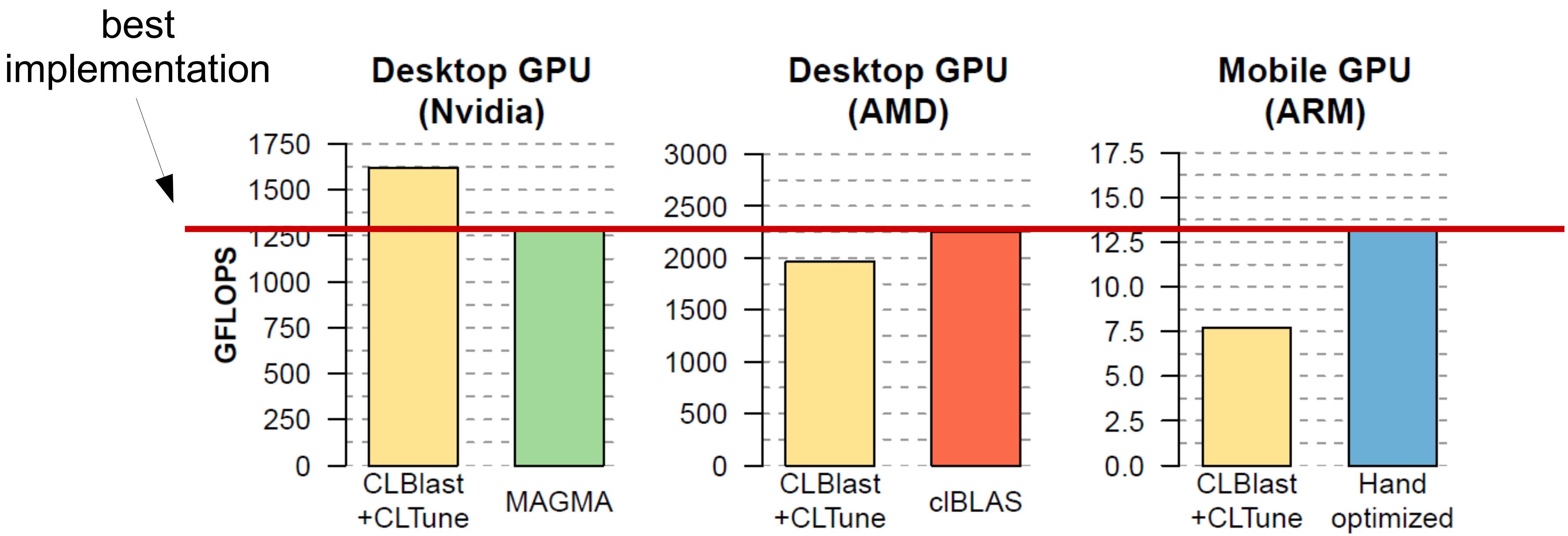
+ Performance is not portable even on same device class

Matrix-matrix multiplication auto-tuner



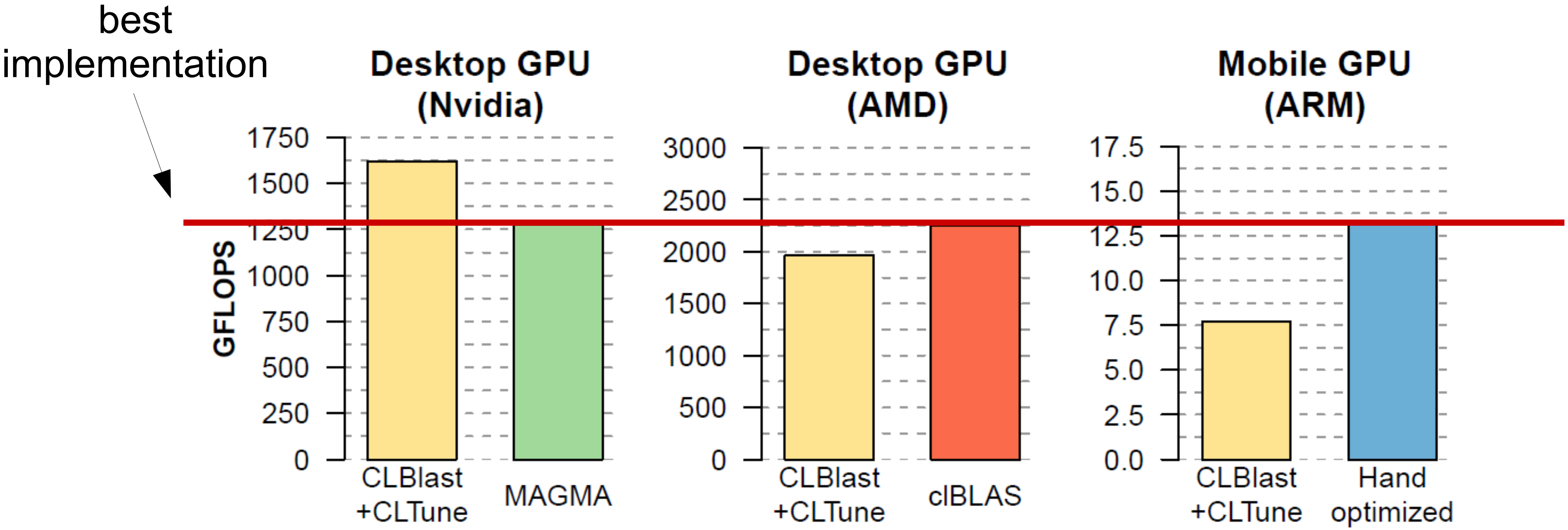
+ Performance is not portable even on same device class

Matrix-matrix multiplication auto-tuner



+ Performance is not portable even on same device class

Matrix-matrix multiplication auto-tuner



Auto-tuning alone fails to achieve portable performance

Need for high-level programming
+
Code generation

High-Level programming approaches for heterogeneous devices

- StreamIt (MIT)
  - Dataflow programming, architecture Independent
  - Multiple backend (tile architecture, C, FPGA, GPU)

- LiquidMetal (IBM)
  - Java + dataflow programming + (map / reduce)
  - C, OpenCL and FPGA backend

- Green-Marl (Stanford / Oracle)
  - DSL for graph analysis
  - operations described on graph's node or edgesl

- Halide (MIT)
  - DSL for imagine processing, functional in nature
  - GPU code generator

- TensorFlow (Google)
  - DSL library for AI applications
  - data flow model

Data flow & functional approaches:

- Hardware complexity is hidden away
  - → code portability

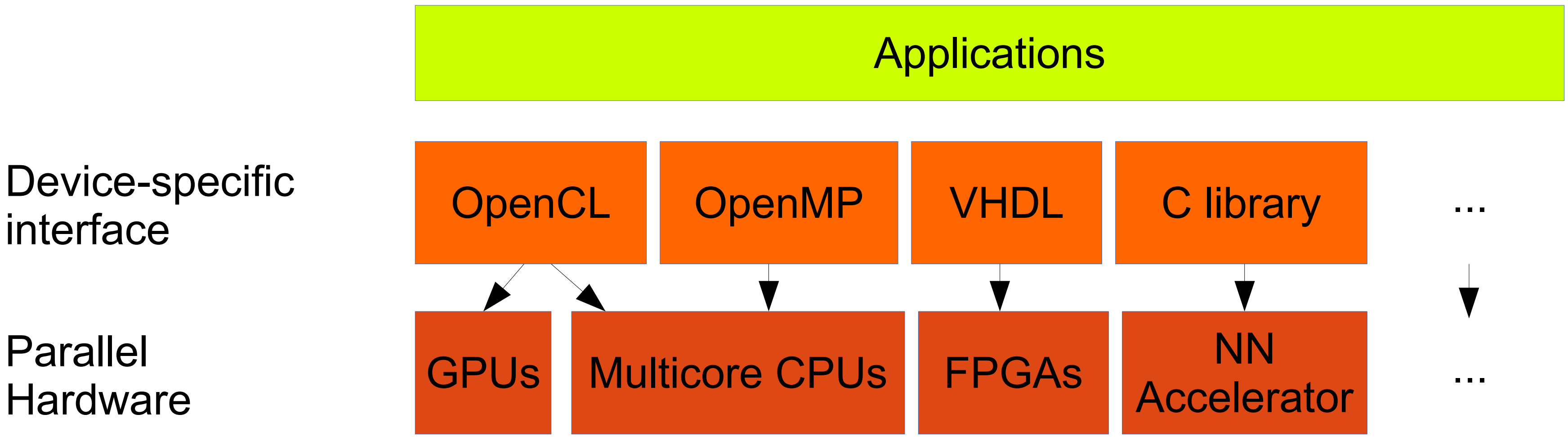
- Empowers the compiler
  - no need for complicated analysis
  - explicit data movement, no global state
  - can be mapped on various hardware
  - scheduling is easier

General purpose framework for data parallelism

- Delite (Stanford + EPFL)
  - framework for developing high-performance DSLs
  - provide built-in parallel primitives
  - Multicore + GPU code generation

- Lift (Edinburgh)
  - intermediate data-parallel language
  - compiler optimisations expressed as rewrite rules
  - Multicore + GPU code generation
  - + more to come (MPI, OpenMP, VHDL)

What we have



Applications

Device-specific interface

OpenCL

OpenMP

VHDL

C library

…

Parallel Hardware

GPUs

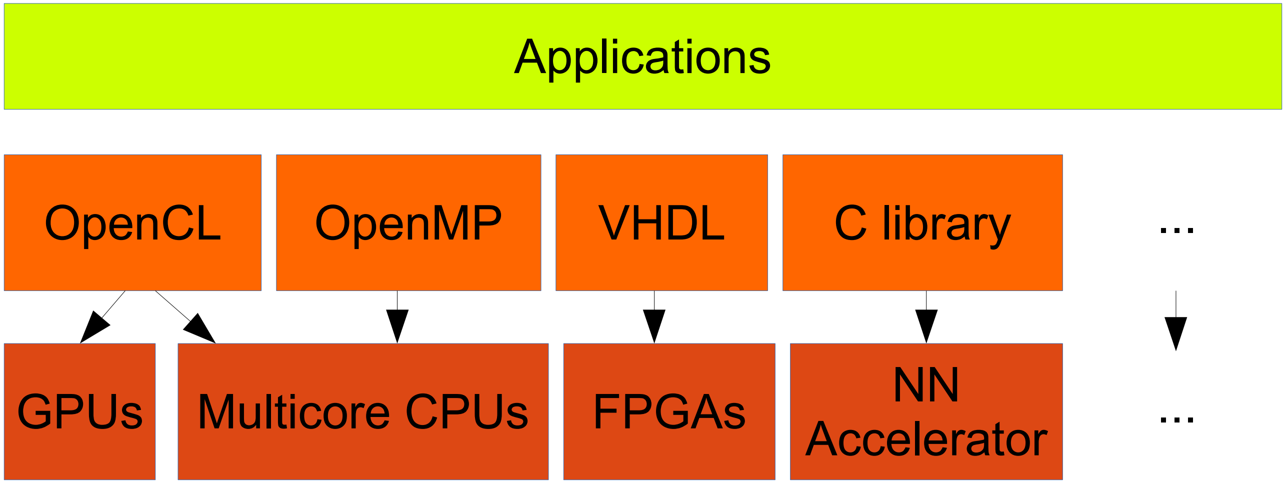
Multicore CPUs

FPGAs

NN Accelerator

…

What we need



Applications

OpenCL

OpenMP

VHDL

C library

…

GPUs

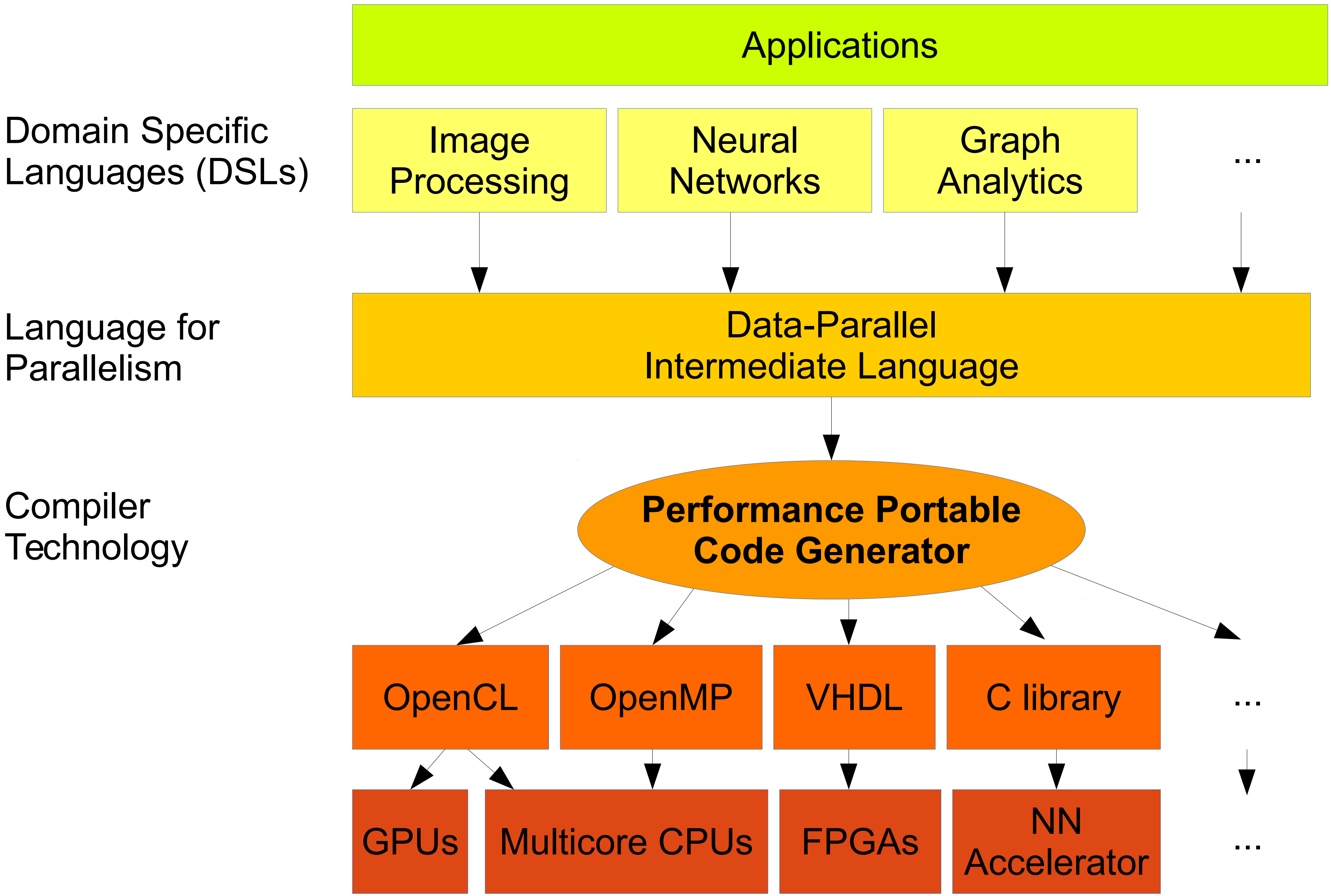
Multicore CPUs

FPGAs

NN Accelerator

…

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

C library

...

GPUs

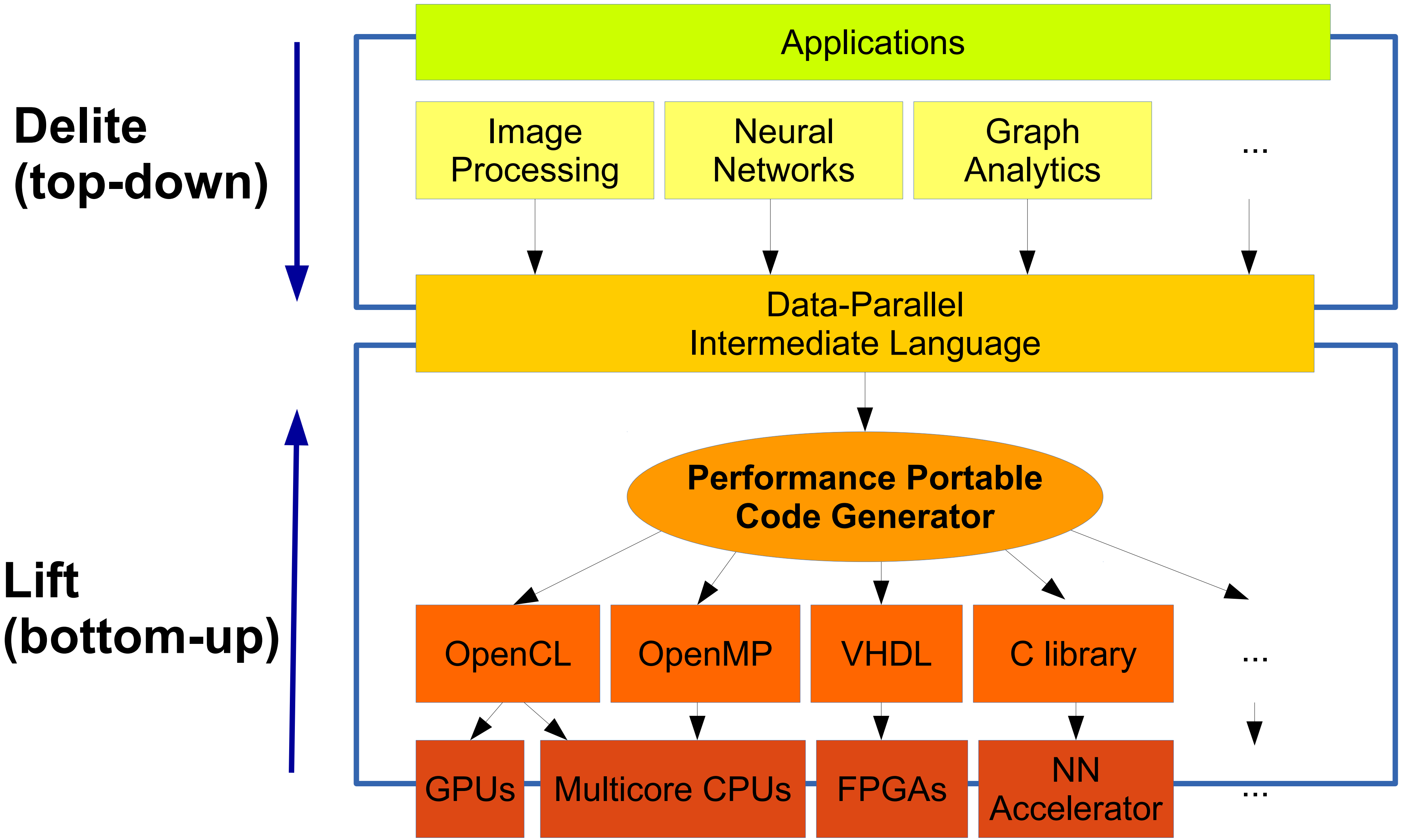
Multicore CPUs

FPGAs

NN Accelerator

...

Different starting point



Delite (top-down)

Lift (bottom-up)

Applications

Image Processing

Neural Networks

Graph Analytics

...

Data-Parallel Intermediate Language

Performance Portable Code Generator

OpenCL

OpenMP

VHDL

C library

...

GPUs

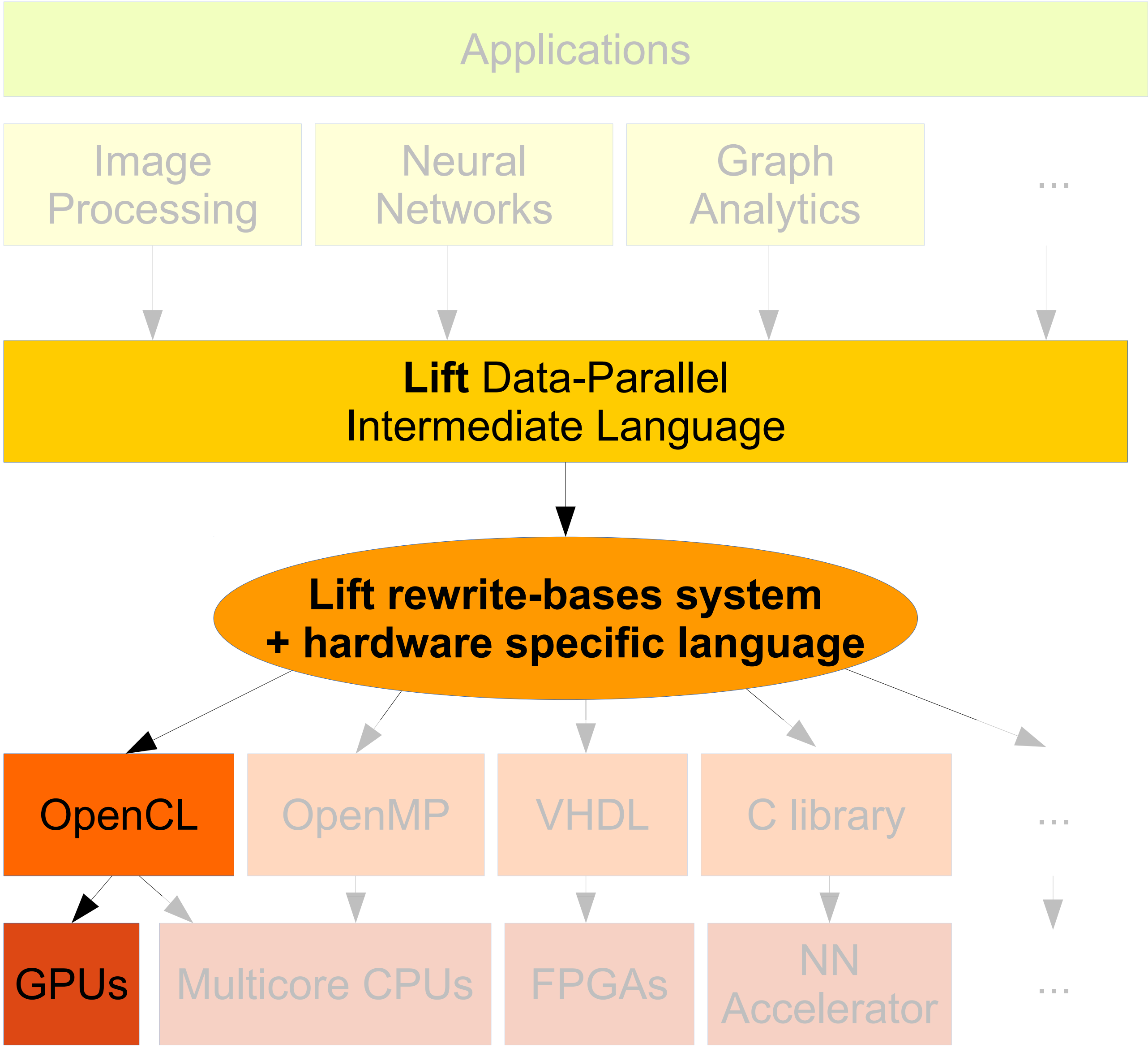
Multicore CPUs

FPGAs

NN Accelerator

...

What this talk is about



Applications

Image Processing

Neural Networks

Graph Analytics

...

Lift Data-Parallel Intermediate Language

Lift rewrite-bases system + hardware specific language

OpenCL

OpenMP

VHDL

C library

...

GPUs

Multicore CPUs

FPGAs

NN Accelerator

...

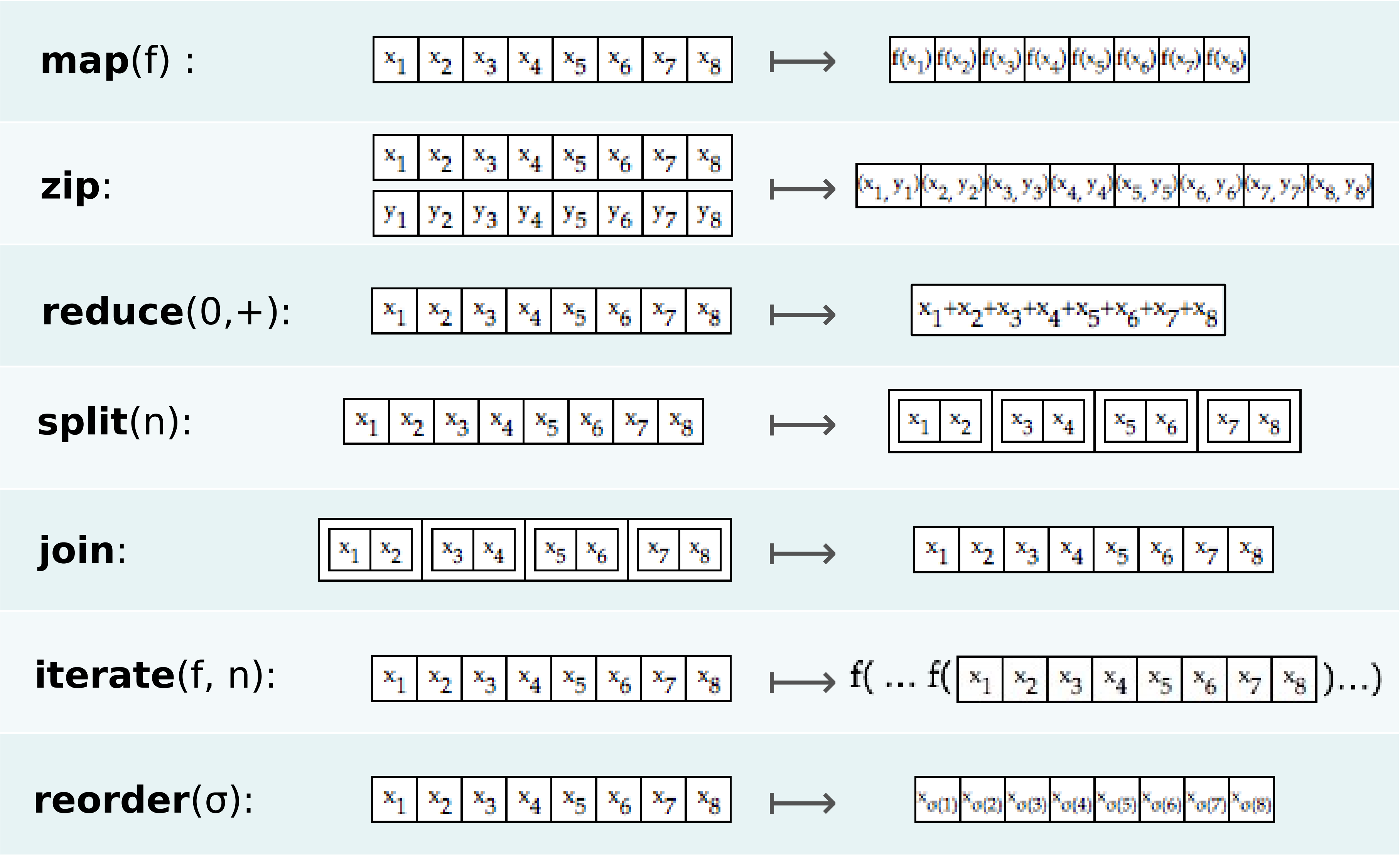
Rest of the talk

- Part I: Lift data parallel language
- Part II: Optimisations as rewrite rules
- Part III: Code generation details

Part I

Lift: a Functional Data Parallel Language

Lift Intermediate Language



map(f) : $x_1\ x_2\ x_3\ x_4\ x_5\ x_6\ x_7\ x_8 \longmapsto f(x_1)\ f(x_2)\ f(x_3)\ f(x_4)\ f(x_5)\ f(x_6)\ f(x_7)\ f(x_8)$

zip: $x_1\ x_2\ x_3\ x_4\ x_5\ x_6\ x_7\ x_8$; $y_1\ y_2\ y_3\ y_4\ y_5\ y_6\ y_7\ y_8 \longmapsto (x_1, y_1)\ (x_2, y_2)\ (x_3, y_3)\ (x_4, y_4)\ (x_5, y_5)\ (x_6, y_6)\ (x_7, y_7)\ (x_8, y_8)$

reduce(0,+): $x_1\ x_2\ x_3\ x_4\ x_5\ x_6\ x_7\ x_8 \longmapsto x_1+x_2+x_3+x_4+x_5+x_6+x_7+x_8$

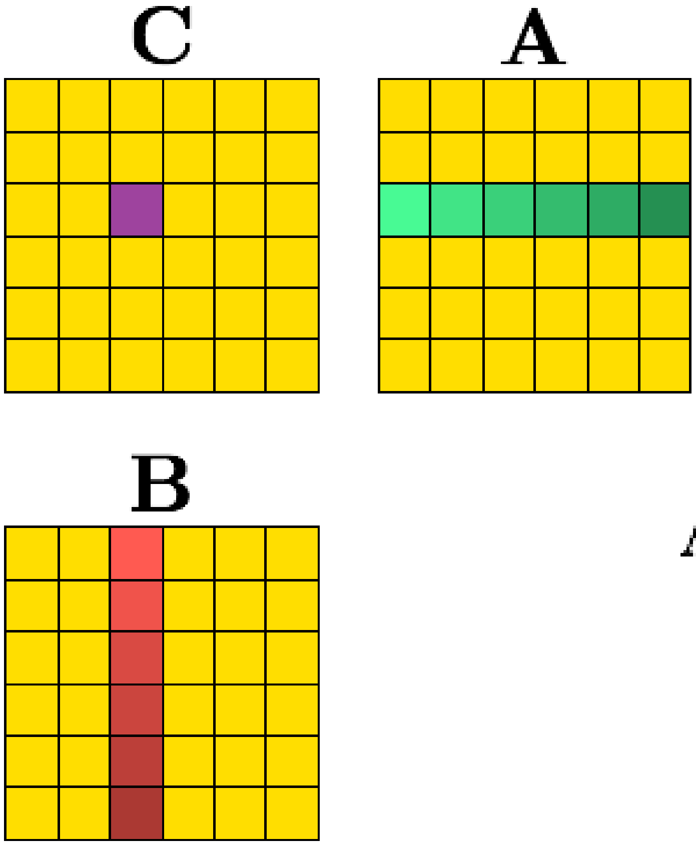
split(n): $x_1\ x_2\ x_3\ x_4\ x_5\ x_6\ x_7\ x_8 \longmapsto \boxed{x_1\ x_2}\ \boxed{x_3\ x_4}\ \boxed{x_5\ x_6}\ \boxed{x_7\ x_8}$

join: $\boxed{x_1\ x_2}\ \boxed{x_3\ x_4}\ \boxed{x_5\ x_6}\ \boxed{x_7\ x_8} \longmapsto x_1\ x_2\ x_3\ x_4\ x_5\ x_6\ x_7\ x_8$

iterate(f, n): $x_1\ x_2\ x_3\ x_4\ x_5\ x_6\ x_7\ x_8 \longmapsto f(\ \ldots\ f(\boxed{x_1\ x_2\ x_3\ x_4\ x_5\ x_6\ x_7\ x_8})\ldots)$

reorder(σ): $x_1\ x_2\ x_3\ x_4\ x_5\ x_6\ x_7\ x_8 \longmapsto x_{\sigma(1)}\ x_{\sigma(2)}\ x_{\sigma(3)}\ x_{\sigma(4)}\ x_{\sigma(5)}\ x_{\sigma(6)}\ x_{\sigma(7)}\ x_{\sigma(8)}$

Matrix Multiplication in Lift



```
A >> map(λ rowOfA ↦
  B >> map(λ colOfB ↦
     zip(rowOfA, colOfB) >>
     map(mult) >> reduce(0.0f,add)
  )
)
```

Lift high-level matrix multiplication

```
A >> map(λ rowOfA ↦
 B >> map(λ colOfB ↦
    zip(rowOfA, colOfB) >>
    map(mult) >> reduce(0.0f,add)
 )
)
```

Naive OpenCL version

```
kernel void KERNEL(
  const global float* restrict A,
  const global float* restrict B
  global float* C,
  int M, int K, int N)
{
  float acc = 0.0f;

  for (int i = 0; i < K; i += 1)
    acc = acc + A[id_A(glb_id_1, i)]
      * B[id_B(i, glb_id_0)];

  C[(id_C(glb_id_0, glb_id_1)] = acc;
}
```

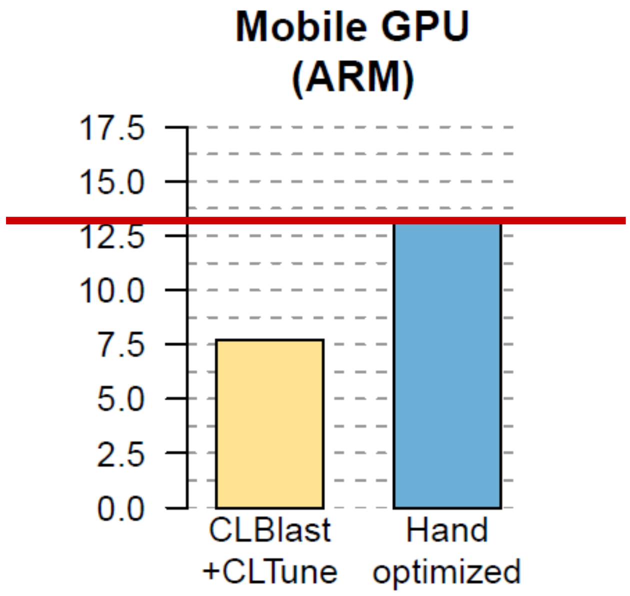
Lift high-level matrix multiplication

```
A >> map(λ rowOfA ↦
 B >> map(λ colOfB ↦
   zip(rowOfA, colOfB) >>
   map(mult) >> reduce(0.0f,add)
 )
)
```

Naive OpenCL version

```
kernel void KERNEL(
  const global float* restrict A,
  const global float* restrict B
  global float* C,
  int M, int K, int N)
{
  float acc = 0.0f;

  for (int i = 0; i < K; i += 1)
    acc = acc + A[id_A(glb_id_1, i)]
      * B[id_B(i, glb_id_0)];

  C[(id_C(glb_id_0, glb_id_1)] = acc;
}
```

?



How to achieve high performance?

Lift high-level matrix multiplication

```
A >> map(λ rowOfA ↦
 B >> map(λ colOfB ↦
   zip(rowOfA, colOfB) >>
   map(mult) >> reduce(0.0f,add)
 )
)
```

Naive OpenCL version

```
kernel void KERNEL(
  const global float* restrict A,
  const global float* restrict B
  global float* C,
  int M, int K, int N)
{
  float acc = 0.0f;

  for (int i = 0; i < K; i += 1)
    acc = acc + A[id_A(glb_id_1, i)]
      * B[id_B(i, glb_id_0)];

  C[(id_C(glb_id_0, glb_id_1)] = acc;
}
```

?

ARM Mali optimised version

```
int i = get_global_id(0);
int j = get_global_id(1);

float4 temp_0; float4 temp_1;
float4 temp_2; float4 temp_3;
float acc_0; float acc_1;
float acc_2; float acc_3;

for (int k = 0;k<K/4; k++) {

  temp_0 = mult4(vload4(k + K*i/2,A),
    vload4(k + K*j/2,B));
  acc_0 += temp_0.s0 + temp_0.s1 +
    temp_0.s2 + temp_0.s3;

  temp_1 = mult4(vload4(k + K*i/2,A,
    vload4(k + K + 2*K*j/4,B));
  acc_1 += temp_1.s0 + temp_1.s1 +
    temp_1.s2 + temp_1.s3;

  temp_2 = mult4(vload4(k + K + 2*K*i/4,A),
    vload4(k + K*j/2,B));
  acc_2 += temp_2.s0 + temp_2.s1 +
     temp_2.s2 + temp_2.s3;

  temp_3 = mult4(vload4(k + K + 2*K*i/4, A),
    vload4(k + K + 2*K*j/4, B));
  acc_3 += temp_3.s0 + temp_3.s1 +
    temp_3.s2 + temp_3.s3;
}
C[2*N*i + 2*j] = id(acc_0);
C[1 + 2*N*i + 2*j] = id(acc_1);
C[N + 2*N*i + 2*j] = id(acc_2);
C[1 + N + 2*N*i + 2*j] = id(acc_3);
```

How to achieve high performance?

Part II

Encoding optimisations choices as Rewrite Rules

Algorithmic Rewrite Rules

- Provably correct rewrite rules
- Express algorithmic and optimisations choices

Split-join rule:

$$map\ f \rightarrow join \circ map\ (map\ f) \circ split\ n$$

Map fusion rule:

$$map\ f \circ map\ g \rightarrow map\ (f \circ g)$$

Reduce rules:

$$reduce\ f\ \ z \rightarrow reduce\ f\ \ z \circ reducePart\ f\ \ z$$

$$reducePart\ f\ \ z \rightarrow reducePart\ f\ \ z \circ reorder$$

$$reducePart\ f\ \ z \rightarrow join\ \ \circ map\ (reducePart\ f\ \ z) \circ split\ n$$

$$reducePart\ f\ \ z \rightarrow iterate\ n\ (reducePart\ f\ \ z)$$

...

```
A >> map(λ rowOfA ↦
  B >> map(λ colOfB ↦
    zip(rowOfA, colOfB) >>
    map(mult) >> reduce(0.0f,add)
  )
)
```

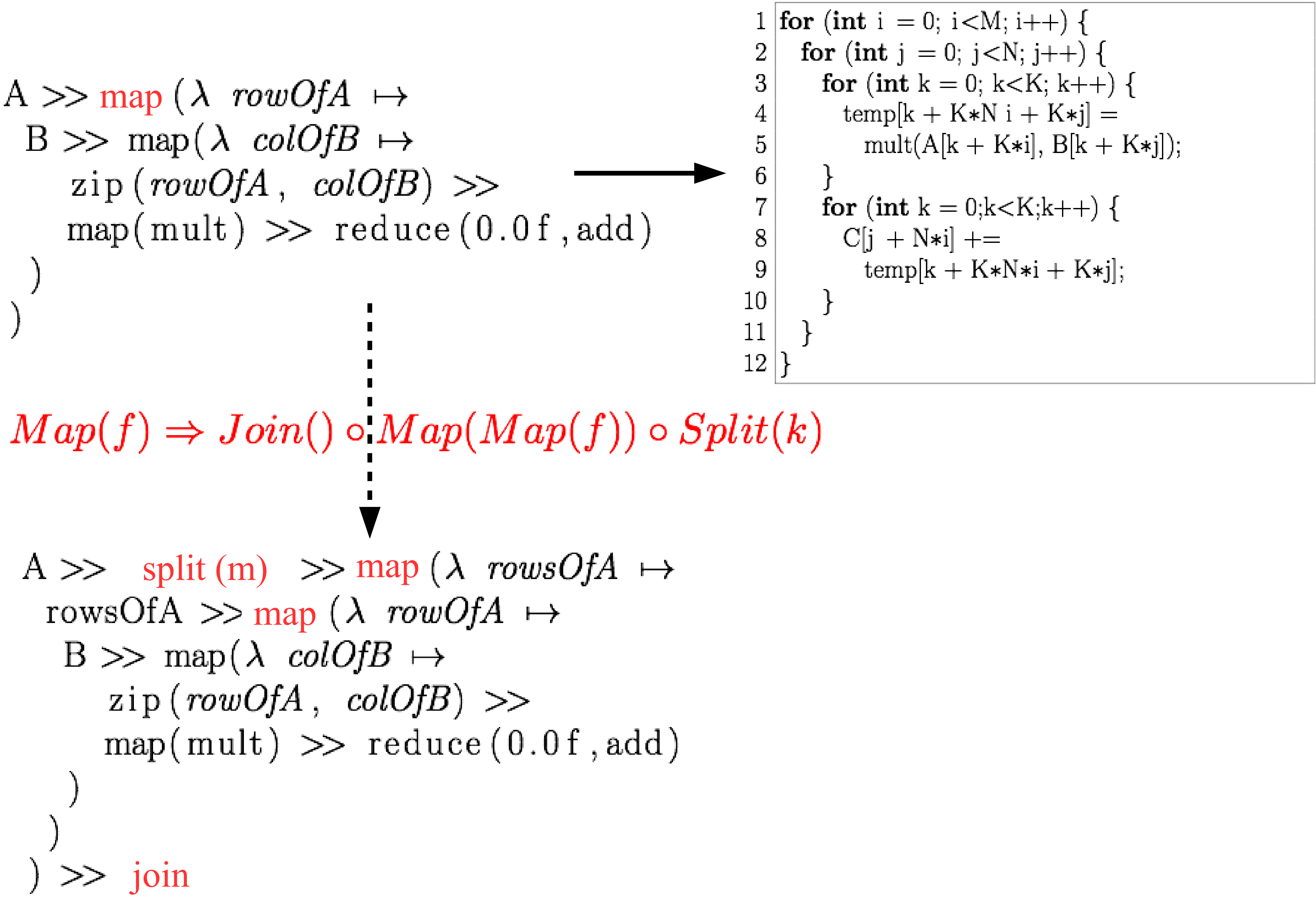
```
A >> map(λ rowOfA ↦
  B >> map(λ colOfB ↦
    zip(rowOfA, colOfB) >>
    map(mult) >> reduce(0.0f,add)
  )
)
```

```
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] =
        mult(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];
    }
  }
}
```

```
A >> map (λ rowOfA ↦
  B >> map(λ colOfB ↦
    zip(rowOfA, colOfB) >>
    map(mult) >> reduce(0.0f, add)
  )
)
```

```
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] =
        mult(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];
    }
  }
}
```

$$Map(f) \Rightarrow Join() \circ Map(Map(f)) \circ Split(k)$$

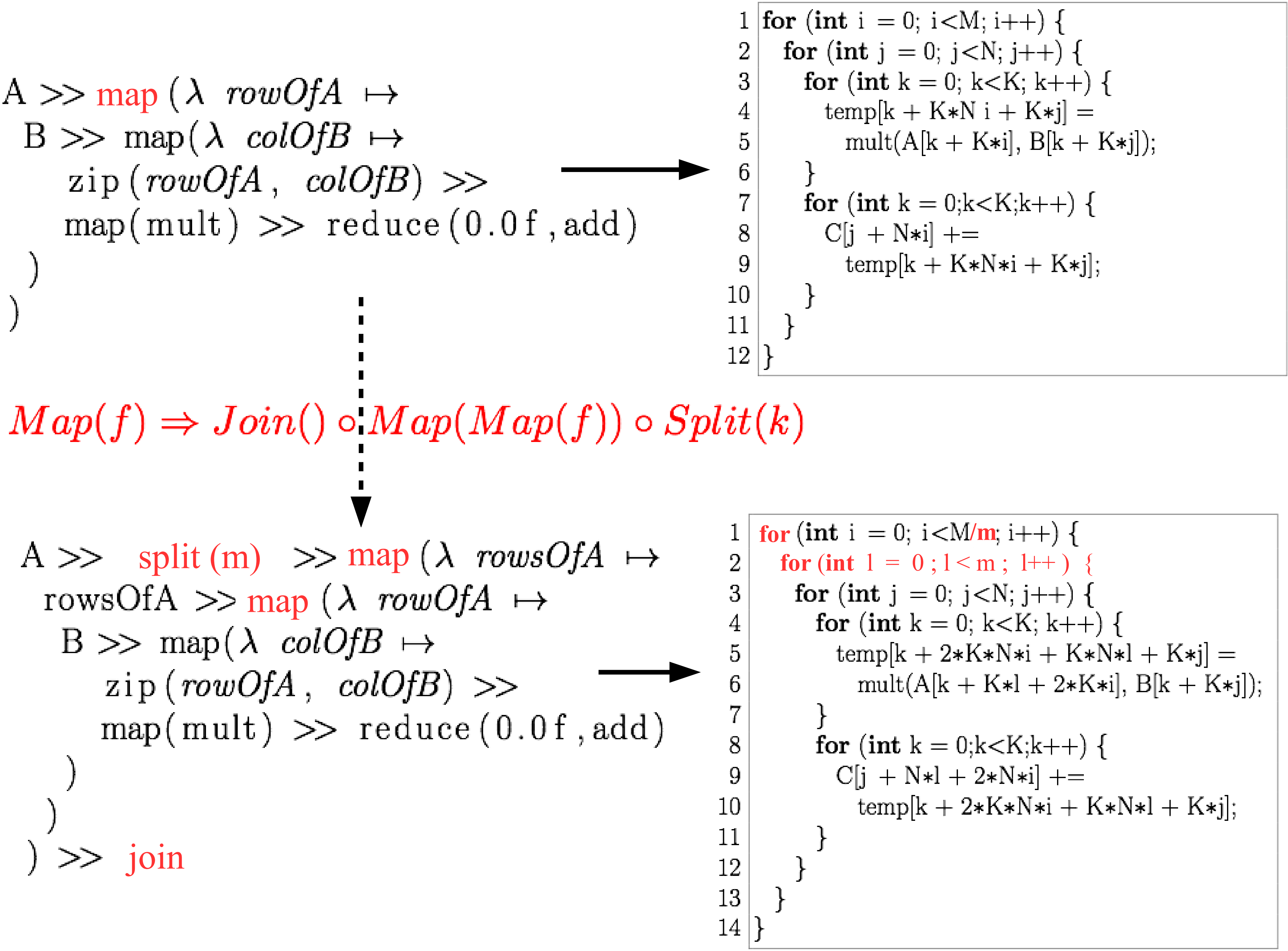


```
A >> map (λ rowOfA ↦
  B >> map(λ colOfB ↦
    zip(rowOfA, colOfB) >>
    map(mult) >> reduce(0.0f,add)
  )
)
```

```
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] =
        mult(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];
    }
  }
}
```

$$Map(f) \Rightarrow Join() \circ Map(Map(f)) \circ Split(k)$$

```
A >> split (m) >> map (λ rowsOfA ↦
  rowsOfA >> map (λ rowOfA ↦
    B >> map(λ colOfB ↦
      zip(rowOfA, colOfB) >>
      map(mult) >> reduce(0.0f,add)
    )
  )
) >> join
```



```
A >> map (λ rowOfA ↦
 B >> map(λ colOfB ↦
   zip(rowOfA, colOfB) >>
   map(mult) >> reduce(0.0f,add)
 )
)
```

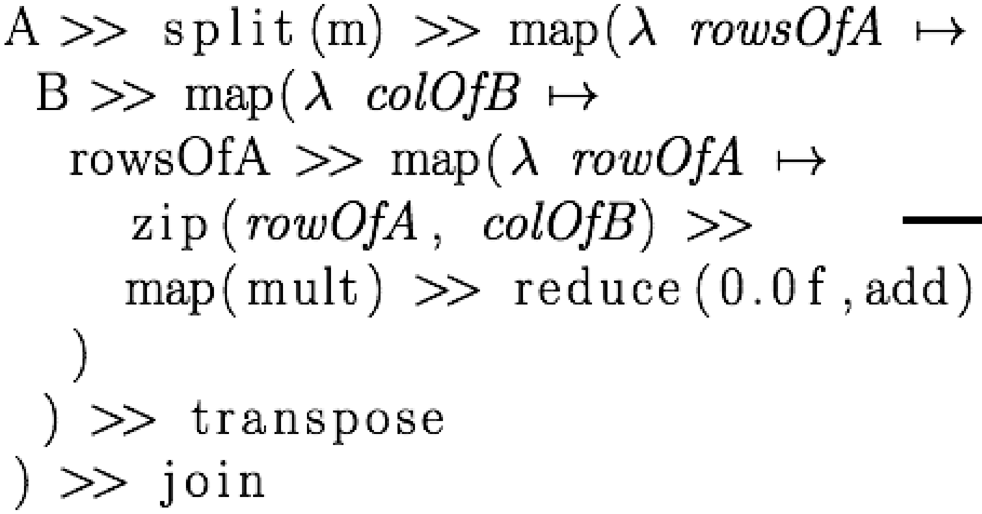
```
for (int i = 0; i<M; i++) {
  for (int j = 0; j<N; j++) {
    for (int k = 0; k<K; k++) {
      temp[k + K*N i + K*j] =
        mult(A[k + K*i], B[k + K*j]);
    }
    for (int k = 0;k<K;k++) {
      C[j + N*i] +=
        temp[k + K*N*i + K*j];
    }
  }
}
```

$$Map(f) \Rightarrow Join() \circ Map(Map(f)) \circ Split(k)$$

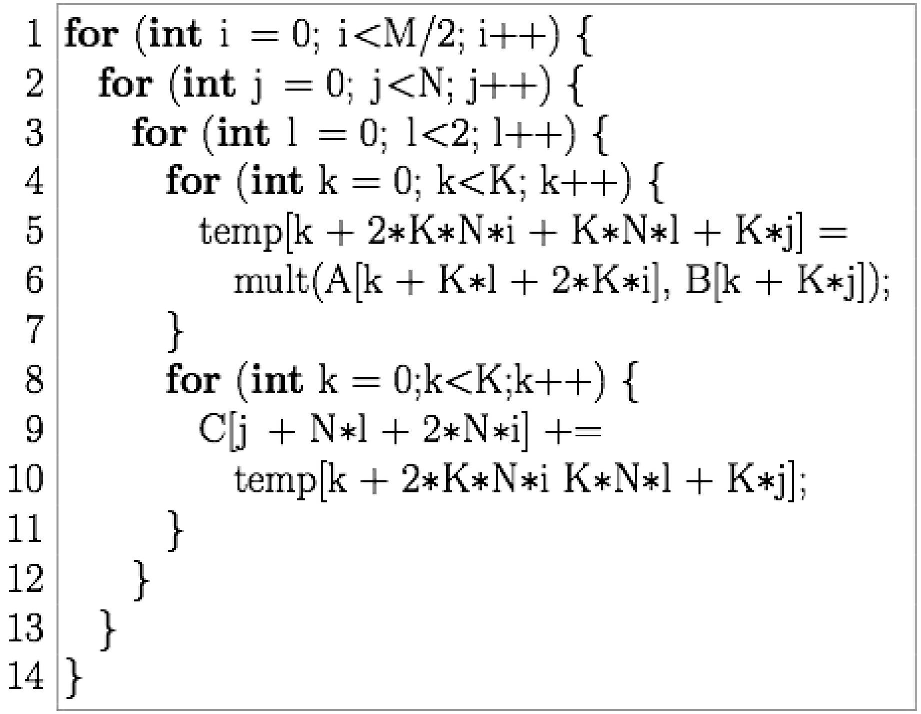
```
A >> split (m) >> map (λ rowsOfA ↦
 rowsOfA >> map (λ rowOfA ↦
  B >> map(λ colOfB ↦
    zip(rowOfA, colOfB) >>
    map(mult) >> reduce(0.0f,add)
  )
 )
) >> join
```

```
for (int i = 0; i<M/m; i++) {
  for (int l = 0 ; l < m ; l++ ) {
    for (int j = 0; j<N; j++) {
      for (int k = 0; k<K; k++) {
        temp[k + 2*K*N*i + K*N*l + K*j] =
          mult(A[k + K*l + 2*K*i], B[k + K*j]);
      }
      for (int k = 0;k<K;k++) {
        C[j + N*l + 2*N*i] +=
          temp[k + 2*K*N*i + K*N*l + K*j];
      }
    }
  }
}
```

Map interchanged

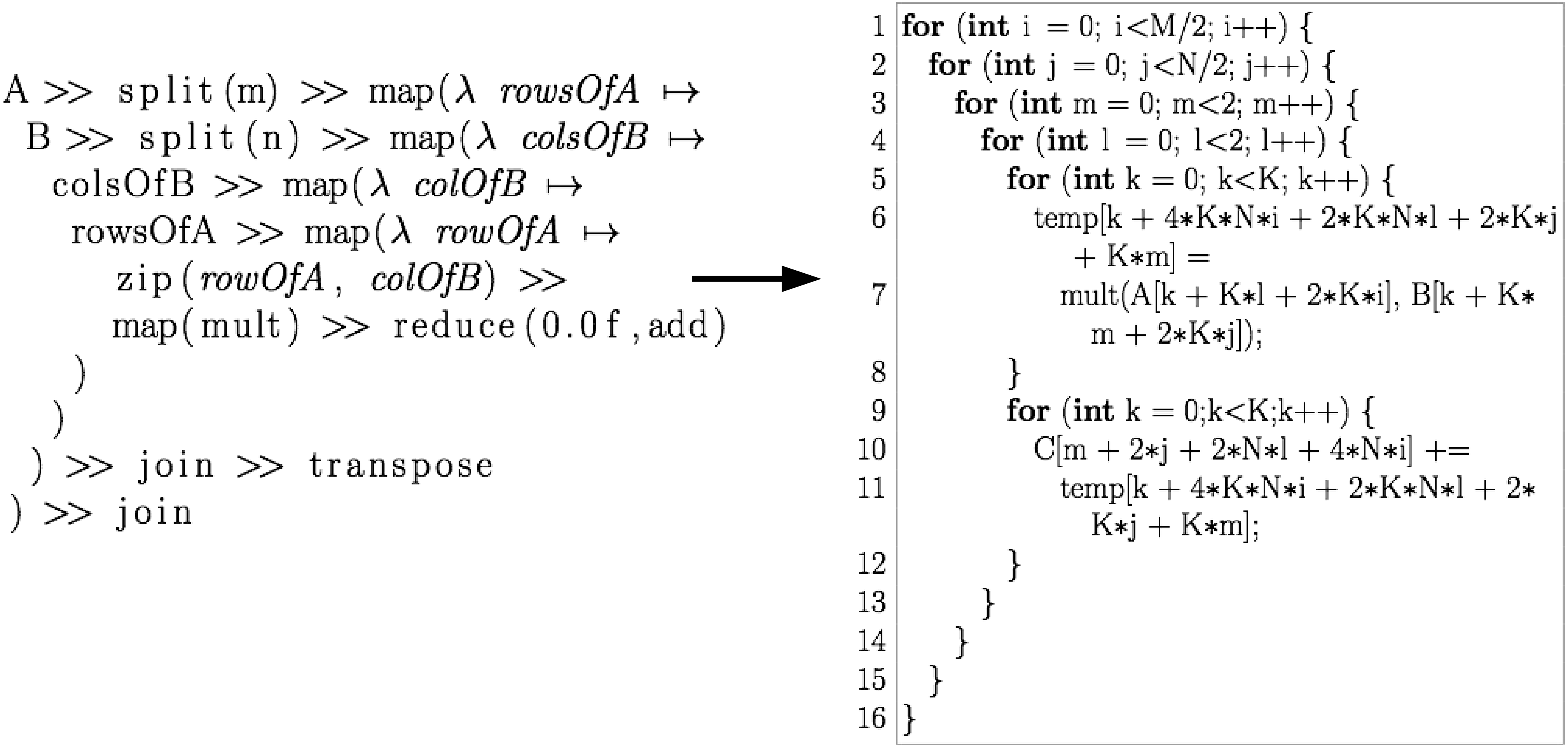


```
A >> split(m) >> map(λ rowsOfA ↦
 B >> map(λ colOfB ↦
  rowsOfA >> map(λ rowOfA ↦
    zip(rowOfA, colOfB) >>
    map(mult) >> reduce(0.0f,add)
  )
 ) >> transpose
) >> join
```



```
for (int i = 0; i<M/2; i++) {
  for (int j = 0; j<N; j++) {
    for (int l = 0; l<2; l++) {
      for (int k = 0; k<K; k++) {
        temp[k + 2*K*N*i + K*N*l + K*j] =
          mult(A[k + K*l + 2*K*i], B[k + K*j]);
      }
      for (int k = 0;k<K;k++) {
        C[j + N*l + 2*N*i] +=
          temp[k + 2*K*N*i K*N*l + K*j];
      }
    }
  }
}
```

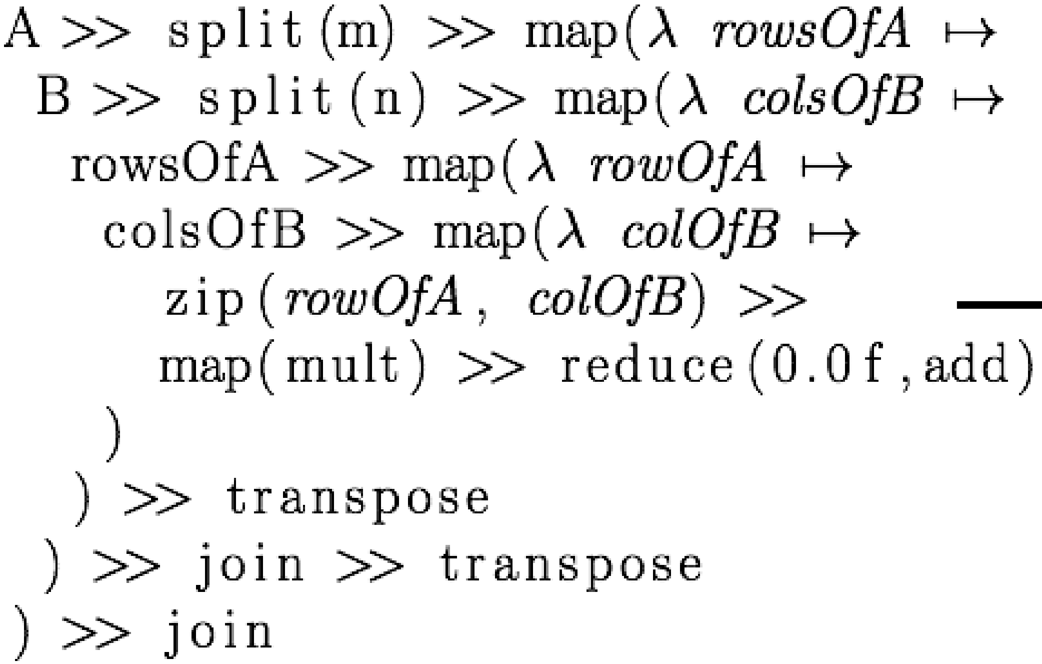
Split-join rule



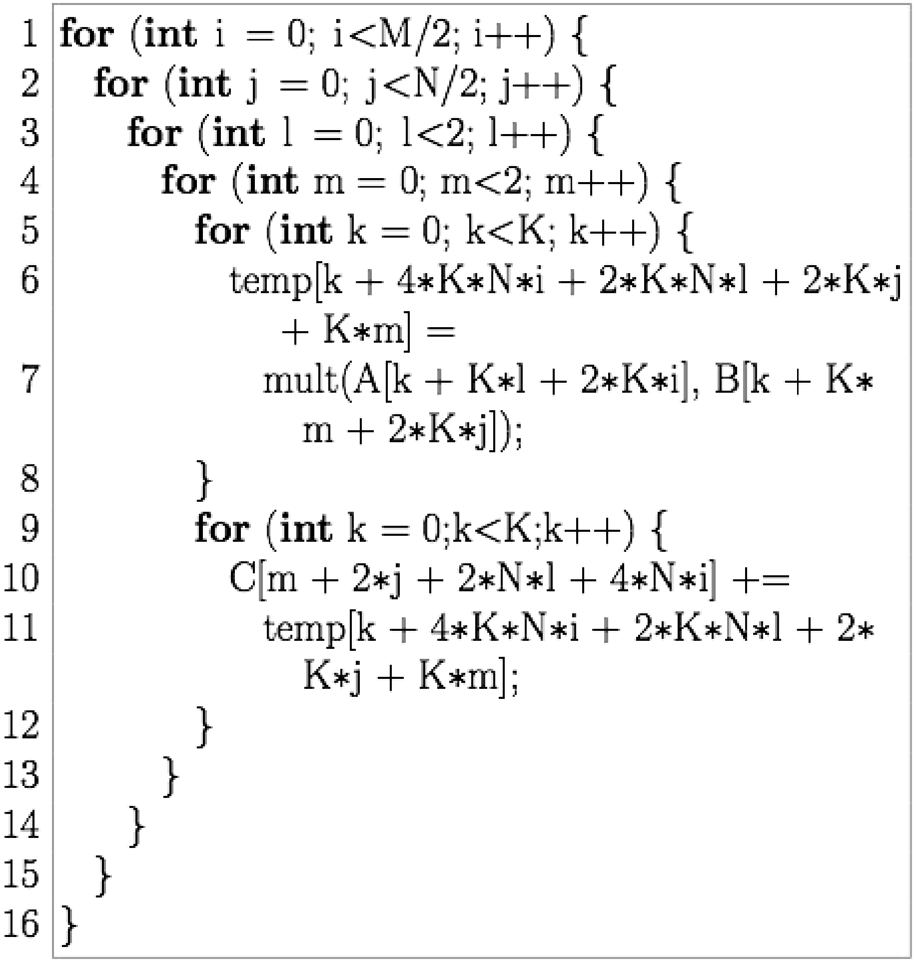
```
A >> split(m) >> map(λ rowsOfA ↦
 B >> split(n) >> map(λ colsOfB ↦
   colsOfB >> map(λ colOfB ↦
    rowsOfA >> map(λ rowOfA ↦
       zip(rowOfA, colOfB) >>
       map(mult) >> reduce(0.0f,add)
     )
    )
 ) >> join >> transpose
) >> join
```

```
for (int i = 0; i<M/2; i++) {
  for (int j = 0; j<N/2; j++) {
    for (int m = 0; m<2; m++) {
      for (int l = 0; l<2; l++) {
        for (int k = 0; k<K; k++) {
          temp[k + 4*K*N*i + 2*K*N*l + 2*K*j + K*m] =
            mult(A[k + K*l + 2*K*i], B[k + K*m + 2*K*j]);
        }
        for (int k = 0;k<K;k++) {
          C[m + 2*j + 2*N*l + 4*N*i] +=
            temp[k + 4*K*N*i + 2*K*N*l + 2*K*j + K*m];
        }
      }
    }
  }
}
```

Map interchanged



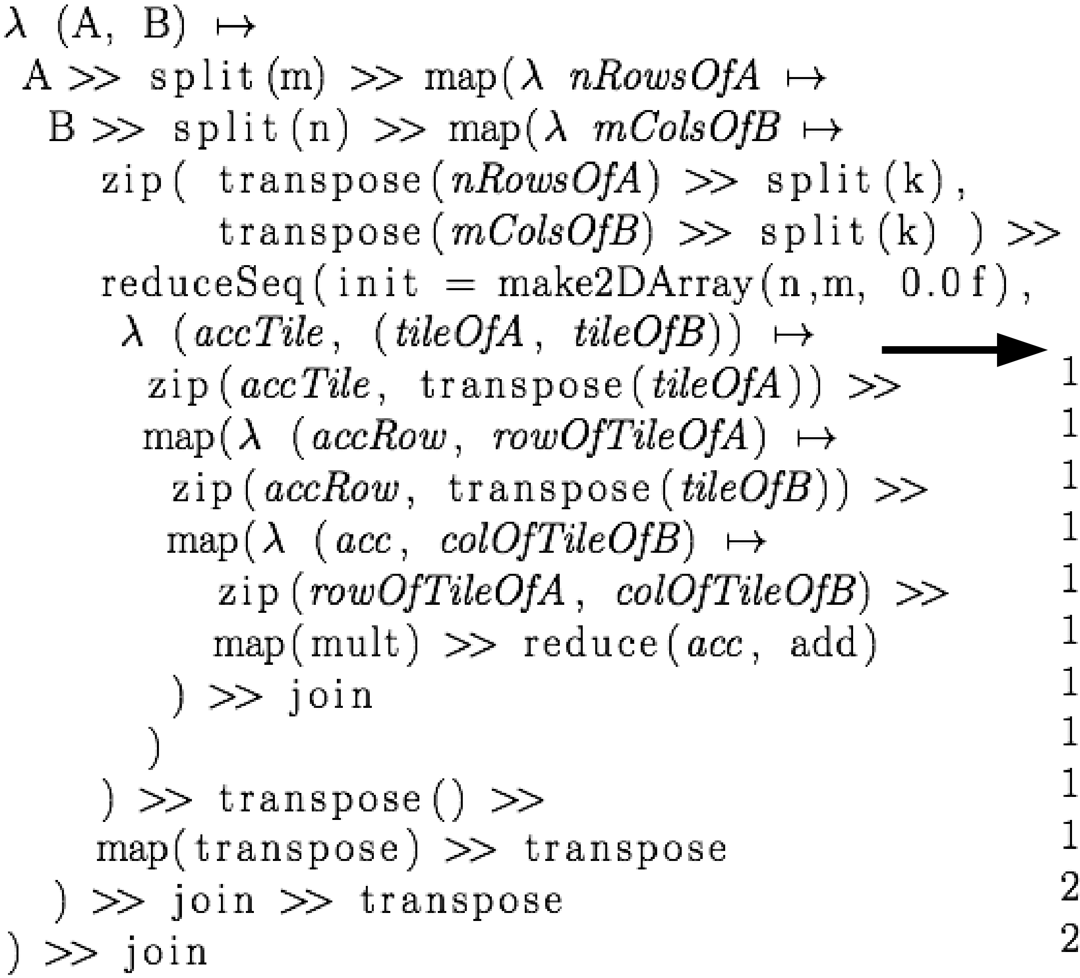
```
A >> split(m) >> map(λ rowsOfA ↦
  B >> split(n) >> map(λ colsOfB ↦
    rowsOfA >> map(λ rowOfA ↦
      colsOfB >> map(λ colOfB ↦
          zip(rowOfA, colOfB) >>
          map(mult) >> reduce(0.0f,add)
      )
    ) >> transpose
  ) >> join >> transpose
) >> join
```



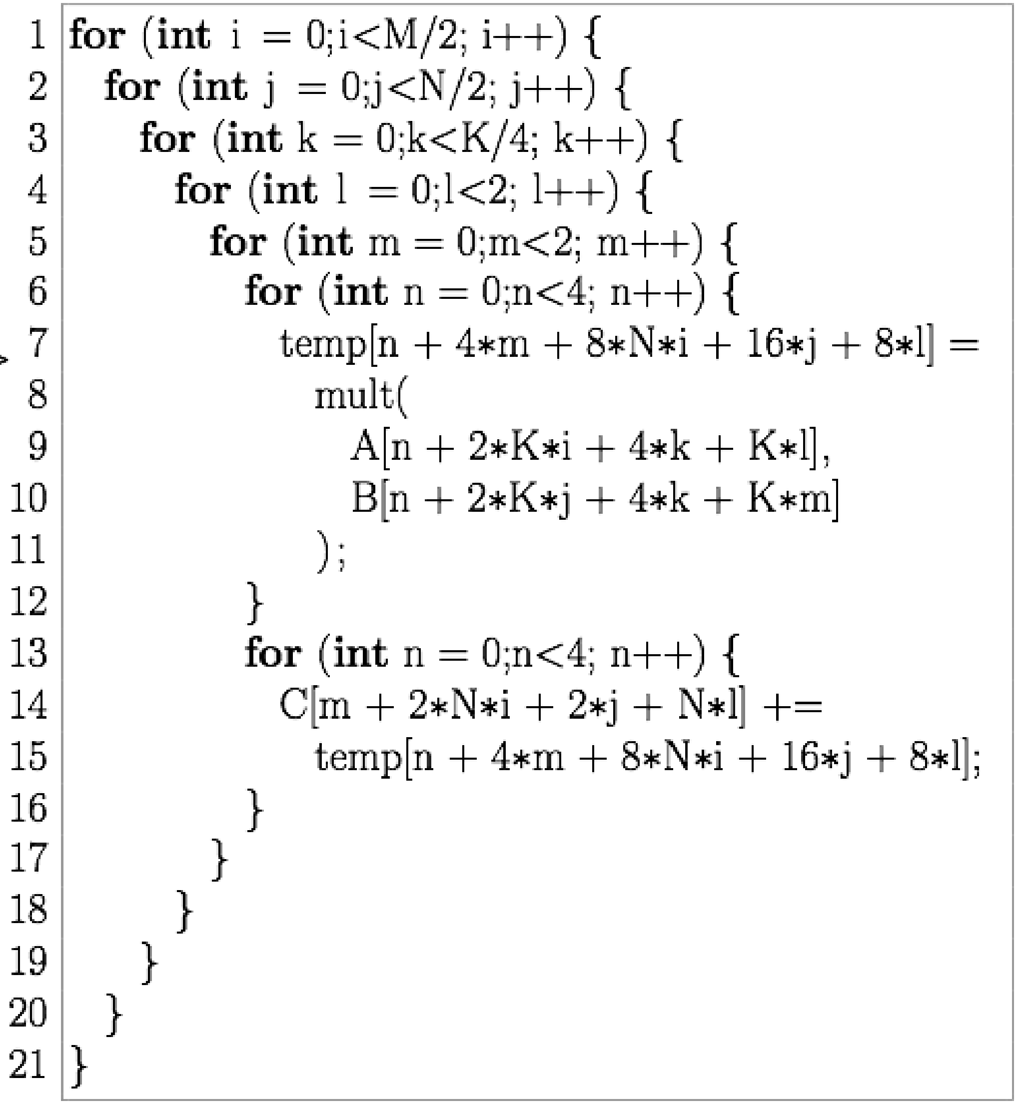
```
for (int i = 0; i<M/2; i++) {
  for (int j = 0; j<N/2; j++) {
    for (int l = 0; l<2; l++) {
      for (int m = 0; m<2; m++) {
        for (int k = 0; k<K; k++) {
          temp[k + 4*K*N*i + 2*K*N*l + 2*K*j + K*m] =
            mult(A[k + K*l + 2*K*i], B[k + K*m + 2*K*j]);
        }
        for (int k = 0;k<K;k++) {
          C[m + 2*j + 2*N*l + 4*N*i] +=
            temp[k + 4*K*N*i + 2*K*N*l + 2*K*j + K*m];
        }
      }
    }
  }
}
```

A few rewrite steps later…

Tiled version



```
λ (A, B) ↦
 A >> split(m) >> map(λ nRowsOfA ↦
  B >> split(n) >> map(λ mColsOfB ↦
     zip( transpose(nRowsOfA) >> split(k),
          transpose(mColsOfB) >> split(k) ) >>
     reduceSeq(init = make2DArray(n,m, 0.0f),
      λ (accTile, (tileOfA, tileOfB)) ↦
       zip(accTile, transpose(tileOfA)) >>
       map(λ (accRow, rowOfTileOfA) ↦
        zip(accRow, transpose(tileOfB)) >>
        map(λ (acc, colOfTileOfB) ↦
          zip(rowOfTileOfA, colOfTileOfB) >>
          map(mult) >> reduce(acc, add)
        ) >> join
       )
     ) >> transpose() >>
     map(transpose) >> transpose
  ) >> join >> transpose
) >> join
```



```
for (int i = 0;i<M/2; i++) {
  for (int j = 0;j<N/2; j++) {
    for (int k = 0;k<K/4; k++) {
      for (int l = 0;l<2; l++) {
        for (int m = 0;m<2; m++) {
          for (int n = 0;n<4; n++) {
            temp[n + 4*m + 8*N*i + 16*j + 8*l] =
              mult(
                A[n + 2*K*i + 4*k + K*l],
                B[n + 2*K*j + 4*k + K*m]
              );
          }
          for (int n = 0;n<4; n++) {
            C[m + 2*N*i + 2*j + N*l] +=
              temp[n + 4*m + 8*N*i + 16*j + 8*l];
          }
        }
      }
    }
  }
}
```

Low-Level OpenCL-specific Extensions

Lift OpenCL-Specific Primitives

map-global(f)	*OpenCL thread hierarchy*
map-workgroup(f)	
map-local(f)	
map-sequential(f)	*Sequential implementations*
reduce-sequential(z,f)	
toLocal(f)	*Memory address spaces*
toGlobal(f)	
vect-n(f)	*Vectorisation*
asScalar	
asVector-n	

OpenCL-specific rewrites (examples)

OpenCL thread hierarchy:

$$\boxed{\mathrm{map}(f) \implies \mathrm{mapGlb}_{\{0,1,2\}}(f)} \qquad \boxed{\mathrm{map}(f) \implies \mathrm{mapLcl}_{\{0,1,2\}}(f)}$$

OpenCL memory hierarchy:

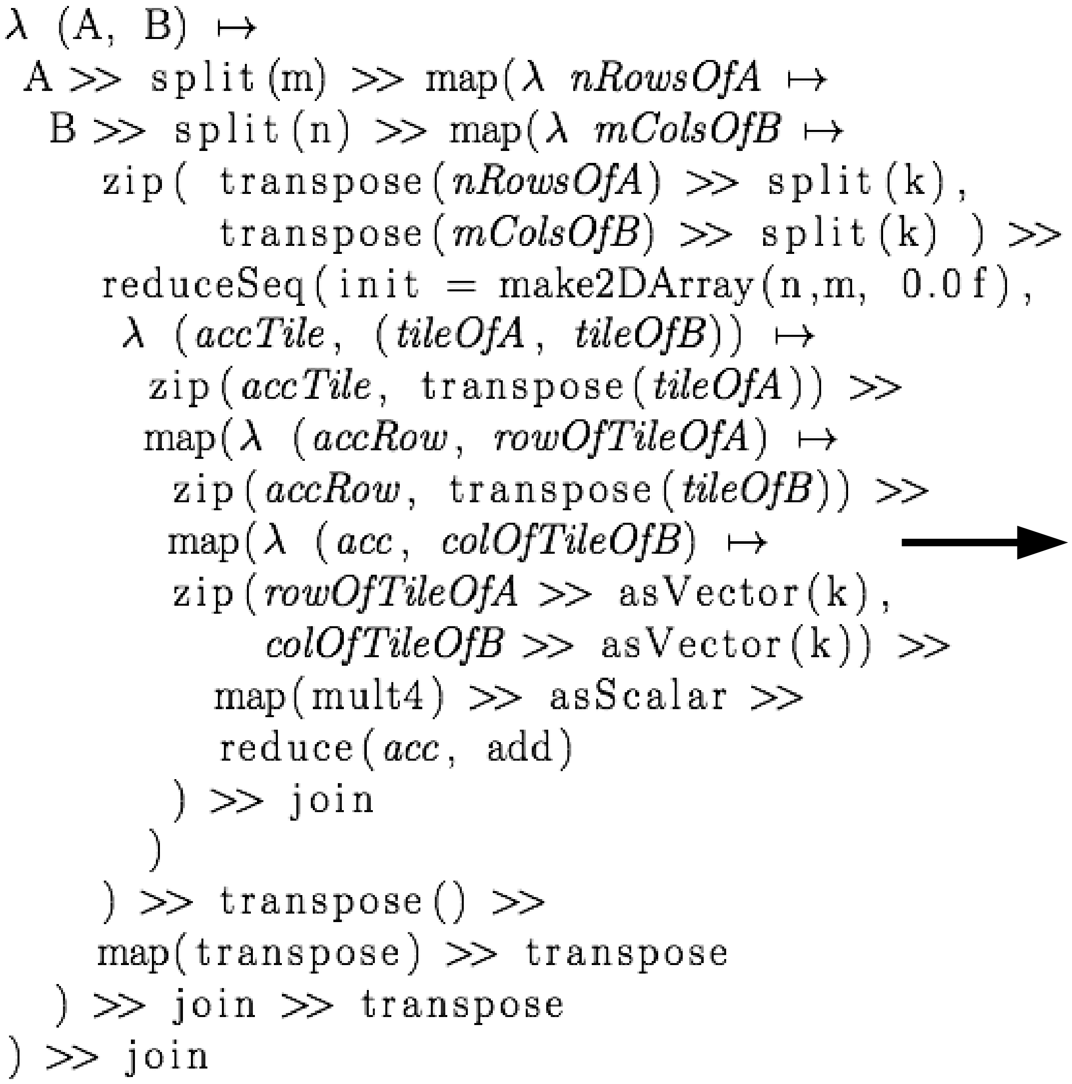
$$\boxed{f \implies \mathtt{toPrivate}(f)} \qquad \boxed{f \implies \mathtt{toLocal}(f)} \qquad \boxed{f \implies \mathtt{toGlobal}(f)}$$

OpenCL vector types and operations:

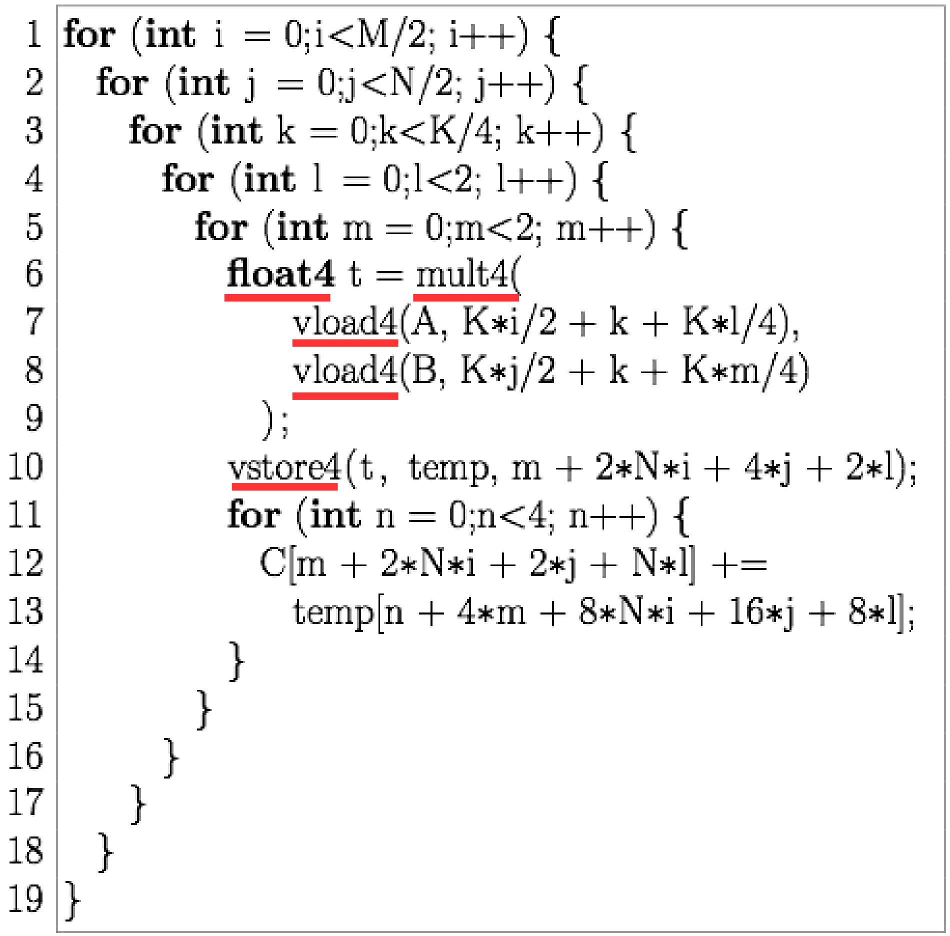
$$\boxed{\begin{array}{l} \mathrm{map}(f) \implies \mathtt{asVector}(n,\ b) \\ \qquad \gg \mathrm{map}(\mathtt{vectorize}(n,\ f)) \gg \mathtt{asScalar} \end{array}}$$

Let's start applying OpenCL-specific rules

Vectorisation

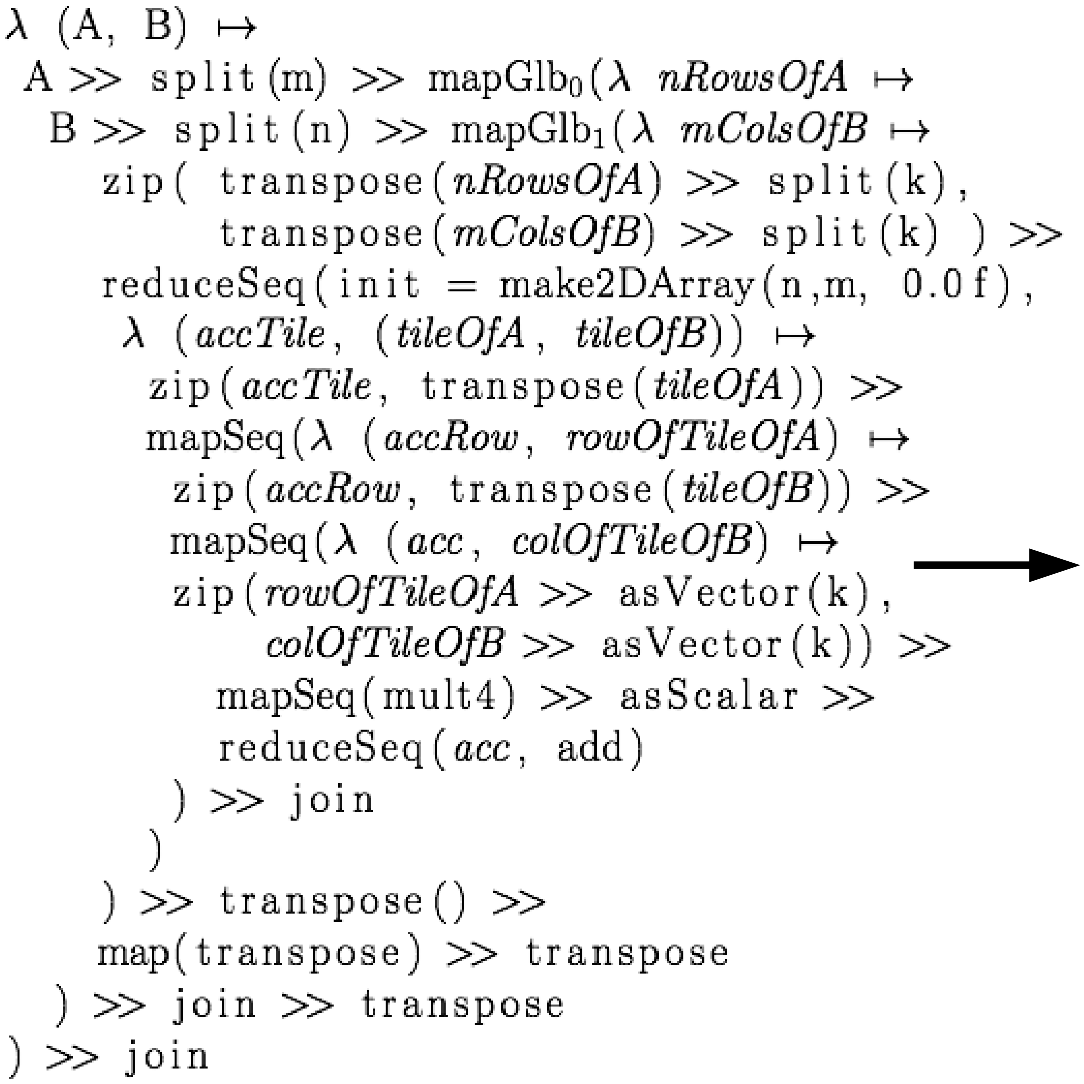


```
λ (A, B) ↦
 A >> split(m) >> map(λ nRowsOfA ↦
  B >> split(n) >> map(λ mColsOfB ↦
    zip( transpose(nRowsOfA) >> split(k),
         transpose(mColsOfB) >> split(k) ) >>
    reduceSeq(init = make2DArray(n,m, 0.0f),
     λ (accTile, (tileOfA, tileOfB)) ↦
      zip(accTile, transpose(tileOfA)) >>
      map(λ (accRow, rowOfTileOfA) ↦
       zip(accRow, transpose(tileOfB)) >>
       map(λ (acc, colOfTileOfB) ↦
       zip(rowOfTileOfA >> asVector(k),
           colOfTileOfB >> asVector(k)) >>
         map(mult4) >> asScalar >>
         reduce(acc, add)
       ) >> join
      )
    ) >> transpose() >>
    map(transpose) >> transpose
  ) >> join >> transpose
) >> join
```

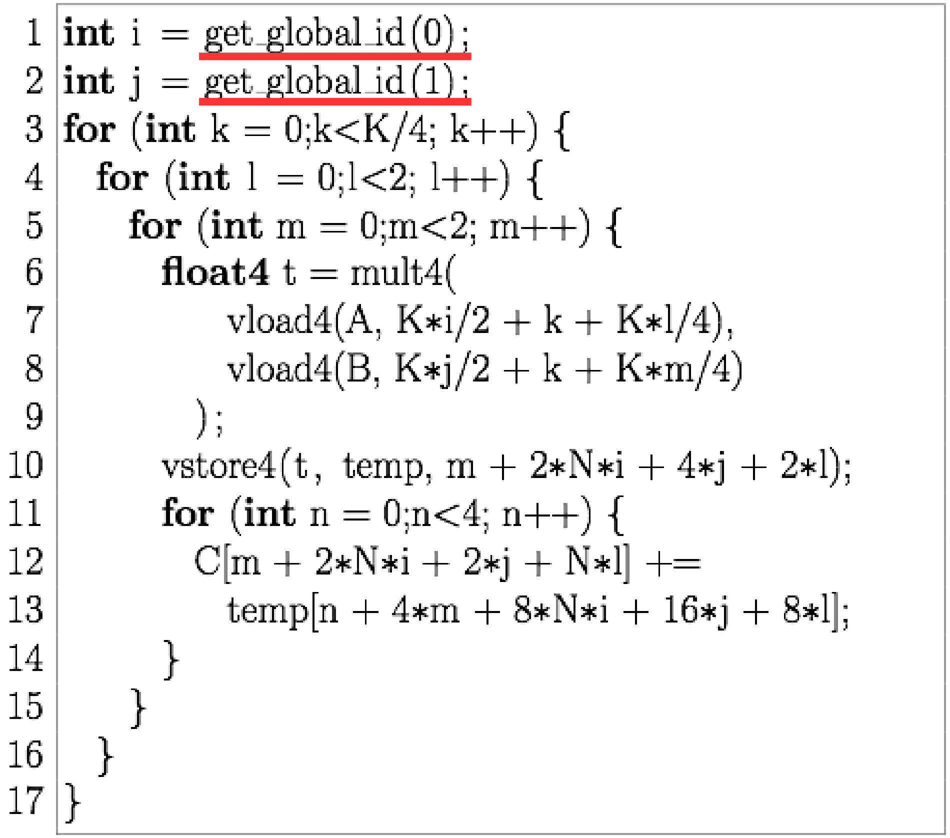


```
for (int i = 0;i<M/2; i++) {
  for (int j = 0;j<N/2; j++) {
    for (int k = 0;k<K/4; k++) {
      for (int l = 0;l<2; l++) {
        for (int m = 0;m<2; m++) {
          float4 t = mult4(
              vload4(A, K*i/2 + k + K*l/4),
              vload4(B, K*j/2 + k + K*m/4)
            );
          vstore4(t, temp, m + 2*N*i + 4*j + 2*l);
          for (int n = 0;n<4; n++) {
            C[m + 2*N*i + 2*j + N*l] +=
              temp[n + 4*m + 8*N*i + 16*j + 8*l];
          }
        }
      }
    }
  }
}
```

Mapping parallelism to global threads



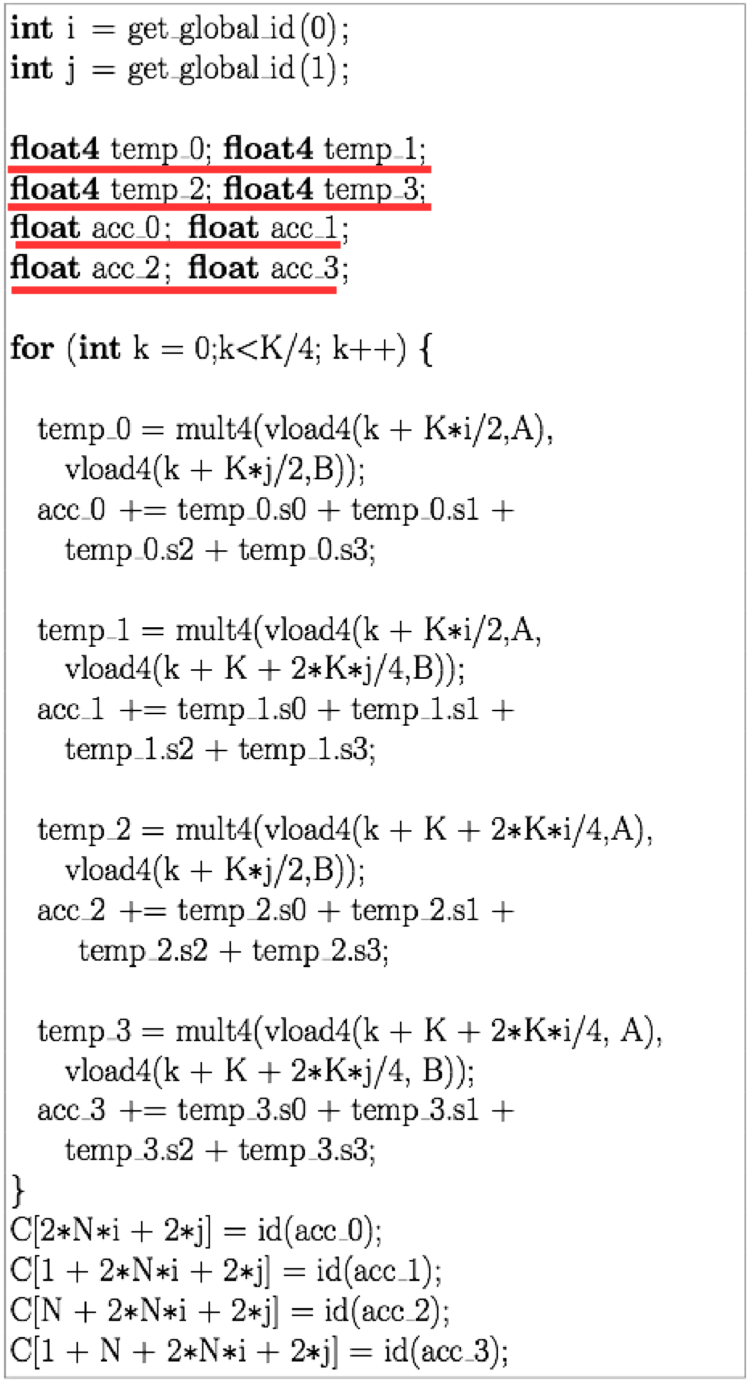
```
λ (A, B) ↦
 A >> split(m) >> mapGlb0(λ nRowsOfA ↦
  B >> split(n) >> mapGlb1(λ mColsOfB ↦
     zip( transpose(nRowsOfA) >> split(k),
          transpose(mColsOfB) >> split(k) ) >>
     reduceSeq(init = make2DArray(n,m, 0.0f),
      λ (accTile, (tileOfA, tileOfB)) ↦
       zip(accTile, transpose(tileOfA)) >>
       mapSeq(λ (accRow, rowOfTileOfA) ↦
        zip(accRow, transpose(tileOfB)) >>
        mapSeq(λ (acc, colOfTileOfB) ↦
        zip(rowOfTileOfA >> asVector(k),
            colOfTileOfB >> asVector(k)) >>
          mapSeq(mult4) >> asScalar >>
          reduceSeq(acc, add)
        ) >> join
       )
     ) >> transpose() >>
     map(transpose) >> transpose
  ) >> join >> transpose
) >> join
```



```
int i = get_global_id(0);
int j = get_global_id(1);
for (int k = 0;k<K/4; k++) {
  for (int l = 0;l<2; l++) {
    for (int m = 0;m<2; m++) {
      float4 t = mult4(
          vload4(A, K*i/2 + k + K*l/4),
          vload4(B, K*j/2 + k + K*m/4)
        );
      vstore4(t, temp, m + 2*N*i + 4*j + 2*l);
      for (int n = 0;n<4; n++) {
        C[m + 2*N*i + 2*j + N*l] +=
          temp[n + 4*m + 8*N*i + 16*j + 8*l];
      }
    }
  }
}
```

Accumulating in private memory

```
λ (A, B) ↦
 A >> split(m) >> mapGlb0(λ nRowsOfA ↦
  B >> split(n) >> mapGlb1(λ mColsOfB ↦
    zip( transpose(nRowsOfA) >> split(k),
         transpose(mColsOfB) >> split(k) ) >>
    reduceSeq(init = make2DArray(n,m, 0.0f) >>
              toPrivate(mapSeq(mapSeq(id))),
     λ (accTile, (tileOfA, tileOfB)) ↦
      zip(accTile, transpose(tileOfA)) >>
      mapSeq(λ (accRow, rowOfTileOfA) ↦
       zip(accRow, transpose(tileOfB)) >>
       mapSeq(λ (acc, colOfTileOfB) ↦
       zip(rowOfTileOfA >> asVector(k),
           colOfTileOfB >> asVector(k)) >>
         mapSeq(mult4) >> asScalar >>
         reduceSeq(acc, add)
       ) >> join
      )
    ) >> toGlobal(mapSeq(mapSeq(mapSeq(id)))
    >> transpose() >>
    map(transpose) >> transpose
  ) >> join >> transpose
) >> join
```



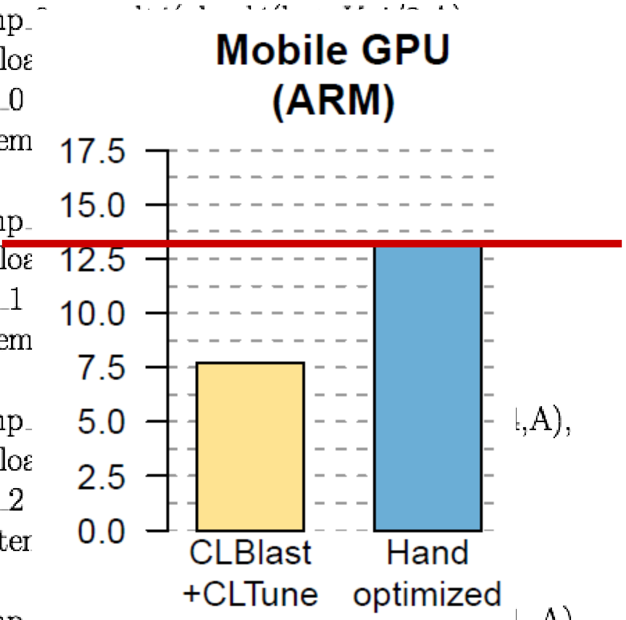
```
int i = get_global_id(0);
int j = get_global_id(1);

float4 temp_0; float4 temp_1;
float4 temp_2; float4 temp_3;
float acc_0;  float acc_1;
float acc_2;  float acc_3;

for (int k = 0;k<K/4; k++) {

  temp_0 = mult4(vload4(k + K*i/2,A),
    vload4(k + K*j/2,B));
  acc_0 += temp_0.s0 + temp_0.s1 +
    temp_0.s2 + temp_0.s3;

  temp_1 = mult4(vload4(k + K*i/2,A,
    vload4(k + K + 2*K*j/4,B));
  acc_1 += temp_1.s0 + temp_1.s1 +
    temp_1.s2 + temp_1.s3;

  temp_2 = mult4(vload4(k + K + 2*K*i/4,A),
    vload4(k + K*j/2,B));
  acc_2 += temp_2.s0 + temp_2.s1 +
     temp_2.s2 + temp_2.s3;

  temp_3 = mult4(vload4(k + K + 2*K*i/4, A),
    vload4(k + K + 2*K*j/4, B));
  acc_3 += temp_3.s0 + temp_3.s1 +
    temp_3.s2 + temp_3.s3;
}
C[2*N*i + 2*j] = id(acc_0);
C[1 + 2*N*i + 2*j] = id(acc_1);
C[N + 2*N*i + 2*j] = id(acc_2);
C[1 + N + 2*N*i + 2*j] = id(acc_3);
```

Result: high performance

```
λ (A, B) ↦
 A >> split(m) >> mapGlb0(λ nRowsOfA ↦
  B >> split(n) >> mapGlb1(λ mColsOfB ↦
    zip( transpose(nRowsOfA) >> split(k),
         transpose(mColsOfB) >> split(k) ) >>
    reduceSeq(init = make2DArray(n,m, 0.0f) >>
              toPrivate(mapSeq(mapSeq(id))),
     λ (accTile, (tileOfA, tileOfB)) ↦
      zip(accTile, transpose(tileOfA)) >>
      mapSeq(λ (accRow, rowOfTileOfA) ↦
       zip(accRow, transpose(tileOfB)) >>
       mapSeq(λ (acc, colOfTileOfB) ↦
       zip(rowOfTileOfA >> asVector(k),
           colOfTileOfB >> asVector(k)) >>
         mapSeq(mult4) >> asScalar >>
         reduceSeq(acc, add)
       ) >> join
      )
    ) >> toGlobal(mapSeq(mapSeq(mapSeq(id)))
    >> transpose() >>
    map(transpose) >> transpose
  ) >> join >> transpose
) >> join
```

→

```
int i = get_global_id(0);
int j = get_global_id(1);

float4 temp_0; float4 temp_1;
float4 temp_2; float4 temp_3;
float acc_0; float acc_1;
float acc_2; float acc_3;

for (int k = 0;k<K/4; k++) {

  temp_
    vloa
  acc_0
    tem

  temp_
    vloa
  acc_1
    tem

  temp_                                   ,A),
    vloa
  acc_2
    ter

  temp_                                   , A),
    vloa
  acc_3 += temp_3.s0 + temp_3.s1 +
    temp_3.s2 + temp_3.s3;
}
C[2*N*i + 2*j] = id(acc_0);
C[1 + 2*N*i + 2*j] = id(acc_1);
C[N + 2*N*i + 2*j] = id(acc_2);
C[1 + N + 2*N*i + 2*j] = id(acc_3);
```



Now just need to search the space

…

100,000 runs later

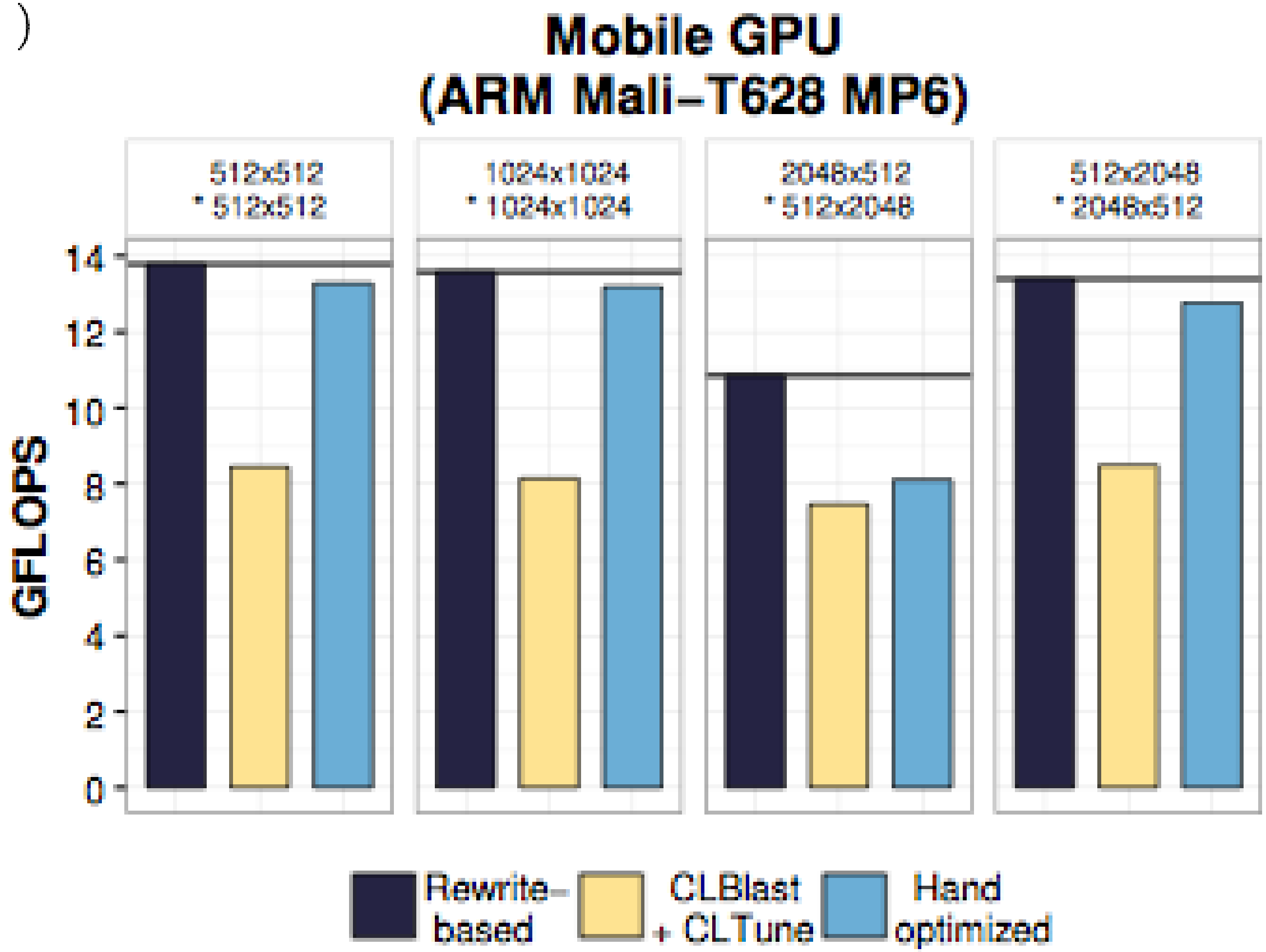
…

Performance Portability Achieved

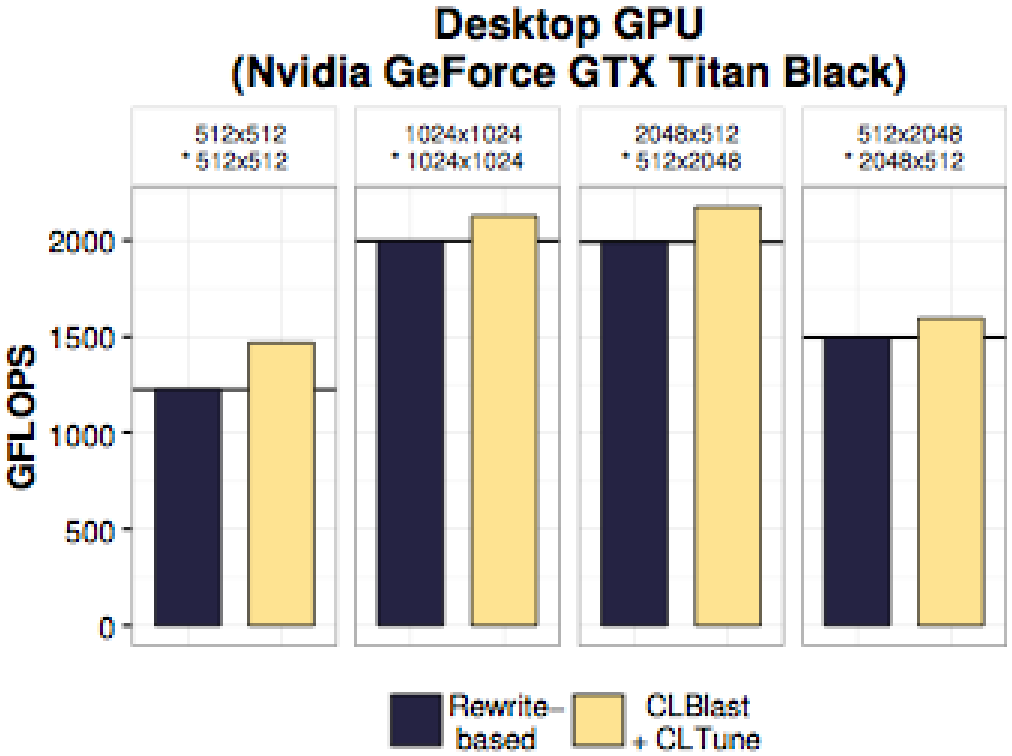
Compiler input:

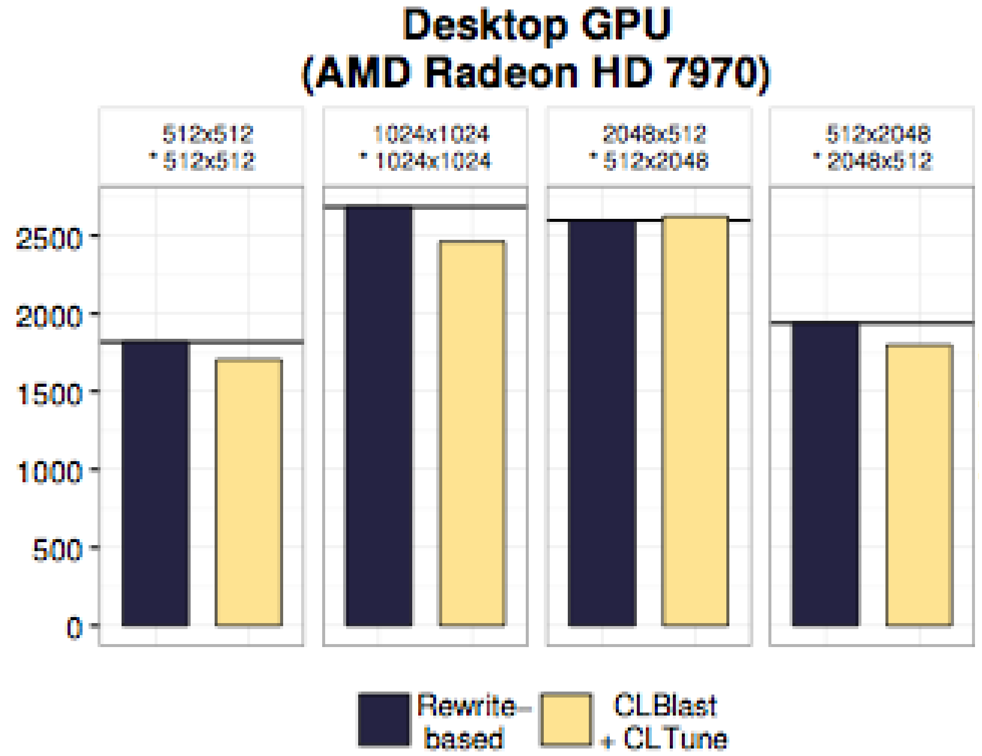
```
A >> map(λ rowOfA ↦
 B >> map(λ colOfB ↦
   zip(rowOfA, colOfB) >>
   map(mult) >> reduce(0.0f,add)
 )
)
```

Search result:



Desktop GPUs

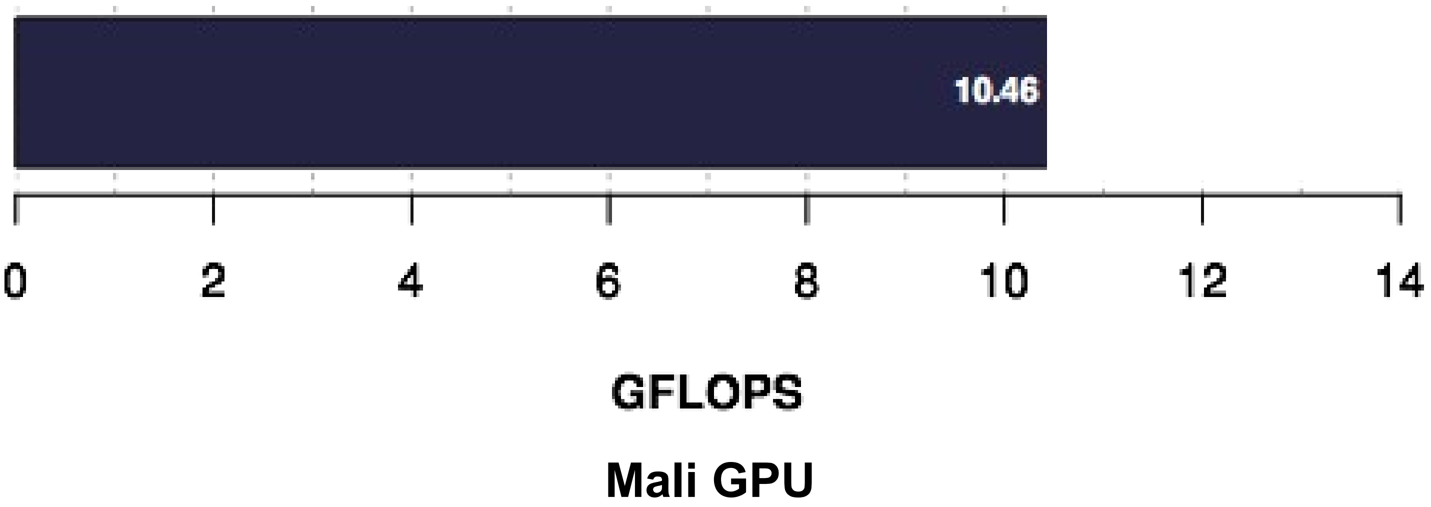




Easily Extensible

- New rules can be added
- E.g. dot built-in

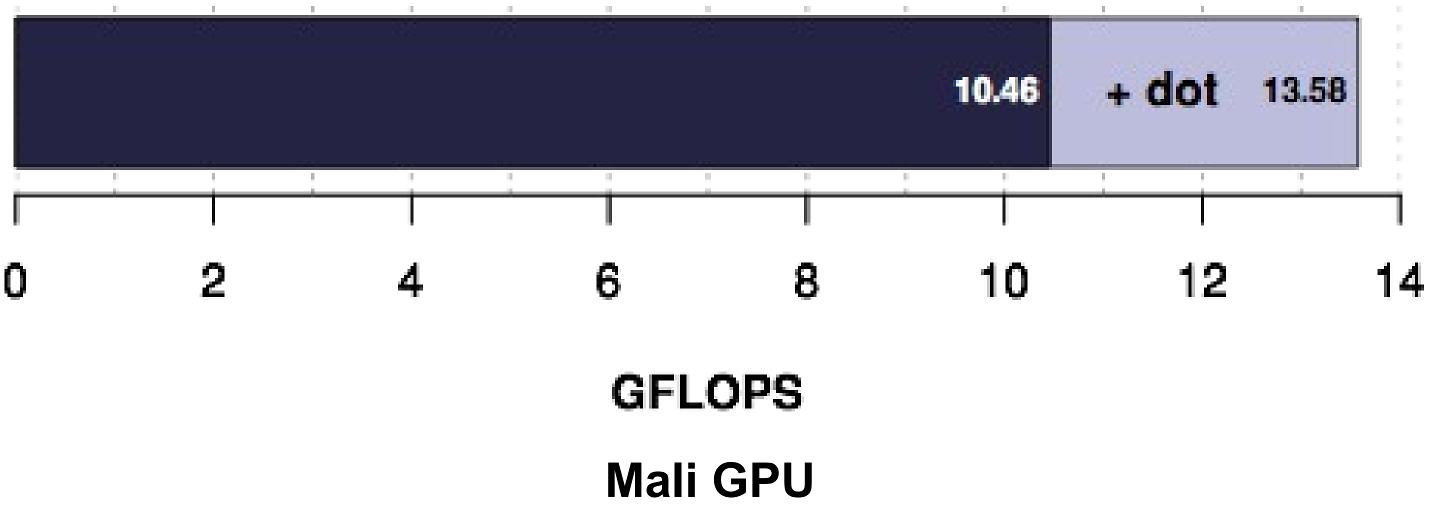
```
reduceSeq(z, add4) o mapSeq(mult4) o zip(x,y)
=>
dot(x, y)
```



Easily Extensible

- New rules can be added
- E.g. dot built-in

```
reduceSeq(z, add4) o mapSeq(mult4) o zip(x,y)
=>
dot(x, y)
```



Part III

Efficient Code Generation

Dot-product example

```
dot(x, y) = reduce(+, 0, map(*, zip(x, y)))
```

Dot-product example

```
dot(x, y) = reduce(+, 0, map(*, zip(x, y)))
```

after rewrite:

```
(join ∘ mapWrg^0(
  join ∘ toGlobal(mapLcl^0(mapSeq(id))) ∘ split^1 ∘
  iterate^6( join ∘
             mapLcl^0( toLocal(mapSeq(id)) ∘
                       reduceSeq(add, 0) ) ∘
             split^2 ) ∘
  join ∘ mapLcl^0( toLocal(mapSeq(id)) ∘
       reduceSeq(multAndSumUp, 0) ) ∘ split^2
) ∘ split^128)( zip(x, y) )
```

Dot-product example

```
dot(x, y) = reduce(+, 0, map(*, zip(x, y)))
```

after rewrite:

```
(join ∘ mapWrg^0(
  join ∘ toGlobal(mapLcl^0(mapSeq(id))) ∘ split^1 ∘
  iterate^6( join ∘
             mapLcl^0( toLocal(mapSeq(id)) ∘
                       reduceSeq(add, 0) ) ∘
             split^2 ) ∘
  join ∘ mapLcl^0( toLocal(mapSeq(id)) ∘
       reduceSeq(multAndSumUp, 0) ) ∘ split^2
) ∘ split^128)( zip(x, y) )
```

copy to local memory

Dot-product example

```
dot(x, y) = reduce(+, 0, map(*, zip(x, y)))
```

after rewrite:

```
(join ∘ mapWrg^0(
  join ∘ toGlobal(mapLcl^0(mapSeq(id))) ∘ split^1 ∘
  iterate^6( join ∘
             mapLcl^0( toLocal(mapSeq(id)) ∘
                       reduceSeq(add, 0) ) ∘
             split^2 ) ∘
  join ∘ mapLcl^0( toLocal(mapSeq(id)) ∘
       reduceSeq(multAndSumUp, 0) ) ∘ split^2
) ∘ split^128)( zip(x, y) )
```

iterative reduction in local memory

Dot-product example

```
dot(x, y) = reduce(+, 0, map(*, zip(x, y)))
```

after rewrite:

```
(join ∘ mapWrg$^{0}$(
  join ∘ toGlobal(mapLcl$^{0}$(mapSeq(id))) ∘ split$^{1}$ ∘
  iterate$^{6}$( join ∘
             mapLcl$^{0}$( toLocal(mapSeq(id)) ∘
                      reduceSeq(add, 0) ) ∘
             split$^{2}$ ) ∘
  join ∘ mapLcl$^{0}$( toLocal(mapSeq(id)) ∘
       reduceSeq(multAndSumUp, 0) ) ∘ split$^{2}$
) ∘ split$^{128}$)( zip(x, y) )
```

write back to global mem.

How to generate efficient OpenCL code?

$$\begin{array}{l}
\texttt{(join} \circ \texttt{mapWrg}^{0}\texttt{(} \\
\quad \texttt{join} \circ \texttt{toGlobal(mapLcl}^{0}\texttt{(mapSeq(id)))} \circ \texttt{split}^{1} \circ \\
\quad \texttt{iterate}^{6}\texttt{(join} \circ \\
\qquad\qquad\quad \texttt{mapLcl}^{0}\texttt{(toLocal(mapSeq(id))} \circ \\
\qquad\qquad\qquad\qquad\quad \texttt{reduceSeq(add, 0))} \circ \\
\qquad\qquad\quad \texttt{split}^{2}\texttt{)} \circ \\
\quad \texttt{join} \circ \texttt{mapLcl}^{0}\texttt{(toLocal(mapSeq(id))} \circ \\
\qquad\quad \texttt{reduceSeq(multAndSumUp, 0))} \circ \texttt{split}^{2} \\
\texttt{)} \circ \texttt{split}^{128}\texttt{)(zip(x, y))}
\end{array}$$

→ Pattern based code generator

map-global (f,input)

```
for (uint i=get_global_id;
     i<n;
     i+= get_global_size) {
   output[i] = f(input[i]);
}
```

...

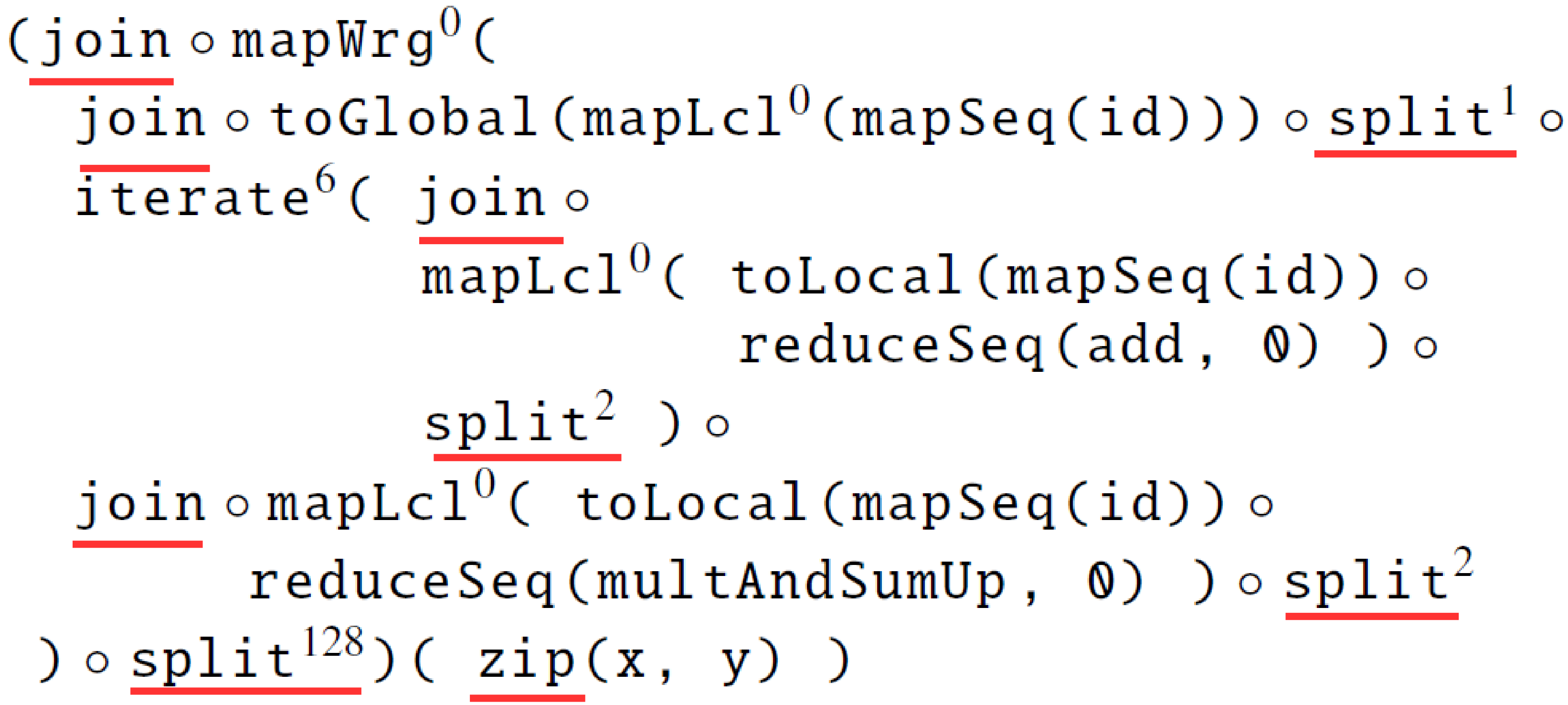
map-sequential (f,input)

```
for (uint i=0; i<n; i++) {
   output[i] = f(input[i]);
}
```

reduce-sequential (f,z,input)

```
T acc = z;
for (uint i=0; i<n; i++) {
   acc = f(acc, input[i]);
}
```

What about split, zip, join, ... ?



```
(join ∘ mapWrg^0(
   join ∘ toGlobal(mapLcl^0(mapSeq(id))) ∘ split^1 ∘
   iterate^6( join ∘
                  mapLcl^0( toLocal(mapSeq(id)) ∘
                               reduceSeq(add, 0) ) ∘
                  split^2 ) ∘
   join ∘ mapLcl^0( toLocal(mapSeq(id)) ∘
          reduceSeq(multAndSumUp, 0) ) ∘ split^2
 ) ∘ split^128)( zip(x, y) )
```

- Need to avoid temporary results
- split, zip, ... merely just change data layout
  - → “lazy evaluation”

dot-product (a tiny bit of it):

$$\begin{array}{l} (\mathtt{join} \circ \mathtt{mapWrg}^{0}(\ \ldots \\ \quad \mathtt{join} \circ \mathtt{mapLcl}^{0}(\ \ldots \\ \quad\quad \mathtt{reduceSeq}(\lambda(\mathtt{a},\mathtt{xy}) \mapsto\ \mathtt{a} + (\boxed{\mathtt{xy}_{0}} * \mathtt{xy}_{1}), 0)) \circ \mathtt{split}^{2} \\ \ ) \circ \mathtt{split}^{128})(\ \mathtt{zip}(\mathtt{x},\ \mathtt{y})\ ) \end{array}$$

desired OpenCL output:

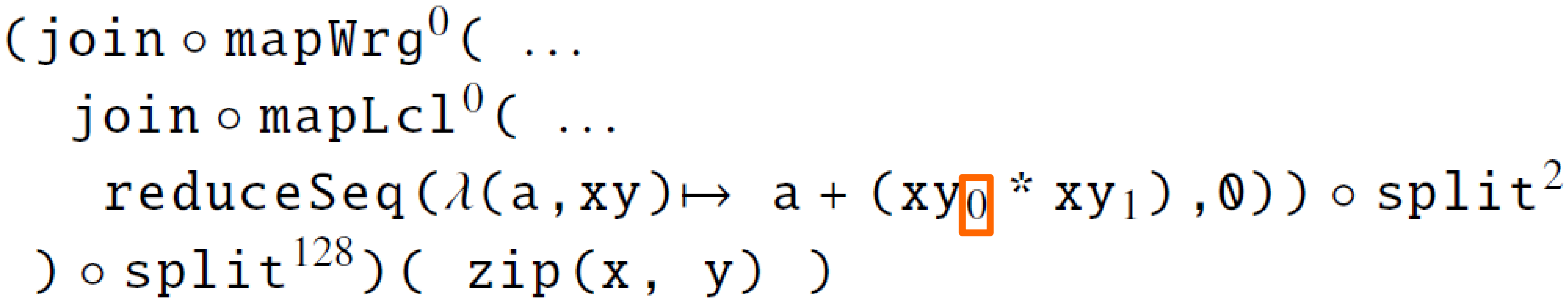
```
int wg_id = get_group_id(0) ;
int l_id = get_local_id(0) ;
...
acc = 0.0f;
for (int i = 0 ; i < 2 ; I += 1) {
   acc = acc +
            x [ 2 * l_id + 128 * wg_id + I ] *
            y [ 2 * l_id + 128 * wg_id + I ] ;
}
```

Construct a representation of the effects of data layout functions:

Views

Consumes the views to generate correct array indices

View Construction



$$\begin{array}{l} (\texttt{join} \circ \texttt{mapWrg}^{0}(\ \ldots \\ \quad \texttt{join} \circ \texttt{mapLcl}^{0}(\ \ldots \\ \qquad \texttt{reduceSeq}(\lambda(\texttt{a},\texttt{xy}) \mapsto \ \texttt{a} + (\texttt{xy}_{\boxed{0}} * \texttt{xy}_{1}),\texttt{0})) \circ \texttt{split}^{2} \\ \ ) \circ \texttt{split}^{128})(\ \texttt{zip}(\texttt{x},\ \texttt{y})\ ) \end{array}$$

View Construction

(join $\circ$ mapWrg^{0}(...

join $\circ$ mapLcl^{0}(...

reduceSeq(λ(a,xy)$\mapsto$ a + (xy$_{\boxed{0}}$ * xy$_{1}$),0)) $\circ$ split^{2}

) $\circ$ split^{128})(zip(x, y))


TupleAccessView(0)

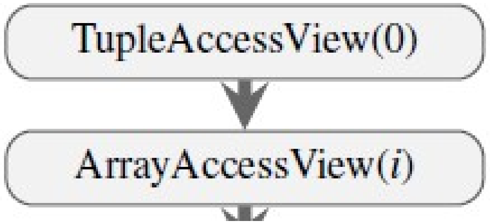
View Construction

$(\texttt{join} \circ \texttt{mapWrg}^{0}(\ \dots$

$\quad\texttt{join} \circ \texttt{mapLcl}^{0}(\ \dots$

$\quad\boxed{\texttt{reduceSeq}}(\lambda(\texttt{a},\texttt{xy}) \mapsto\ \texttt{a} + (\texttt{xy}_0 * \texttt{xy}_1),\texttt{0})) \circ \texttt{split}^{2}$

$) \circ \texttt{split}^{128})(\ \texttt{zip}(\texttt{x},\ \texttt{y})\ )$



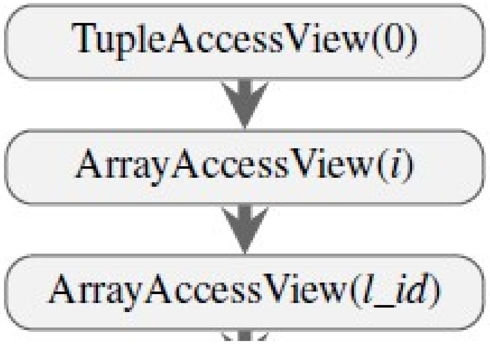
View Construction

$(\texttt{join} \circ \texttt{mapWrg}^{0}(\ \ldots$

$\quad \texttt{join} \circ \boxed{\texttt{mapLcl}^{0}}(\ \ldots$

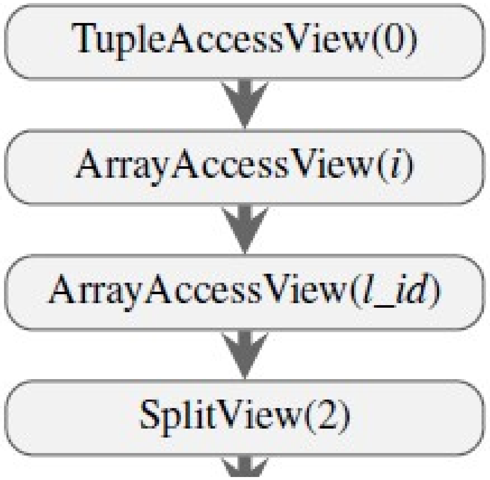
$\quad\quad \texttt{reduceSeq}(\lambda(\texttt{a},\texttt{xy}) \mapsto \texttt{a} + (\texttt{xy}_0 * \texttt{xy}_1),0)) \circ \texttt{split}^{2}$

$\ ) \circ \texttt{split}^{128})(\ \texttt{zip}(\texttt{x},\ \texttt{y})\ )$



View Construction

```
(join ∘ mapWrg^0( ...
  join ∘ mapLcl^0( ...
   reduceSeq(λ(a,xy)↦ a + (xy_0 * xy_1),0)) ∘ split^2
 ) ∘ split^128)( zip(x, y) )
```



TupleAccessView(0)

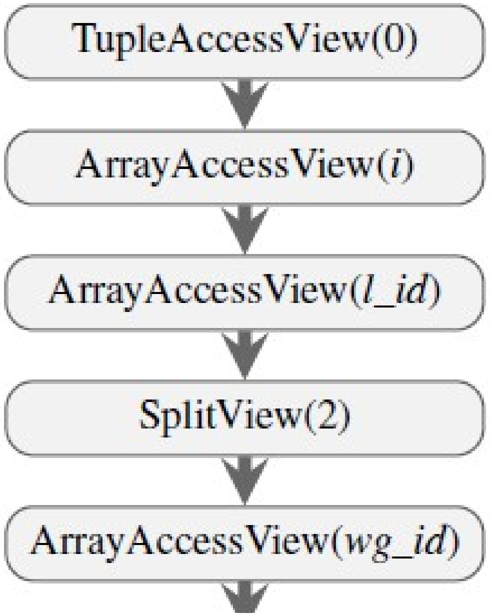
ArrayAccessView(*i*)

ArrayAccessView(*l_id*)

SplitView(2)

View Construction

$$\begin{array}{l}
\texttt{(join} \circ \boxed{\texttt{mapWrg}^{0}}\texttt{(...} \\
\quad \texttt{join} \circ \texttt{mapLcl}^{0}\texttt{(...} \\
\quad\; \texttt{reduceSeq(}\lambda\texttt{(a,xy)}\mapsto \texttt{ a + (xy}_{0} \texttt{ * xy}_{1}\texttt{),0))} \circ \texttt{split}^{2} \\
\;\texttt{)} \circ \texttt{split}^{128}\texttt{)(zip(x, y))}
\end{array}$$


TupleAccessView(0)

ArrayAccessView(*i*)

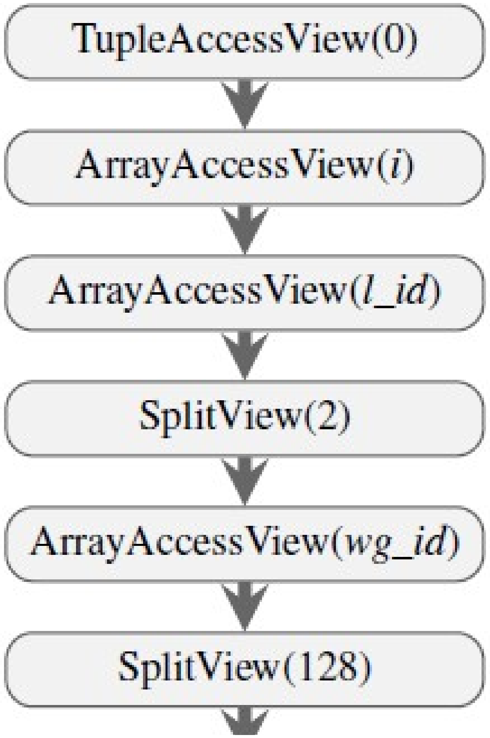
ArrayAccessView(*l_id*)

SplitView(2)

ArrayAccessView(*wg_id*)

View Construction

$$
\begin{array}{l}
(\texttt{join} \circ \texttt{mapWrg}^{0}(\ \ldots \\
\quad \texttt{join} \circ \texttt{mapLcl}^{0}(\ \ldots \\
\quad\quad \texttt{reduceSeq}(\lambda(\texttt{a},\texttt{xy}) \mapsto\ \texttt{a} + (\texttt{xy}_{0} * \texttt{xy}_{1}),\texttt{0})) \circ \texttt{split}^{2} \\
\ ) \circ \boxed{\texttt{split}^{128}})(\ \texttt{zip}(\texttt{x},\ \texttt{y})\ )
\end{array}
$$



TupleAccessView(0)

ArrayAccessView(*i*)

ArrayAccessView(*l_id*)

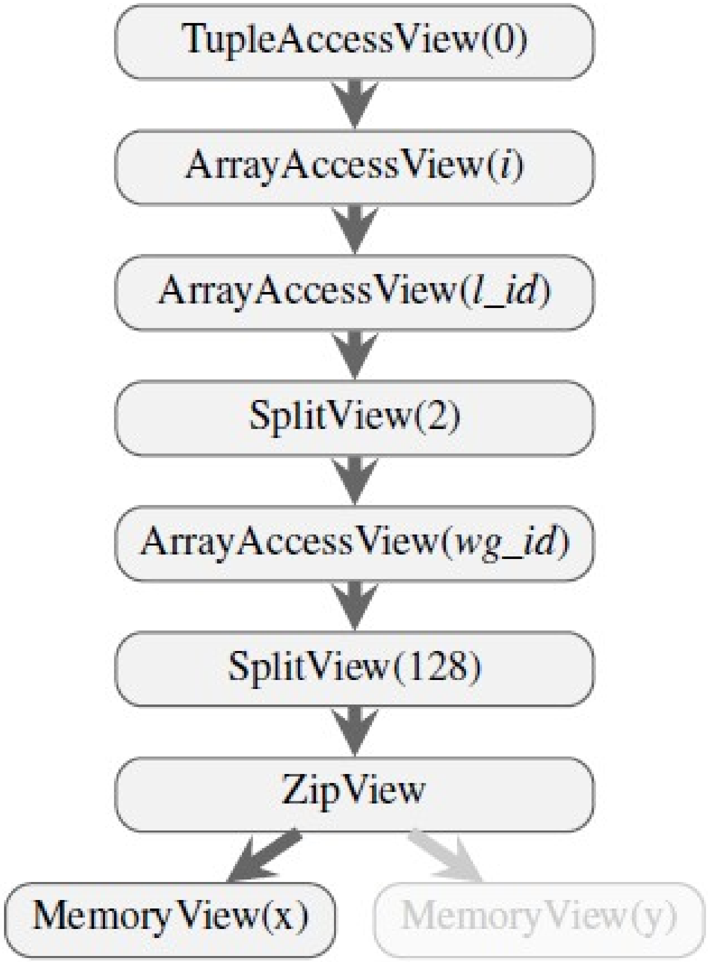
SplitView(2)

ArrayAccessView(*wg_id*)

SplitView(128)

View Construction

$$\begin{array}{l}
\texttt{(join} \circ \texttt{mapWrg}^{0}\texttt{(...} \\
\quad \texttt{join} \circ \texttt{mapLcl}^{0}\texttt{(...} \\
\quad\quad \texttt{reduceSeq(}\lambda\texttt{(a,xy)}\mapsto\ \texttt{a} + \texttt{(xy}_{0} * \texttt{xy}_{1}\texttt{),0))} \circ \texttt{split}^{2} \\
\ \texttt{)} \circ \texttt{split}^{128}\texttt{)(}\ \boxed{\texttt{zip(x, y)}}\ \texttt{)}
\end{array}$$



TupleAccessView(0)

ArrayAccessView(*i*)

ArrayAccessView(*l_id*)

SplitView(2)

ArrayAccessView(*wg_id*)

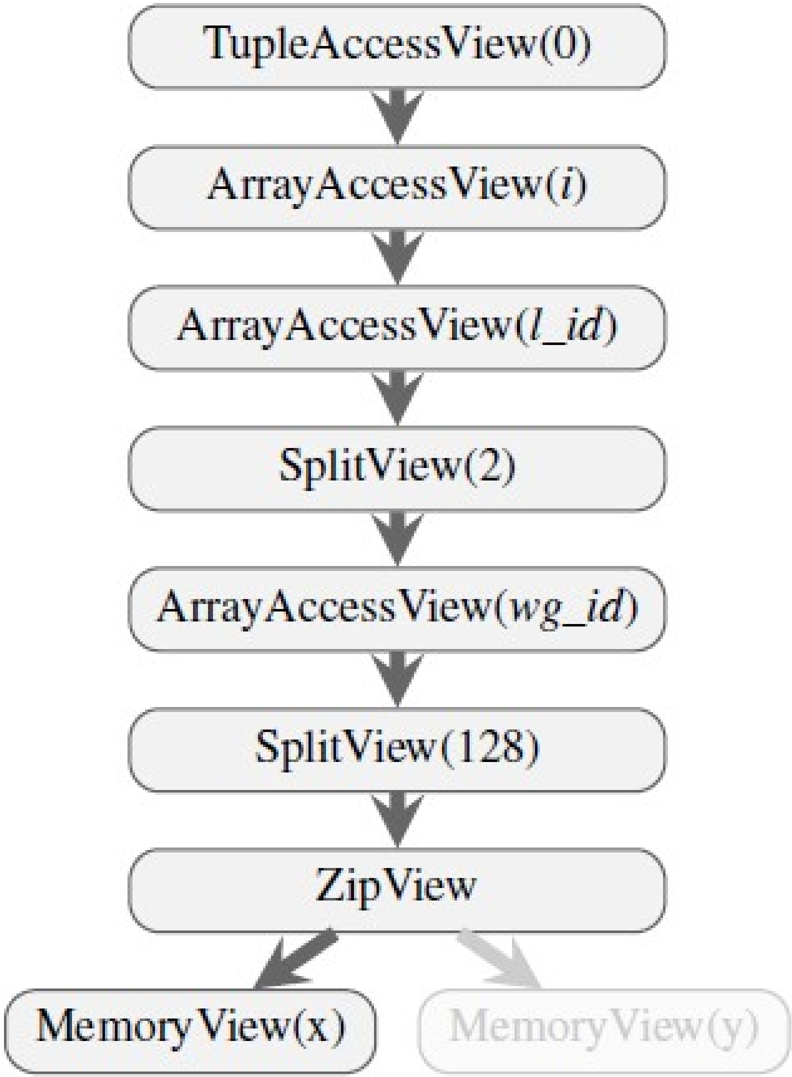
SplitView(128)

ZipView

MemoryView(x)

MemoryView(y)

View Consumption



TupleAccessView(0)

ArrayAccessView(*i*)

ArrayAccessView(*l_id*)

SplitView(2)

ArrayAccessView(*wg_id*)

SplitView(128)

ZipView

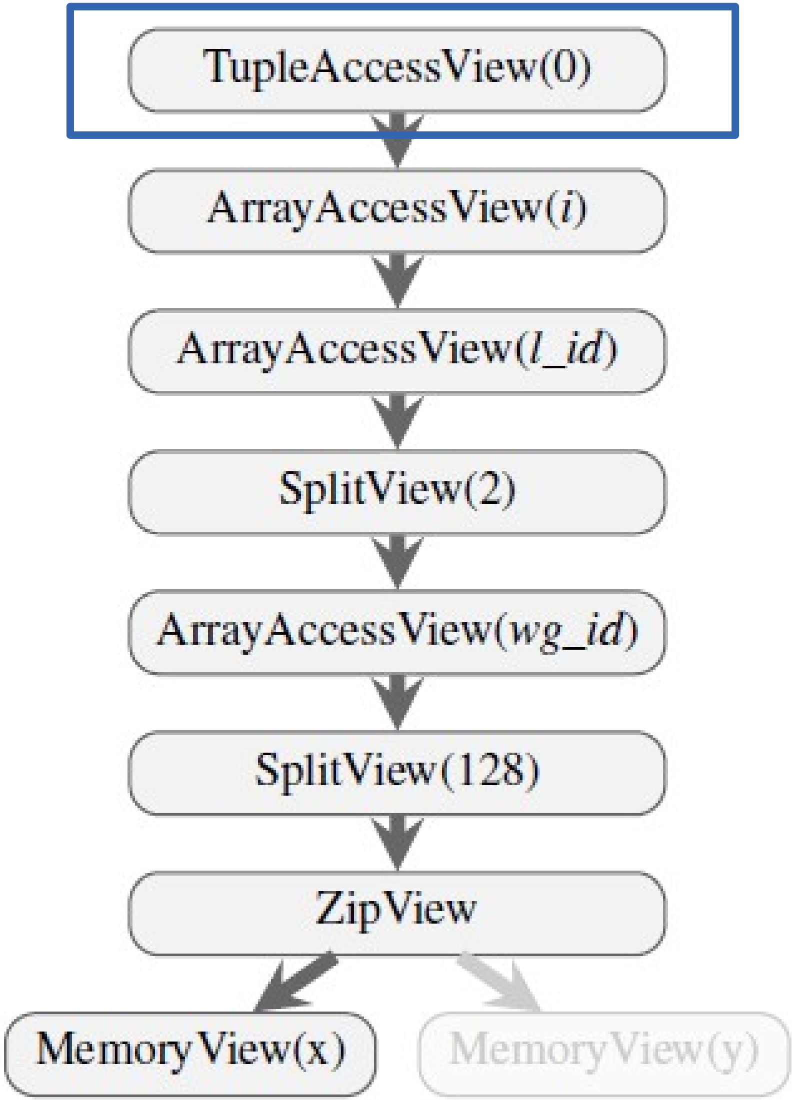
MemoryView(x)

MemoryView(y)

Array Stack

Tuple Stack

View Consumption



TupleAccessView(0)

ArrayAccessView(*i*)

ArrayAccessView(*l_id*)

SplitView(2)

ArrayAccessView(*wg_id*)

SplitView(128)

ZipView

MemoryView(x)

MemoryView(y)

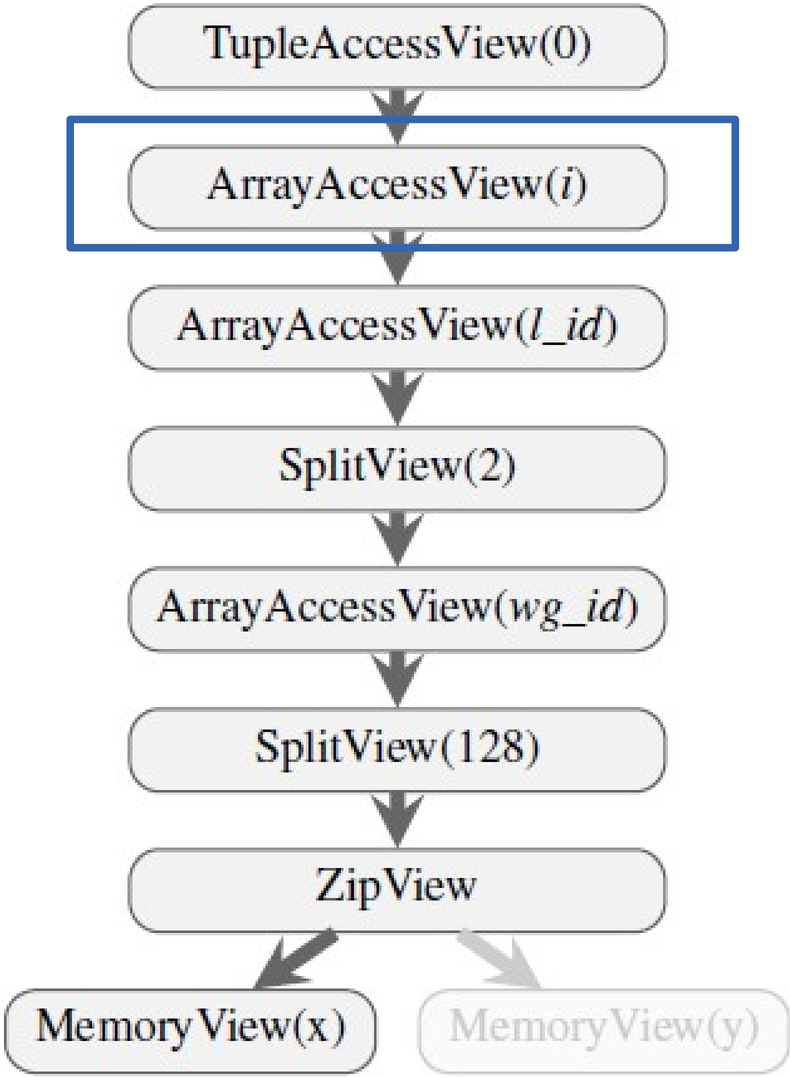
Array Stack

[]

Tuple Stack

[0]

View Consumption



TupleAccessView(0)

ArrayAccessView(*i*)

ArrayAccessView(*l_id*)

SplitView(2)

ArrayAccessView(*wg_id*)

SplitView(128)

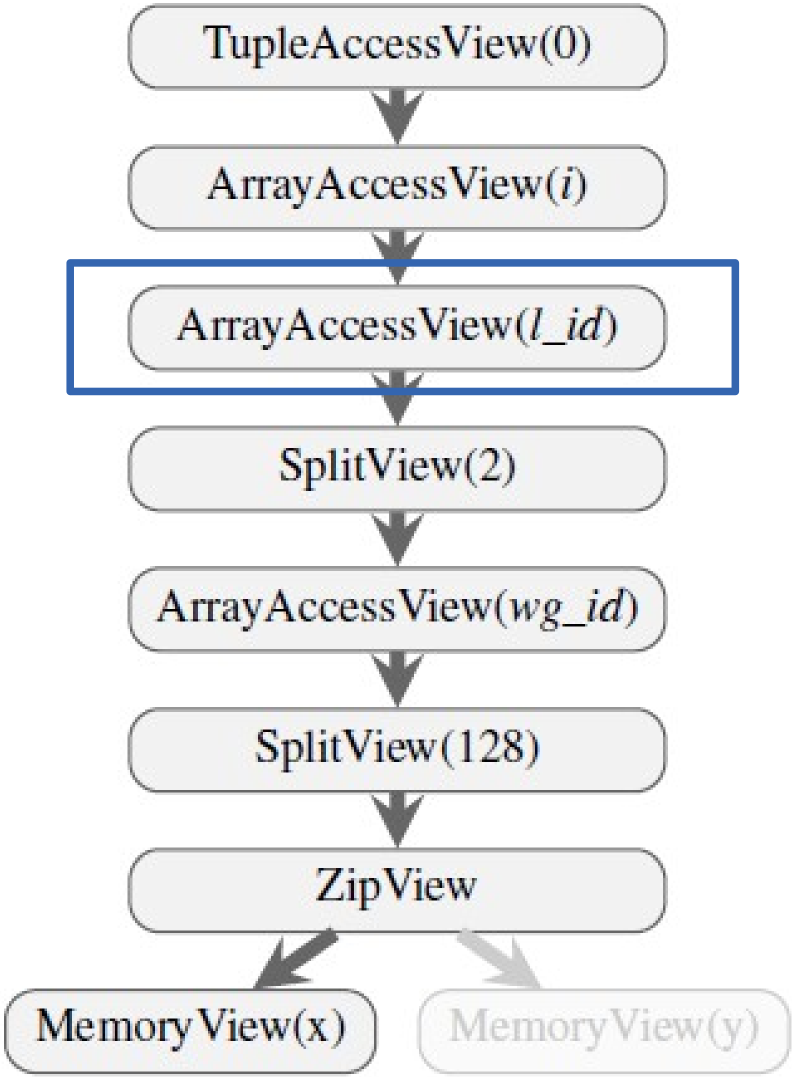
ZipView

MemoryView(x)

MemoryView(y)

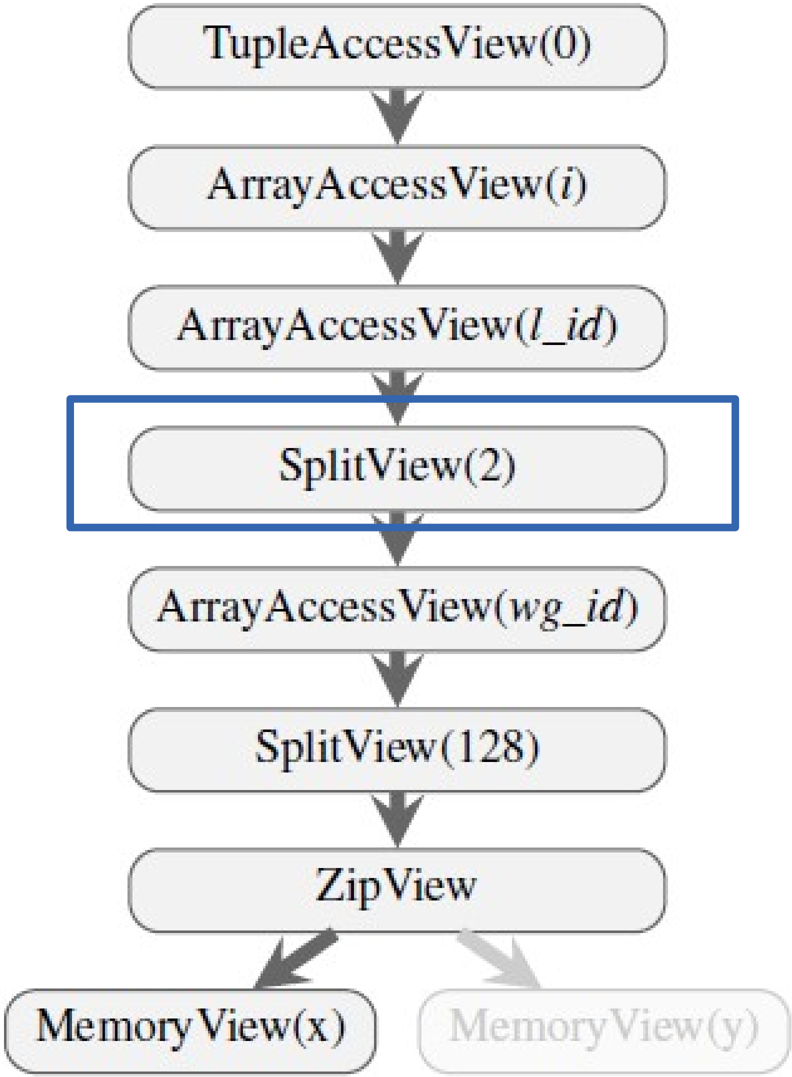
Array Stack	Tuple Stack
[]	[0]
[*i*]	[0]

View Consumption



Array Stack	Tuple Stack
[]	[0]
[i]	[0]
[l_id, i]	[0]

View Consumption



TupleAccessView(0)

ArrayAccessView(i)

ArrayAccessView(l_id)

SplitView(2)

ArrayAccessView(wg_id)

SplitView(128)

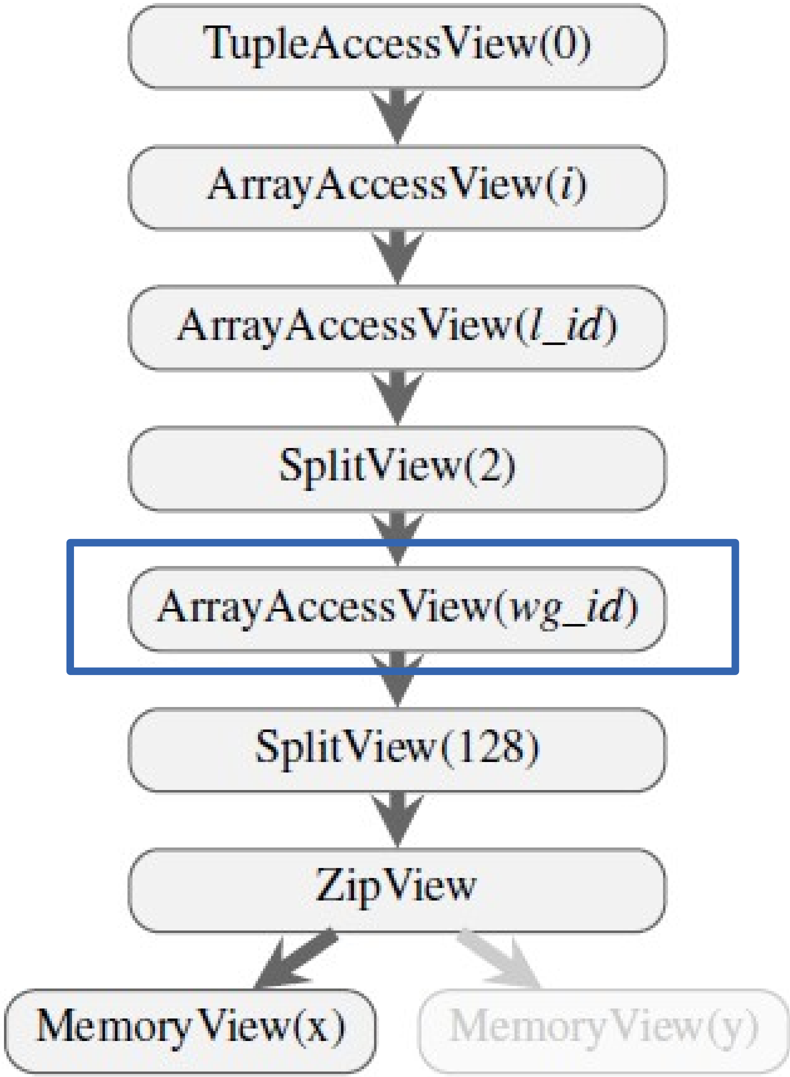
ZipView

MemoryView(x)

MemoryView(y)

Array Stack	Tuple Stack
[]	[0]
$[i]$	[0]
$[l\_id, i]$	[0]
$((2 * l\_id) + i]$	[0]

View Consumption



TupleAccessView(0)

ArrayAccessView(i)

ArrayAccessView(l_id)

SplitView(2)

ArrayAccessView(wg_id)

SplitView(128)

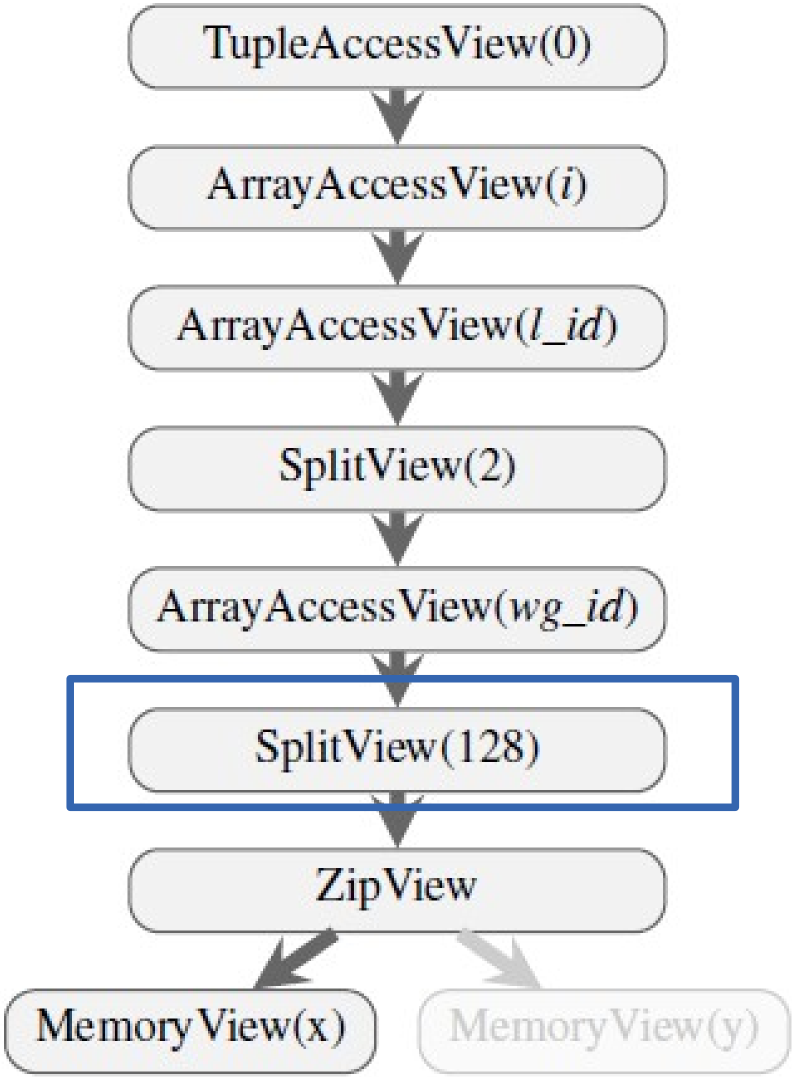
ZipView

MemoryView(x)

MemoryView(y)

Array Stack	Tuple Stack
[]	[0]
[i]	[0]
[l_id, i]	[0]
(($2 * l\_id$) + i]	[0]
[wg_id, ($2 * l\_id$) + i]	[0]

View Consumption



TupleAccessView(0)

ArrayAccessView(*i*)

ArrayAccessView(*l_id*)

SplitView(2)

ArrayAccessView(*wg_id*)

SplitView(128)

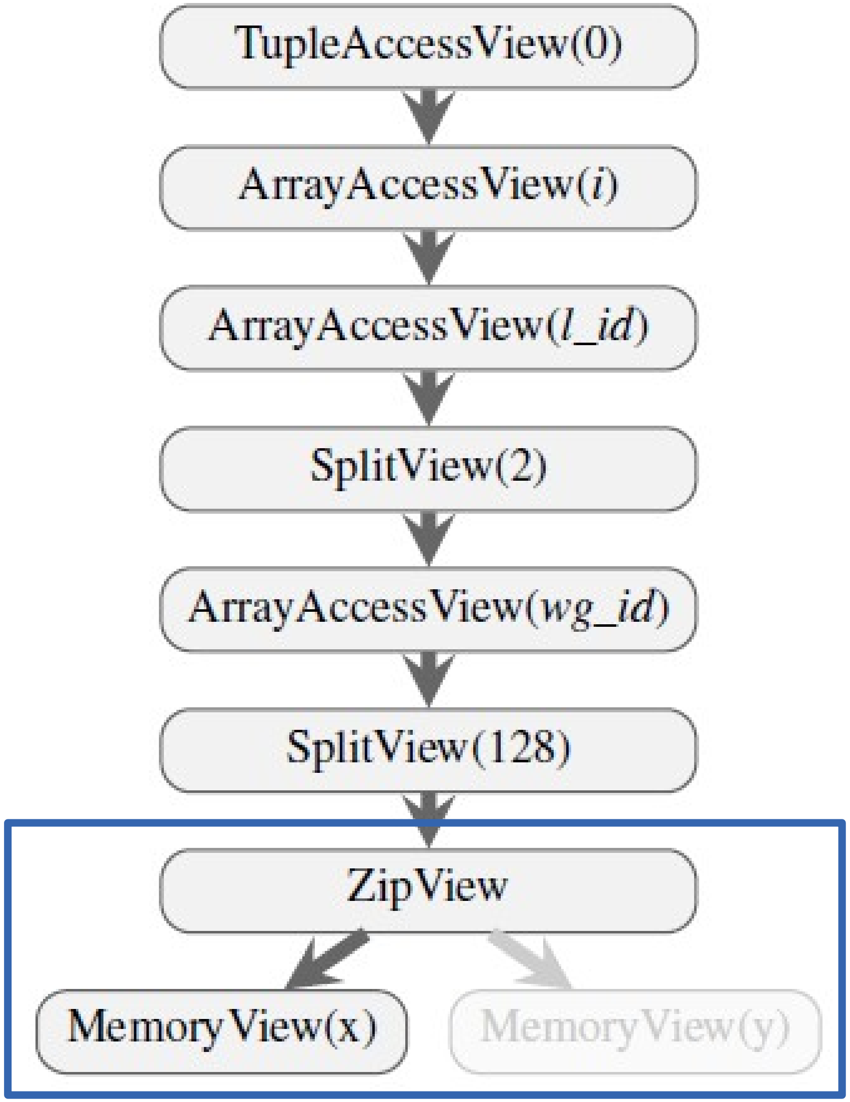
ZipView

MemoryView(x)

MemoryView(y)

Array Stack	Tuple Stack
$[]$	$[0]$
$[i]$	$[0]$
$[l\_id, i]$	$[0]$
$((2 * l\_id) + i]$	$[0]$
$[wg\_id, (2 * l\_id) + i]$	$[0]$
$[(2 * l\_id) + (128 * wg\_id) + i]$	$[0]$

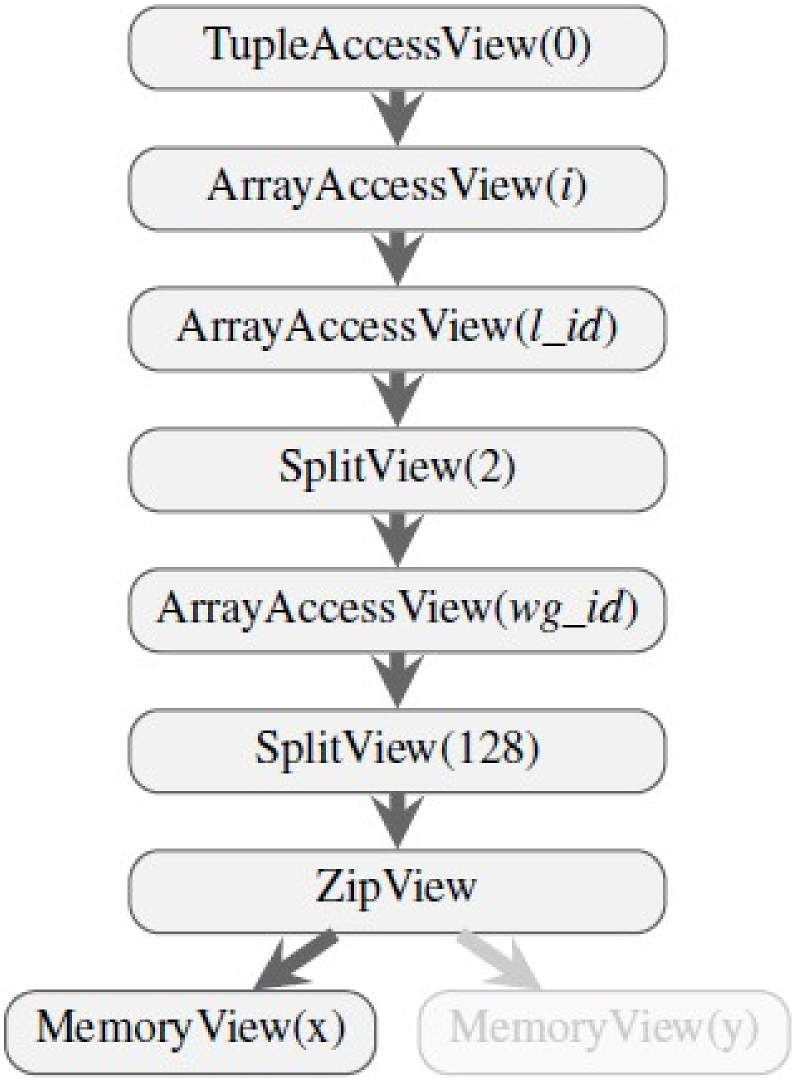
View Consumption

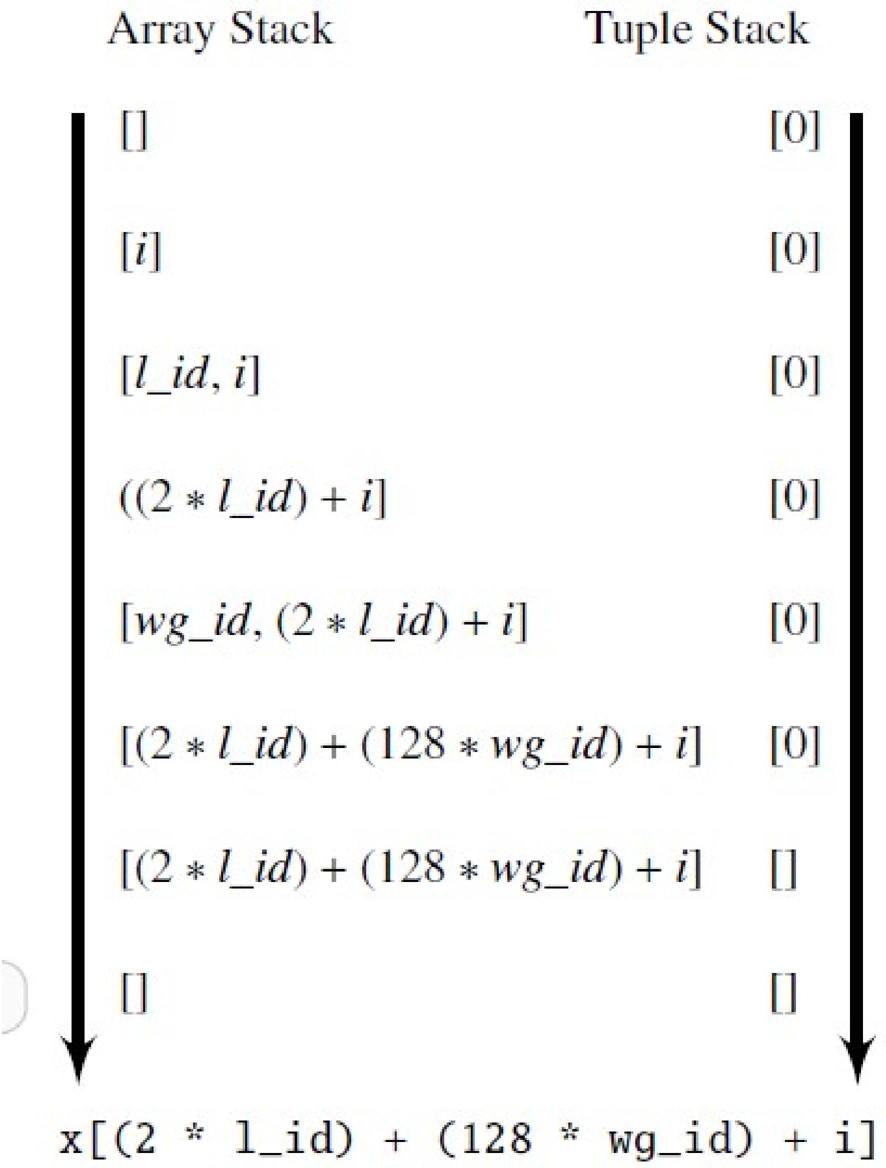


Array Stack	Tuple Stack
$[]$	$[0]$
$[i]$	$[0]$
$[l\_id, i]$	$[0]$
$((2 * l\_id) + i]$	$[0]$
$[wg\_id, (2 * l\_id) + i]$	$[0]$
$[(2 * l\_id) + (128 * wg\_id) + i]$	$[0]$
$[(2 * l\_id) + (128 * wg\_id) + i]$	$[]$

x

View Consumption





Array Stack	Tuple Stack
$[]$	$[0]$
$[i]$	$[0]$
$[l\_id, i]$	$[0]$
$((2 * l\_id) + i]$	$[0]$
$[wg\_id, (2 * l\_id) + i]$	$[0]$
$[(2 * l\_id) + (128 * wg\_id) + i]$	$[0]$
$[(2 * l\_id) + (128 * wg\_id) + i]$	$[]$
$[]$	$[]$

`x[(2 * l_id) + (128 * wg_id) + i]`

dot-product (a tiny bit of it):

$$
\begin{array}{l}
\mathtt{(join} \circ \mathtt{mapWrg}^{0}\mathtt{(}\ \ldots \\
\quad \mathtt{join} \circ \mathtt{mapLcl}^{0}\mathtt{(}\ \ldots \\
\quad\quad \mathtt{reduceSeq(}\lambda\mathtt{(a,xy)}\mapsto\ \mathtt{a} + \mathtt{(}\boxed{\mathtt{xy}_{0}} * \mathtt{xy}_{1}\mathtt{),0))} \circ \mathtt{split}^{2} \\
\ \ \mathtt{)} \circ \mathtt{split}^{128}\mathtt{)(\ zip(x,\ y)\ )}
\end{array}
$$

produced OpenCL output:

```
int wg_id = get_group_id(0) ;
int l_id = get_local_id(0) ;
...
acc = 0.0f;
for (int i = 0 ; i < 2 ; I += 1) {
   acc = acc +
             x [ 2 * l_id + 128 * wg_id + I ] *
             y [ 2 * l_id + 128 * wg_id + I ] ;
}
```

Array indices not so simple

```
((((wg_id*M+l_id)/M)+(((wg_id*M+l_id)%M)*N))/N)*N+(((wg_id*M+l_id)/M)+(((wg_id*M+l_id)%M)*N))%N
```

Array indices not so simple

```
((((wg_id*M+l_id)/M)+(((wg_id*M+l_id)%M)*N))/N)*N+(((wg_id*M+l_id)/M)+(((wg_id*M+l_id)%M)*N))%N
```

Can use well-known algebraic rules

$$x/y = 0, \quad \text{if } x < y \text{ and } y \neq 0 \tag{1}$$

$$(x \times y + z)/y = x + z/y, \quad \text{if } y \neq 0 \tag{2}$$

$$x \,\%\, y = x, \quad \text{if } x < y \text{ and } y \neq 0 \tag{3}$$

$$(x + y) \,\%\, z = (x \,\%\, z + y \,\%\, z) \,\%\, z, \quad \text{if } z \neq 0 \tag{4}$$

$$(x/y) * y + x \,\%\, y = x, \quad \text{if } y \neq 0 \tag{5}$$

$$(x * y) \,\%\, y = 0, \quad \text{if } y \neq 0 \tag{6}$$

Array indices not so simple

```
((((wg_id*M+l_id)/M)+(((wg_id*M+l_id)%M)*N))/N)*N+(((wg_id*M+l_id)/M)+(((wg_id*M+l_id)%M)*N))%N
```

Can use well-known algebraic rules

$$x/y = 0, \quad \text{if } x < y \text{ and } y \neq 0 \tag{1}$$

$$(x \times y + z)/y = x + z/y, \quad \text{if } y \neq 0 \tag{2}$$

$$x \,\%\, y = x, \quad \text{if } x < y \text{ and } y \neq 0 \tag{3}$$

$$(x + y) \,\%\, z = (x \,\%\, z + y \,\%\, z) \,\%\, z, \quad \text{if } z \neq 0 \tag{4}$$

$$(x/y) * y + x \,\%\, y = x, \quad \text{if } y \neq 0 \tag{5}$$

$$(x * y) \,\%\, y = 0, \quad \text{if } y \neq 0 \tag{6}$$

and simplify array index expression

```
((((wg_id*M+l_id)/M)+(((wg_id*M+l_id)%M)*N))/N)*N+(((wg_id*M+l_id)/M)+(((wg_id*M+l_id)%M)*N))%N
```

↓

```
                    l_id          *N+   wg_id
```

Putting it all together

```
dot(x, y) = reduce(+, 0, map(*, zip(x, y)))
```

rewriting

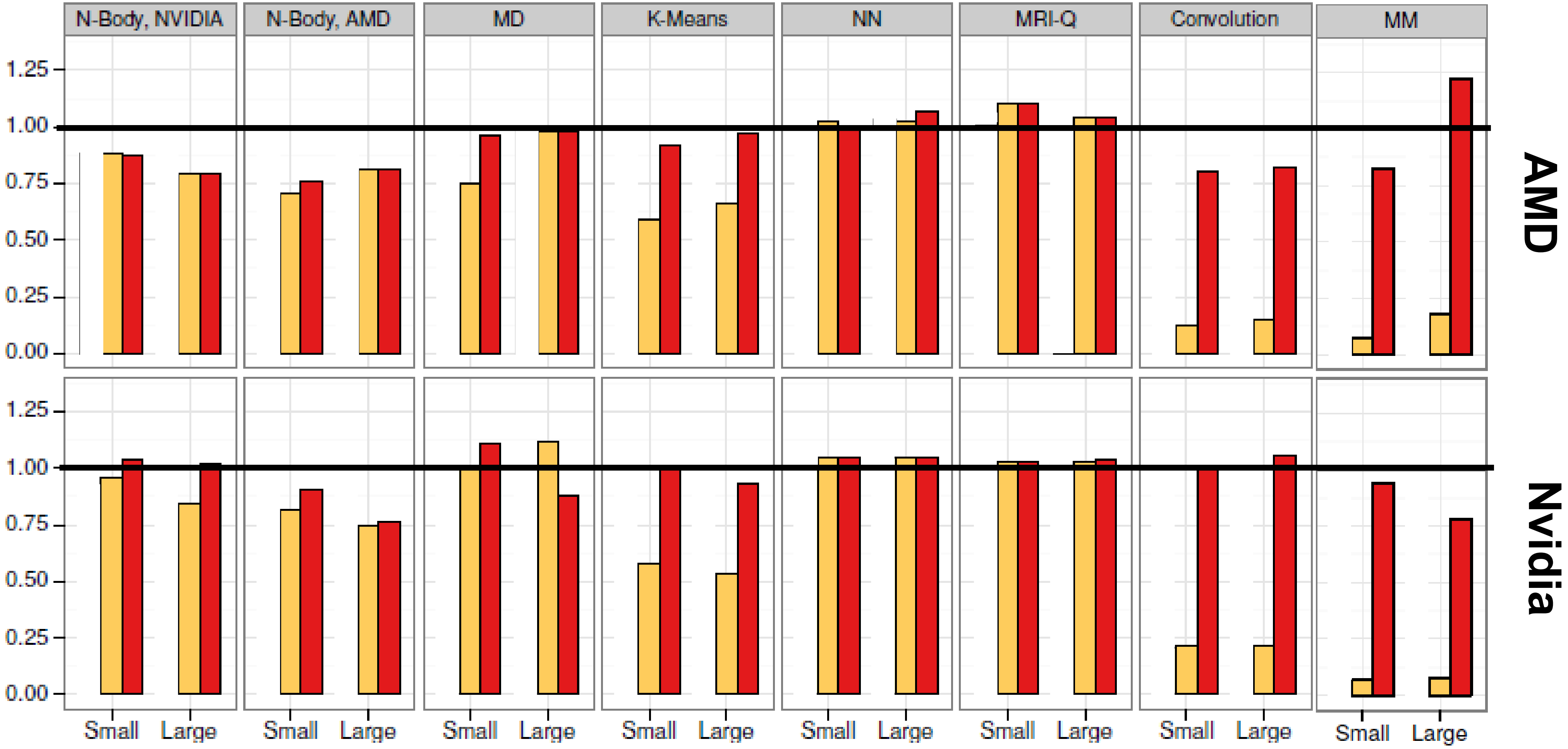
$$
\begin{array}{l}
\texttt{(join} \circ \texttt{mapWrg}^{0}\texttt{(} \\
\quad \texttt{join} \circ \texttt{toGlobal(mapLcl}^{0}\texttt{(mapSeq(id)))} \circ \texttt{split}^{1} \circ \\
\quad \texttt{iterate}^{6}\texttt{(}\ \texttt{join} \circ \\
\qquad\qquad \texttt{mapLcl}^{0}\texttt{(}\ \texttt{toLocal(mapSeq(id))} \circ \\
\qquad\qquad\qquad \texttt{reduceSeq(add,}\ \texttt{0)}\ \texttt{)} \circ \\
\qquad\qquad \texttt{split}^{2}\ \texttt{)} \circ \\
\quad \texttt{join} \circ \texttt{mapLcl}^{0}\texttt{(}\ \texttt{toLocal(mapSeq(id))} \circ \\
\qquad \texttt{reduceSeq(multAndSumUp,}\ \texttt{0)}\ \texttt{)} \circ \texttt{split}^{2} \\
\texttt{)} \circ \texttt{split}^{128}\texttt{)(}\ \texttt{zip(x,}\ \texttt{y)}\ \texttt{)}
\end{array}
$$

code generation

```
kernel void KERNEL(const global float *restrict x,
                   const global float *restrict y,
                   global float *z, int N) {
  local float tmp1[64];  local float tmp2[64];
  local float tmp3[32];
  float acc1; float acc2;
  for (int wg_id  = get_group_id(0); wg_id < N/128;
           wg_id += get_num_groups(0)) {
    { int l_id = get_local_id(0);
      acc1 = 0.0f;
      for (int i = 0; i < 2; i += 1) {
        acc1 = multAndSumUp(acc1,
                 x[2 * l_id + 128 * wg_id + i],
                 y[2 * l_id + 128 * wg_id + i]); }
      tmp1[l_id] = id(acc1); }
    barrier(CLK_LOCAL_MEM_FENCE);
    int size = 64;
    local float *in = tmp1; local float *out = tmp2;
    for (int iter = 0; iter < 6; iter += 1) {
      if (get_local_id(0) < size / 2) {
        acc2 = 0.0f;
        for (int i = 0; i < 2; i += 1) {
          acc2 = add(acc2, in[2 * l_id + i]); }
        out[l_id] = id(acc2); }
      barrier(CLK_LOCAL_MEM_FENCE);
      size = size / 2;
      in   = (out == tmp1) ? tmp1 : tmp3;
      out  = (out == tmp1) ? tmp3 : tmp1;
      barrier(CLK_LOCAL_MEM_FENCE); }
    if (get_local_id(0) < 1) {
      z[wg_id] = id(tmp3[l_id]); }
    barrier(CLK_GLOBAL_MEM_FENCE); } }
```

Code Generator vs Human

🟨 no simplification 🟥 arithmetic simplification



The Lift Approach: Summary

- Hardware-agnostic data-parallel functional intermediate language
  - hides hardware complexity, can be targeted by DSLs

- **Low-level functional language for each hardware type**
  - OpenCL language (this talk), in the future: MPI, OpenMP

- **Rewrite rules based optimisation**
  - maps the hardware-agnostic language to the hardware-specific language
  - extensible

- Possible to achieve high performance through exploration
  - opportunity for smarter technique (e.g. predictive modelling)

if you want to know more: www.lift-project.org

supported by: Google