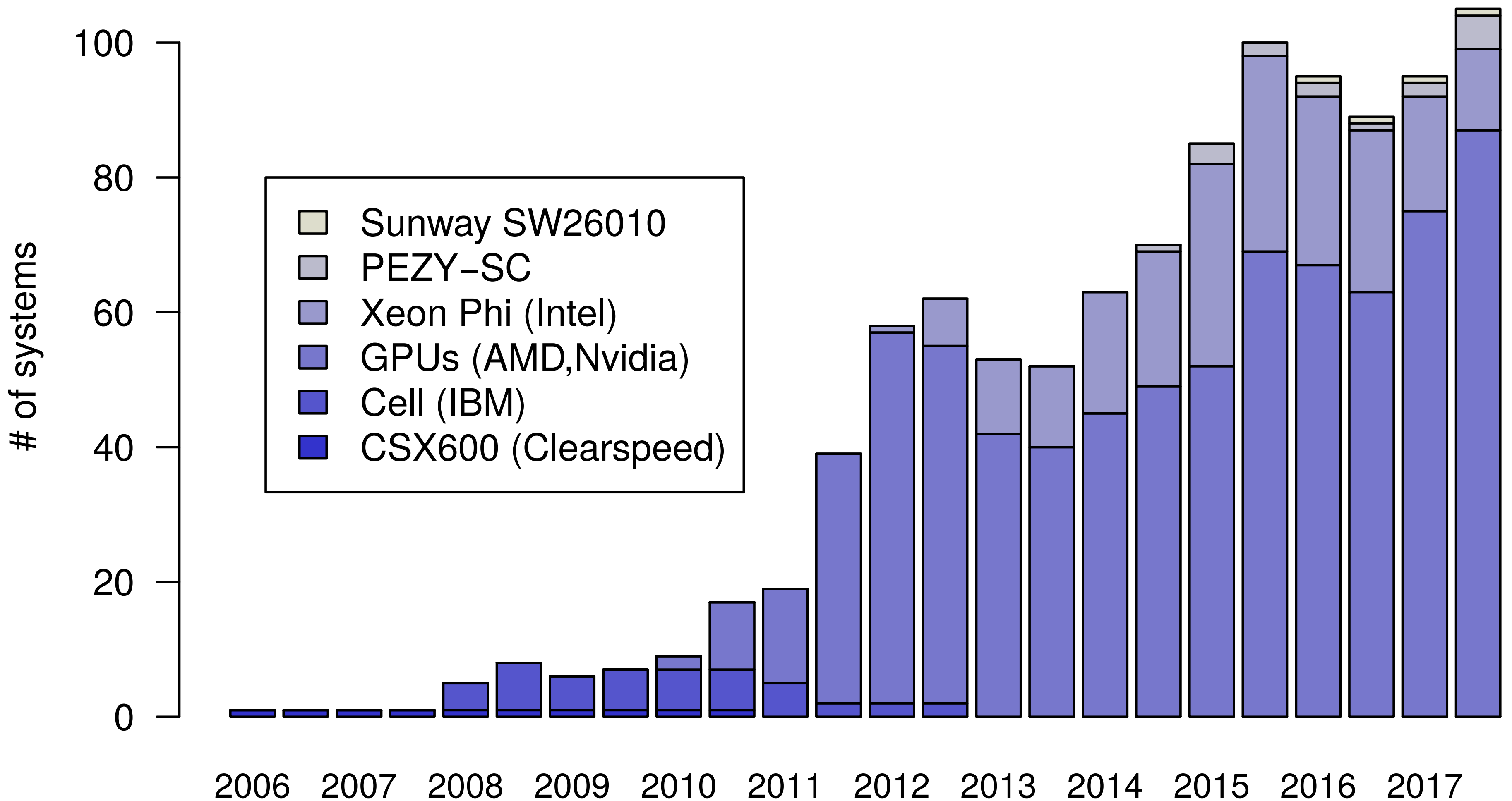
# Lift
# Performance Portable Code Generation on Parallel Accelerators

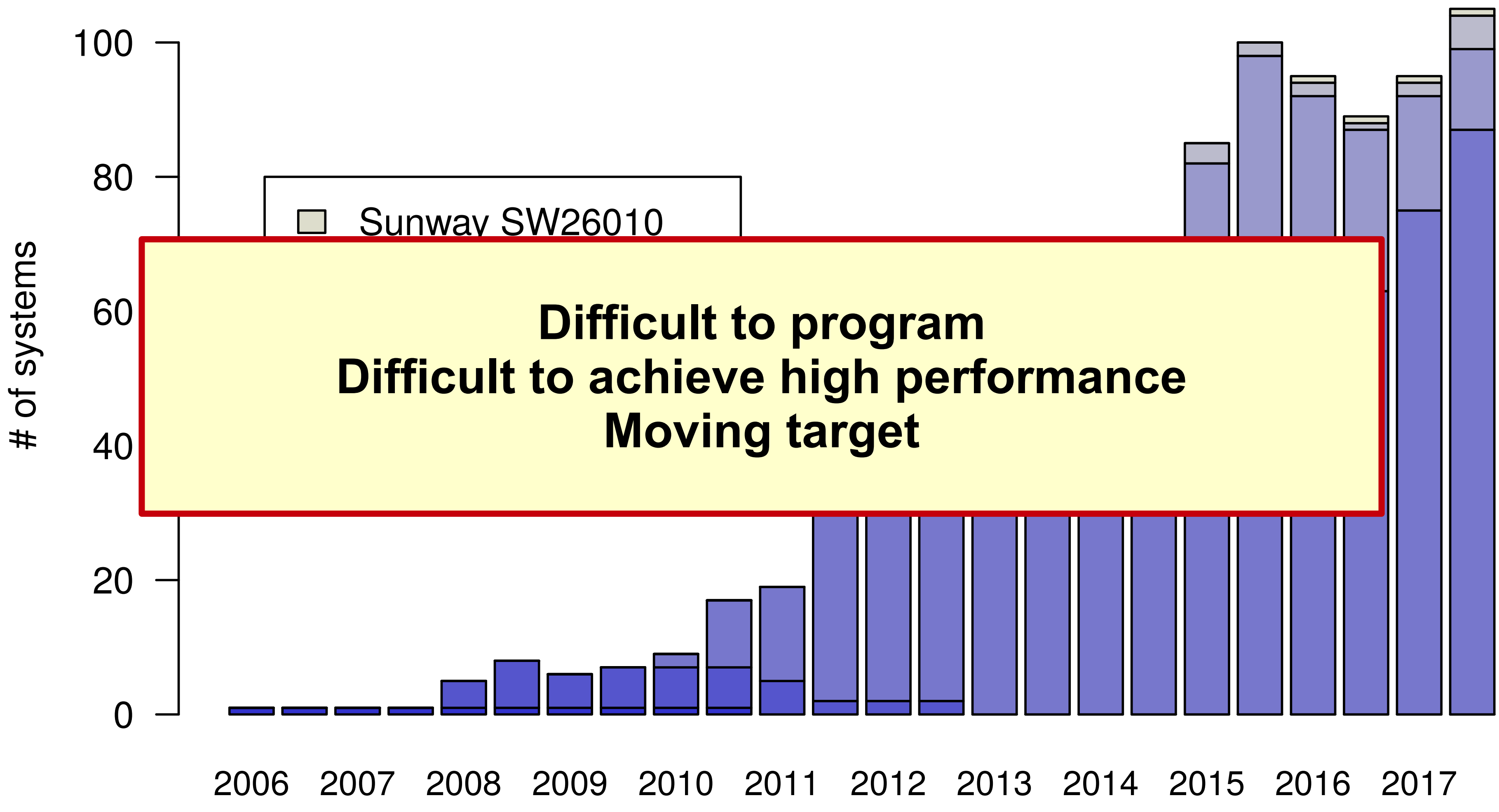
ISPASS tutorial

2 Apr. 2018

# Top 500 with parallel accelerators



# Top 500 with parallel accelerators




**Difficult to program**
**Difficult to achieve high performance**
**Moving target**

# How to sum an array?

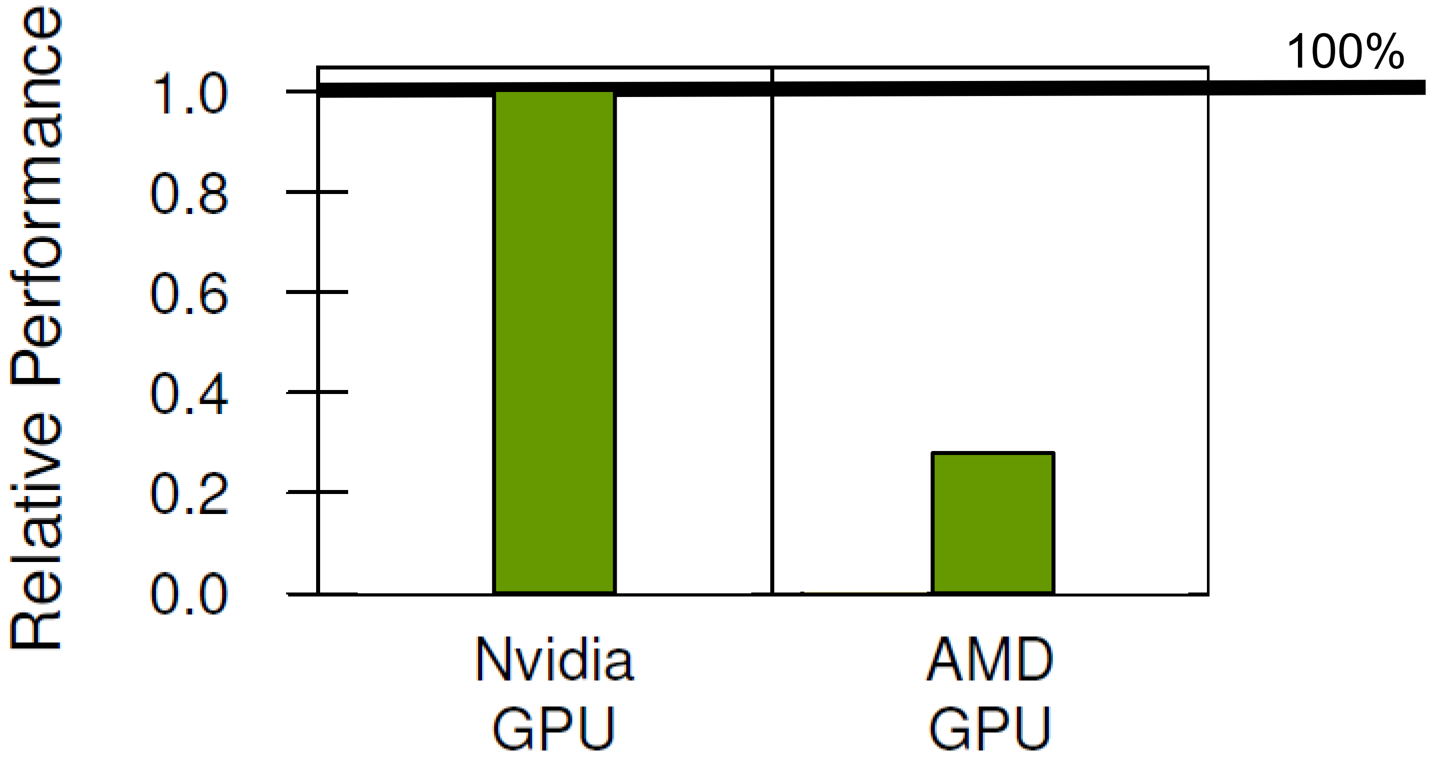
# How to sum an array?

```
float acc = 0;
for (int i=0; i<N; i++)
  acc += input[i];
```

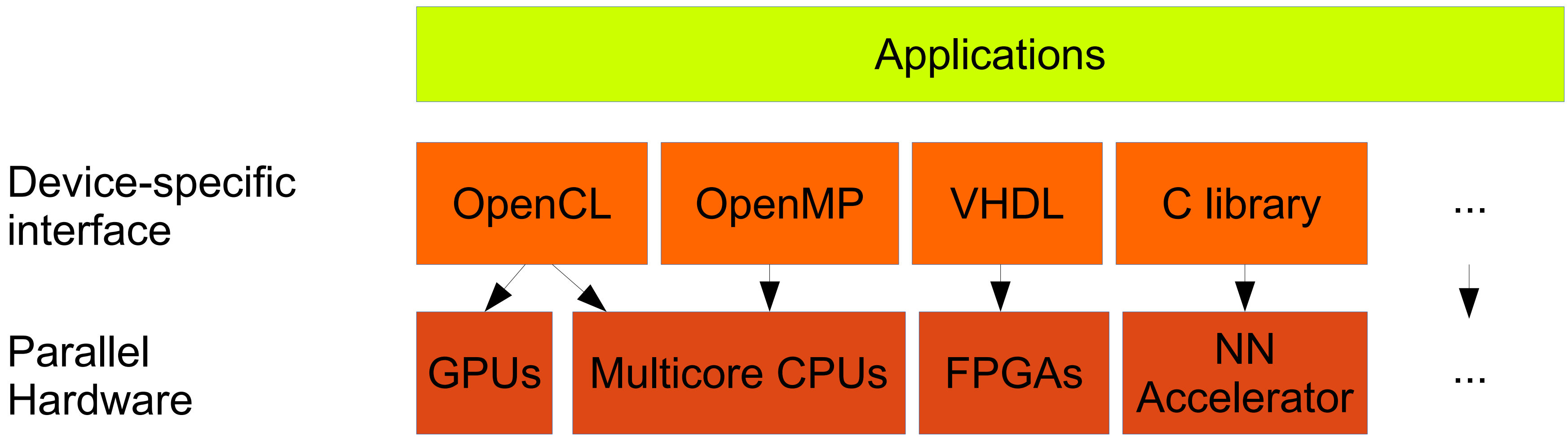
# How to really sum an array:

```
kernel void sum(global float* g_in, global float* g_out,
                unsigned int n, local volatile float* l_data) {

  unsigned int tid = get_local_id(0);
  unsigned int i   = get_group_id(0) * 256 + get_local_id(0);

  l_data[tid] = 0;
  while (i < n) {
    l_data[tid] += g_in[i];
    i += 256 * get_num_groups(0);
  }
  barrier(CLK_LOCAL_MEM_FENCE);

  if (tid < 128)
    l_data[tid] += l_data[tid+128];
  barrier(CLK_LOCAL_MEM_FENCE);
  if (tid <  64)
    l_data[tid] += l_data[tid+ 64];
  barrier(CLK_LOCAL_MEM_FENCE)

  if (tid < 32) {
    l_data[tid] += l_data[tid+32]; l_data[tid] += l_data[tid+16];
    l_data[tid] += l_data[tid+ 8]; l_data[tid] += l_data[tid+ 4];
    l_data[tid] += l_data[tid+ 2]; l_data[tid] += l_data[tid+ 1];
  }
  if (tid == 0)
    g_out[get_group_id(0)] = l_data[0];
}
```

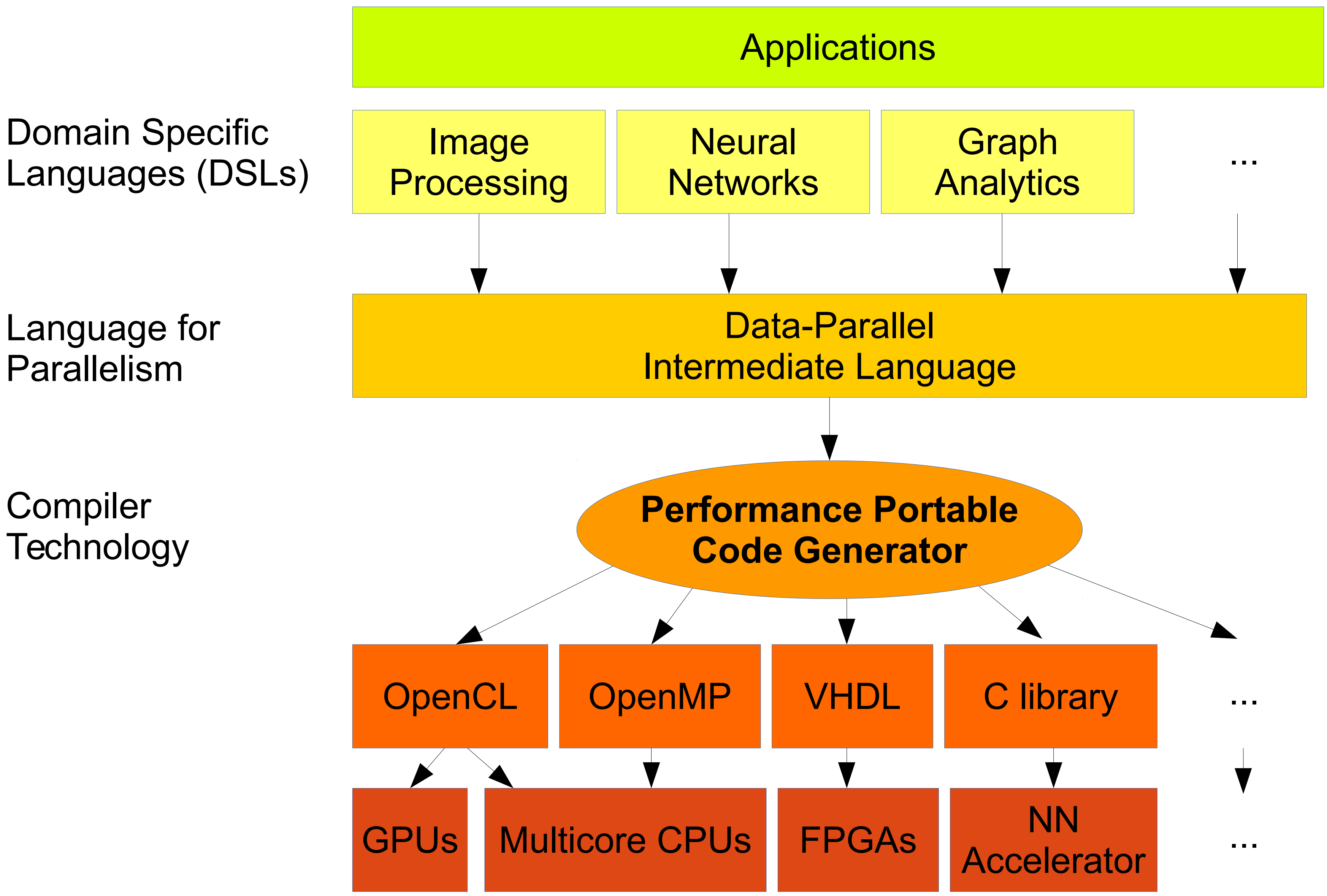
# Performance is not portable



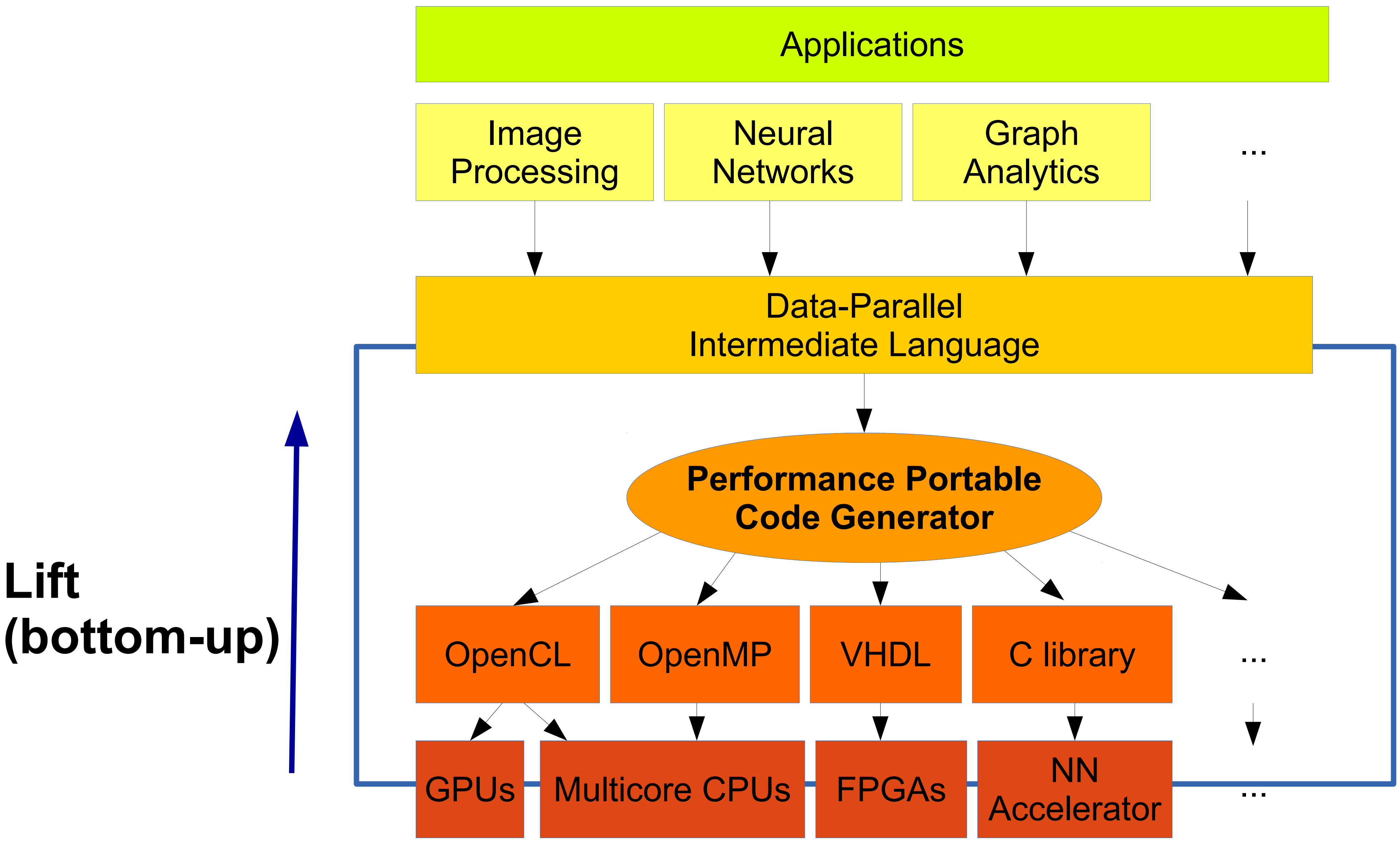
# Current landscape



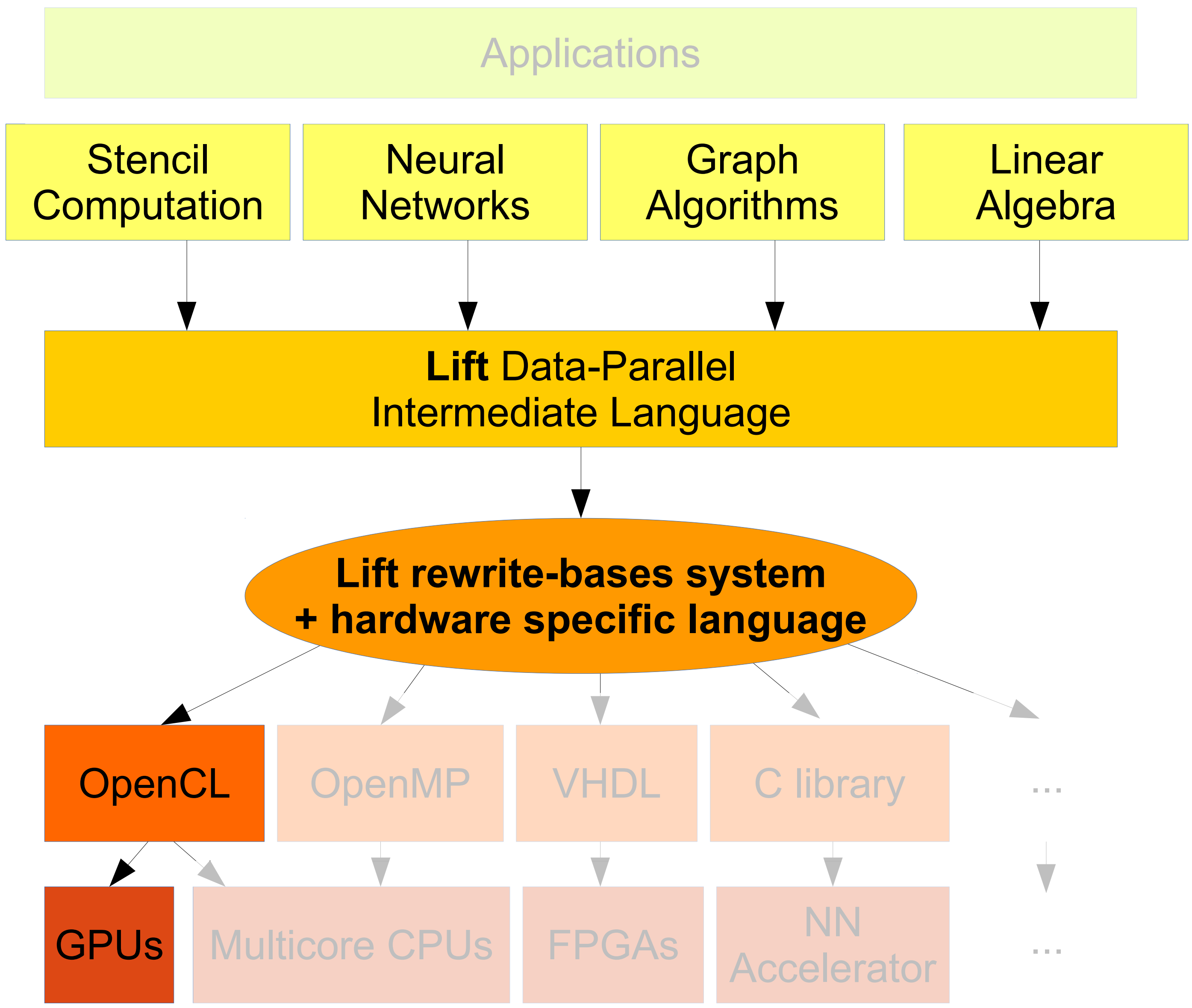
# What we need


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
C library
...
GPUs
Multicore CPUs
FPGAs
NN Accelerator
...

# Bottom up Approach



# Current work



# Evolution of programming

# Evolution of programming

Assembly

```
        la    $s0, input;
        li    $t1, 0
sum:    beq   $t1, $t0, endSun
        lw    $t7, 0(s0)
        addi  $t4, $t4, $t7
        addi  $s0, $s0, 4
        addi  $t3, $t3, 1
        jump  sum
```

# Evolution of programming

Assembly

```
        la    $s0, input;
        li    $t1, 0
sum:    beq   $t1, $t0, endSun
        lw    $t7, 0(s0)
        addi  $t4, $t4, $t7
        addi  $s0, $s0, 4
        addi  $t3, $t3, 1
        jump  sum
```

---

Structured

```
float acc = 0;
  for (int i=0; i<N; i++)
    acc += input[i];
```

# Evolution of programming

Assembly

```
        la    $s0, input;
        li    $t1, 0
sum:    beq   $t1, $t0, endSun
        lw    $t7, 0(s0)
        addi  $t4, $t4, $t7
        addi  $s0, $s0, 4
        addi  $t3, $t3, 1
        jump  sum
```

Structured

```
float acc = 0;
  for (int i=0; i<N; i++)
    acc += input[i];
```

Algorithmic

```
reduce(0,+,input)
```

# Evolution of programming

Assembly

```
la    $s0, input;
li    $t1, 0
```

**Generic**
**Hard to program**
**Hard to optimise automatically**

```
addi  $t3, $t3, 1
jump  sum
```

---

Structured

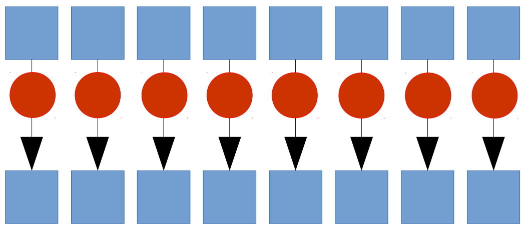
```
float acc = 0;
  for (int i=0; i<N; i++)
    acc += input[i];
```

---

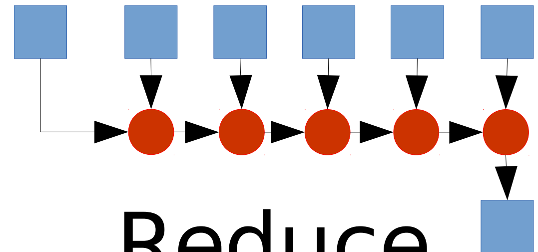
Algorithmic

**More constraints**
**Easier to use**
**Easier to optimise automatically**

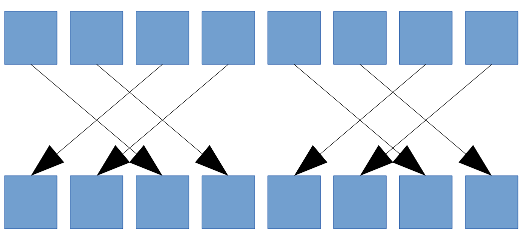
# Algorithmic Patterns



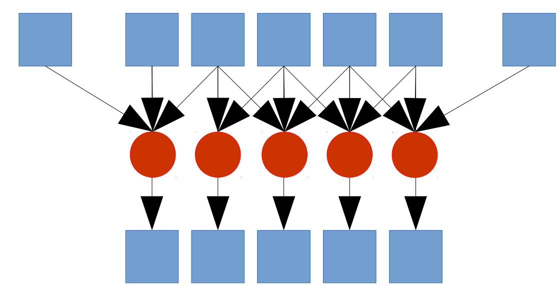
Map



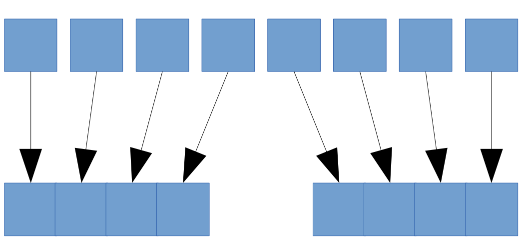
Reduce



Gather



Stencil

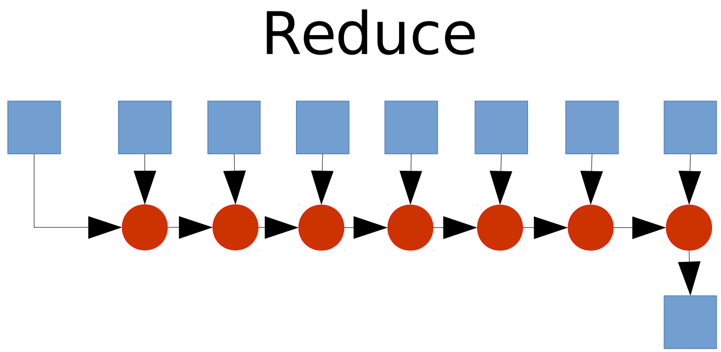


Partition

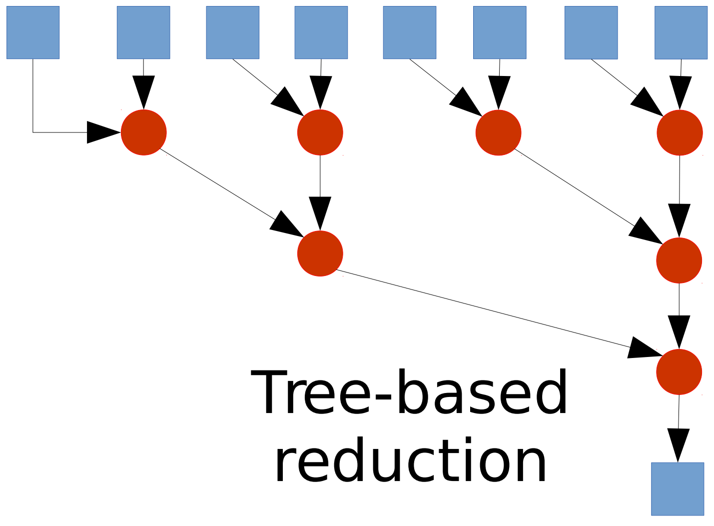
...

# Separation of concern

**Programming abstraction**




**Implementation**



# Back to high performance code

```
kernel void sum(global float* g_in, global float* g_out,
                unsigned int n, local volatile float* l_data) {

  unsigned int tid = get_local_id(0);
  unsigned int i   = get_group_id(0) * 256 + get_local_id(0);

  l_data[tid] = 0;
  while (i < n) {
    l_data[tid] += g_in[i];
    i += 256 * get_num_groups(0);
  }
  barrier(CLK_LOCAL_MEM_FENCE);

  if (tid < 128)
    l_data[tid] += l_data[tid+128];
  barrier(CLK_LOCAL_MEM_FENCE);
  if (tid <  64)
    l_data[tid] += l_data[tid+ 64];
  barrier(CLK_LOCAL_MEM_FENCE)

  if (tid < 32) {
    l_data[tid] += l_data[tid+32]; l_data[tid] += l_data[tid+16];
    l_data[tid] += l_data[tid+ 8]; l_data[tid] += l_data[tid+ 4];
    l_data[tid] += l_data[tid+ 2]; l_data[tid] += l_data[tid+ 1];
  }
  if (tid == 0)
    g_out[get_group_id(0)] = l_data[0];
}
```


## How did we get there?

- Built step by step by hand
- Human “knows” good patterns of optimisations

# Back to high performance code

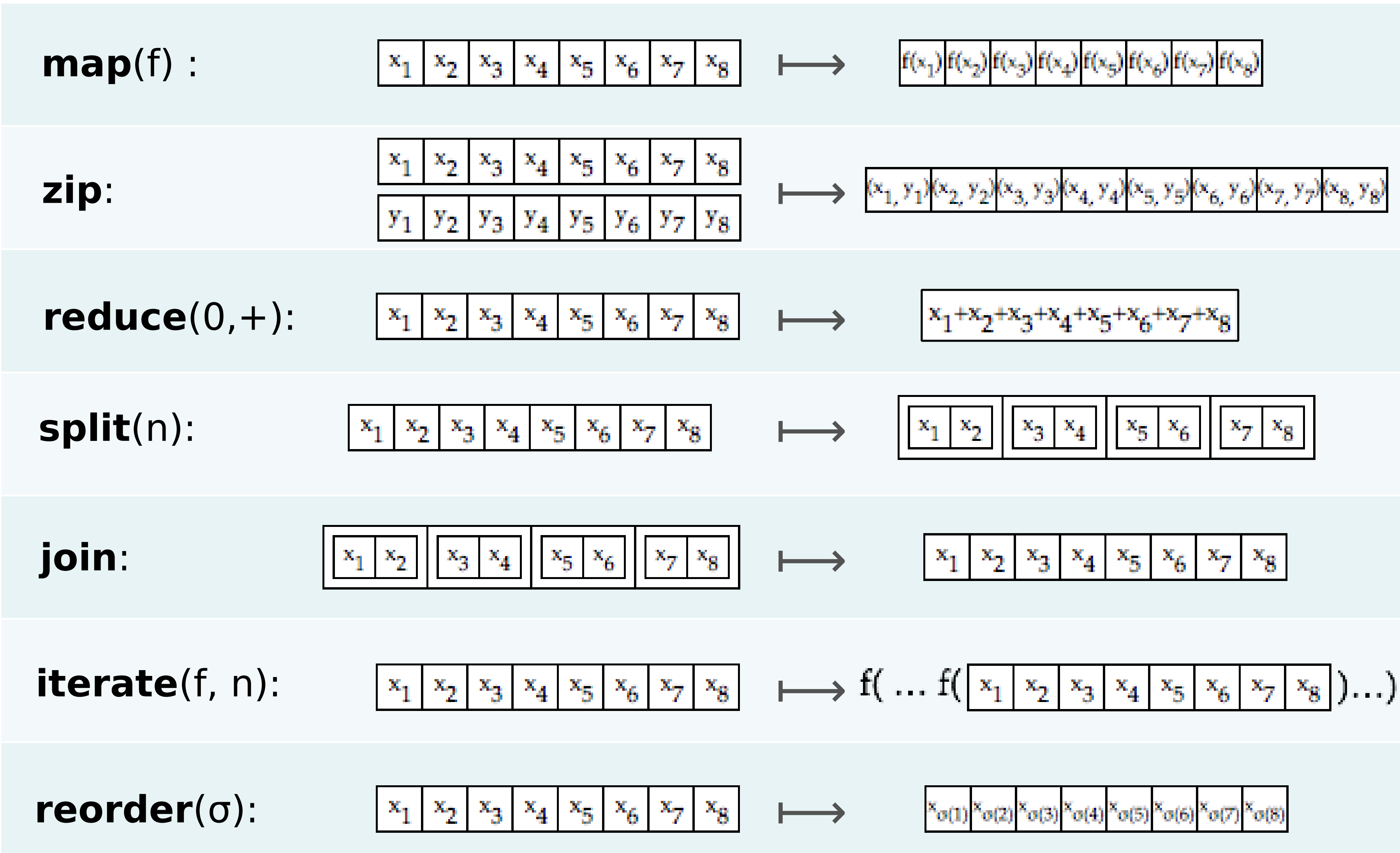
```
kernel void sum(global float* g_in, global float* g_out,
                unsigned int n, local volatile float* l_data) {

  unsigned int tid = get_local_id(0);
  unsigned int i   = get_group_id(0) * 256 + get_local_id(0);

  l_data[tid] = 0;
  while (i < n) {
    l_data[tid] += g_in[i];
    i += 256 * get_num_groups(0);
  }
  barrier(CLK_LOCAL_MEM_FENCE);

  if (tid < 128)
    l_data[tid] += l_data[tid+128];
  barrier(CLK_LOCAL_MEM_FENCE);
  if (tid <  64)
    l_data[tid] += l_data[tid+ 64];
  barrier(CLK_LOCAL_MEM_FENCE)

  if (tid < 32) {
    l_data[tid] += l_data[tid+32]; l_data[tid] += l_data[tid+16];
    l_data[tid] += l_data[tid+ 8]; l_data[tid] += l_data[tid+ 4];
    l_data[tid] += l_data[tid+ 2]; l_data[tid] += l_data[tid+ 1];
  }
  if (tid == 0)
    g_out[get_group_id(0)] = l_data[0];
}
```


## How did we get there?

**How to get there automatically?**

# The Lift Approach

# Lift Intermediate Language




Focus on the **what** rather than **how**

# Abstracting the implementation

```
kernel void sum(global float* g_in, global float* g_out,
                unsigned int n, local volatile float* l_data) {

  unsigned int tid = get_local_id(0);
  unsigned int i   = get_group_id(0) * 256 + get_local_id(0);

  l_data[tid] = 0;
  while (i < n) {
    l_data[tid] += g_in[i];
    i += 256 * get_num_groups(0);
  }
  barrier(CLK_LOCAL_MEM_FENCE);

  if (tid < 128)
    l_data[tid] += l_data[tid+128];
  barrier(CLK_LOCAL_MEM_FENCE);
  if (tid <  64)
    l_data[tid] += l_data[tid+ 64];
  barrier(CLK_LOCAL_MEM_FENCE)

  if (tid < 32) {
    l_data[tid] += l_data[tid+32]; l_data[tid] += l_data[tid+16];
    l_data[tid] += l_data[tid+ 8]; l_data[tid] += l_data[tid+ 4];
    l_data[tid] += l_data[tid+ 2]; l_data[tid] += l_data[tid+ 1];
  }
  if (tid == 0)
    g_out[get_group_id(0)] = l_data[0];
}
```

**OpenCL**

```
split(256) >>
reorder >>
mapWorkGroup(
    reduceSeq(
        toLocal(
            mapLocalThread(0)),
        zip(input) >>
        mapLocalThread(
            +)) >>
    iterate(
        reorder  >>
        split(2) >>
        mapLocalThread(
            reduceSeq(0,+))) >>
    iterate(
        reorder  >>
        split(2) >>
        mapWarp(
            reduceSeq(0,+))) >>
    toGlobal(
        mapLocaThread(id)))
```

**Lift IR**

# Abstracting the implementation

```
kernel void sum(global float* g_in, global float* g_out,
                unsigned int n, local volatile float* l_data) {
  unsigned int tid = get_local_id(0);
  unsigned int i   = get_group_id(0) * 256 + get_local_id(0);

  l_data[tid] = 0;
  while (i < n) {
    l_data[tid] += g_in[i];
    i += 256 * get_num_groups(0);
  }
  barrier(CLK_LOCAL_MEM_FENCE);

  if (tid < 128)
    l_data[tid] += l_data[tid+128];
  barrier(CLK_LOCAL_MEM_FENCE);
  if (tid <  64)
    l_data[tid] += l_data[tid+ 64];
  barrier(CLK_LOCAL_MEM_FENCE)

  if (tid < 32) {
    l_data[tid] += l_data[tid+32]; l_data[tid] += l_data[tid+16];
    l_data[tid] += l_data[tid+ 8]; l_data[tid] += l_data[tid+ 4];
    l_data[tid] += l_data[tid+ 2]; l_data[tid] += l_data[tid+ 1];
  }
  if (tid == 0)
    g_out[get_group_id(0)] = l_data[0];
}
```

**OpenCL**

```
split(256) >>
reorder >>
mapWorkGroup(
    reduceSeq(
        toLocal(
            mapLocalThread(0)),
        zip(input) >>
        mapLocalThread(
            +)) >>
    iterate(
        reorder  >>
        split(2) >>
        mapLocalThread(
            reduceSeq(0,+))) >>
    iterate(
        reorder  >>
        split(2) >>
        mapWarp(
            reduceSeq(0,+))) >>
    toGlobal(
        mapLocaThread(id)))
```

**Lift IR**

# Abstracting the implementation

```
kernel void sum(global float* g_in, global float* g_out,
                unsigned int n, local volatile float* l_data) {

  unsigned int tid = get_local_id(0);
  unsigned int i   = get_group_id(0) * 256 + get_local_id(0);

  l_data[tid] = 0;
  while (i < n) {
    l_data[tid] += g_in[i];
    i += 256 * get_num_groups(0);
  }
  barrier(CLK_LOCAL_MEM_FENCE);

  if (tid < 128)
    l_data[tid] += l_data[tid+128];
  barrier(CLK_LOCAL_MEM_FENCE);
  if (tid <  64)
    l_data[tid] += l_data[tid+ 64];
  barrier(CLK_LOCAL_MEM_FENCE)

  if (tid < 32) {
    l_data[tid] += l_data[tid+32]; l_data[tid] += l_data[tid+16];
    l_data[tid] += l_data[tid+ 8]; l_data[tid] += l_data[tid+ 4];
    l_data[tid] += l_data[tid+ 2]; l_data[tid] += l_data[tid+ 1];
  }
  if (tid == 0)
    g_out[get_group_id(0)] = l_data[0];
}
```

**OpenCL**

```
split(256) >>
reorder >>
mapWorkGroup(
    reduceSeq(
        toLocal(
            mapLocalThread(0)),
        zip(input) >>
        mapLocalThread(
            +)) >>
    iterate(
        reorder  >>
        split(2) >>
        mapLocalThread(
            reduceSeq(0,+))) >>
    iterate(
        reorder  >>
        split(2) >>
        mapWarp(
            reduceSeq(0,+))) >>
    toGlobal(
        mapLocaThread(id)))
```

**Lift IR**

# Abstracting the implementation

```
kernel void sum(global float* g_in, global float* g_out,
                unsigned int n, local volatile float* l_data) {

  unsigned int tid = get_local_id(0);
  unsigned int i   = get_group_id(0) * 256 + get_local_id(0);

  l_data[tid] = 0;
  while (i < n) {
    l_data[tid] += g_in[i];
    i += 256 * get_num_groups(0);
  }
  barrier(CLK_LOCAL_MEM_FENCE);

  if (tid < 128)
    l_data[tid] += l_data[tid+128];
  barrier(CLK_LOCAL_MEM_FENCE);
  if (tid <  64)
    l_data[tid] += l_data[tid+ 64];
  barrier(CLK_LOCAL_MEM_FENCE)

  if (tid < 32) {
    l_data[tid] += l_data[tid+32]; l_data[tid] += l_data[tid+16];
    l_data[tid] += l_data[tid+ 8]; l_data[tid] += l_data[tid+ 4];
    l_data[tid] += l_data[tid+ 2]; l_data[tid] += l_data[tid+ 1];
  }
  if (tid == 0)
    g_out[get_group_id(0)] = l_data[0];
}
```

**OpenCL**

```
split(256) >>
reorder >>
mapWorkGroup(
    reduceSeq(
        toLocal(
            mapLocalThread(0)),
        zip(input) >>
        mapLocalThread(
            +)) >>
    iterate(
        reorder  >>
        split(2) >>
        mapLocalThread(
            reduceSeq(0,+))) >>
    iterate(
        reorder  >>
        split(2) >>
        mapWarp(
            reduceSeq(0,+))) >>
    toGlobal(
        mapLocaThread(id)))
```

**Lift IR**

# Abstracting the implementation

```
kernel void sum(global float* g_in, global float* g_out,
                unsigned int n, local volatile float* l_data) {

  unsigned int tid = get_local_id(0);
  unsigned int i   = get_group_id(0) * 256 + get_local_id(0);

  l_data[tid] = 0;
  while (i < n) {
    l_data[tid] += g_in[i];
    i += 256 * get_num_groups(0);
  }
  barrier(CLK_LOCAL_MEM_FENCE);

  if (tid < 128)
    l_data[tid] += l_data[tid+128];
  barrier(CLK_LOCAL_MEM_FENCE);
  if (tid <  64)
    l_data[tid] += l_data[tid+ 64];
  barrier(CLK_LOCAL_MEM_FENCE)

  if (tid < 32) {
    l_data[tid] += l_data[tid+32]; l_data[tid] += l_data[tid+16];
    l_data[tid] += l_data[tid+ 8]; l_data[tid] += l_data[tid+ 4];
    l_data[tid] += l_data[tid+ 2]; l_data[tid] += l_data[tid+ 1];
  }
  if (tid == 0)
    g_out[get_group_id(0)] = l_data[0];
}
```

**OpenCL**

```
split(256) >>
reorder >>
mapWorkGroup(
    reduceSeq(
        toLocal(
            mapLocalThread(0)),
        zip(input) >>
        mapLocalThread(
            +)) >>
    iterate(
        reorder  >>
        split(2) >>
        mapLocalThread(
            reduceSeq(0,+))) >>
    iterate(
        reorder  >>
        split(2) >>
        mapWarp(
            reduceSeq(0,+))) >>
    toGlobal(
        mapLocaThread(id)))
```

**Lift IR**

# Abstracting the implementation

```
kernel void sum(global float* g_in, global float* g_out,
                unsigned int n, local volatile float* l_data) {

  unsigned int tid = get_local_id(0);
  unsigned int i   = get_group_id(0) * 256 + get_local_id(0);

  l_data[tid] = 0;
  while (i < n) {
    l_data[tid] += g_in[i];
    i += 256 * get_num_groups(0);
  }
  barrier(CLK_LOCAL_MEM_FENCE);

  if (tid < 128)
    l_data[tid] += l_data[tid+128];
  barrier(CLK_LOCAL_MEM_FENCE);
  if (tid <  64)
    l_data[tid] += l_data[tid+ 64];
  barrier(CLK_LOCAL_MEM_FENCE)

  if (tid < 32) {
    l_data[tid] += l_data[tid+32]; l_data[tid] += l_data[tid+16];
    l_data[tid] += l_data[tid+ 8]; l_data[tid] += l_data[tid+ 4];
    l_data[tid] += l_data[tid+ 2]; l_data[tid] += l_data[tid+ 1];
  }
  if (tid == 0)
    g_out[get_group_id(0)] = l_data[0];
}
```

**OpenCL**

```
split(256) >>
reorder >>
mapWorkGroup(
    reduceSeq(
        toLocal(
            mapLocalThread(0)),
        zip(input) >>
        mapLocalThread(
            +)) >>
    iterate(
        reorder  >>
        split(2) >>
        mapLocalThread(
            reduceSeq(0,+))) >>
    iterate(
        reorder  >>
        split(2) >>
        mapWarp(
            reduceSeq(0,+))) >>
    toGlobal(
        mapLocaThread(id)))
```

**Lift IR**

# Observations

- Can describe implementation as composition of primitives
  - map, reduce, zip, ...
  - same ones we use at the high level!

- Can express complex optimisations
  - e.g. shared memory use

# How to generate optimised version?

## Programming abstraction

```
reduce(0,+,input)
```

→

## Implementation

```
split(256) >>
reorder >>
mapWorkGroup(
    reduceSeq(
        toLocal(
            mapLocalThread(0)),
        zip(input) >>
        mapLocalThread(
            +)) >>
    iterate(
        reorder  >>
        split(2) >>
        mapLocalThread(
            reduceSeq(0,+))) >>
    iterate(
        reorder  >>
        split(2) >>
        mapWarp(
            reduceSeq(0,+))) >>
    toGlobal(
        mapLocaThread(id)))
```

# Algorithmic Rewrite Rules

- Provably correct rewrite rules
- Express algorithmic implementation choices

## Split-join rule:

```
map(f) → split(n) >> map(map(f)) >> join
```

## Map fusion rule:

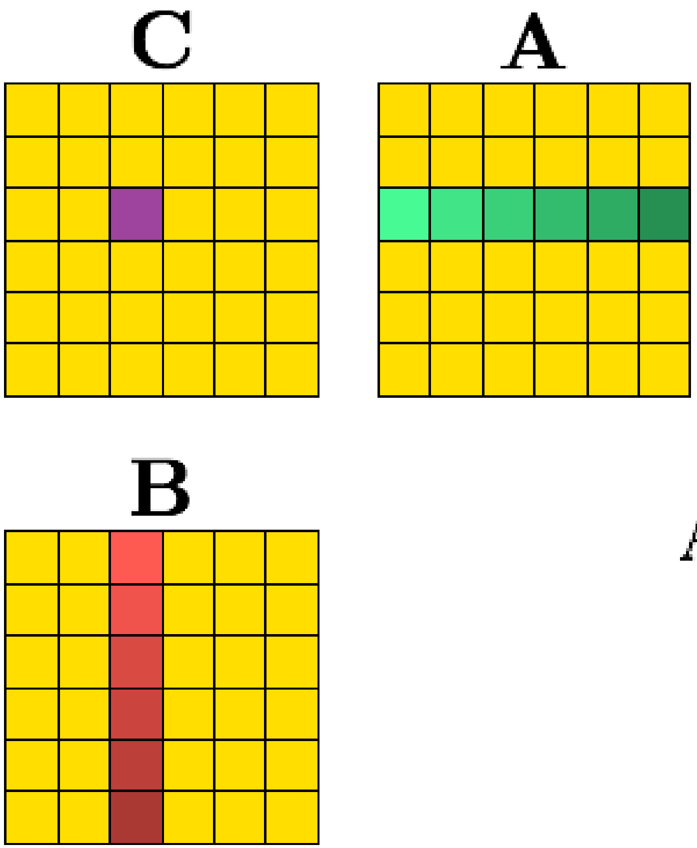
```
map(f) >> map(g) → map(f >> g)
```

## Reduce rules:

```
reduce(z,f)     → reducePart(z,f) >> reduce(f)

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)
```

# Matrix-multiplication example

# Matrix Multiplication in Lift




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

$$
\begin{array}{l}
\mathrm{A} \gg \mathrm{map}(\lambda\ \ rowOfA\ \mapsto \\
\quad \mathrm{B} \gg \mathrm{map}(\lambda\ \ colOfB \mapsto \\
\qquad \mathtt{zip}(rowOfA,\ \ colOfB) \gg \\
\qquad \mathrm{map}(\mathtt{mult}) \gg \mathtt{reduce}(0.0\mathtt{f}, \mathtt{add}) \\
\quad ) \\
)
\end{array}
$$

```
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];
          }
        }
      }
    }
  }
}
```

# 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

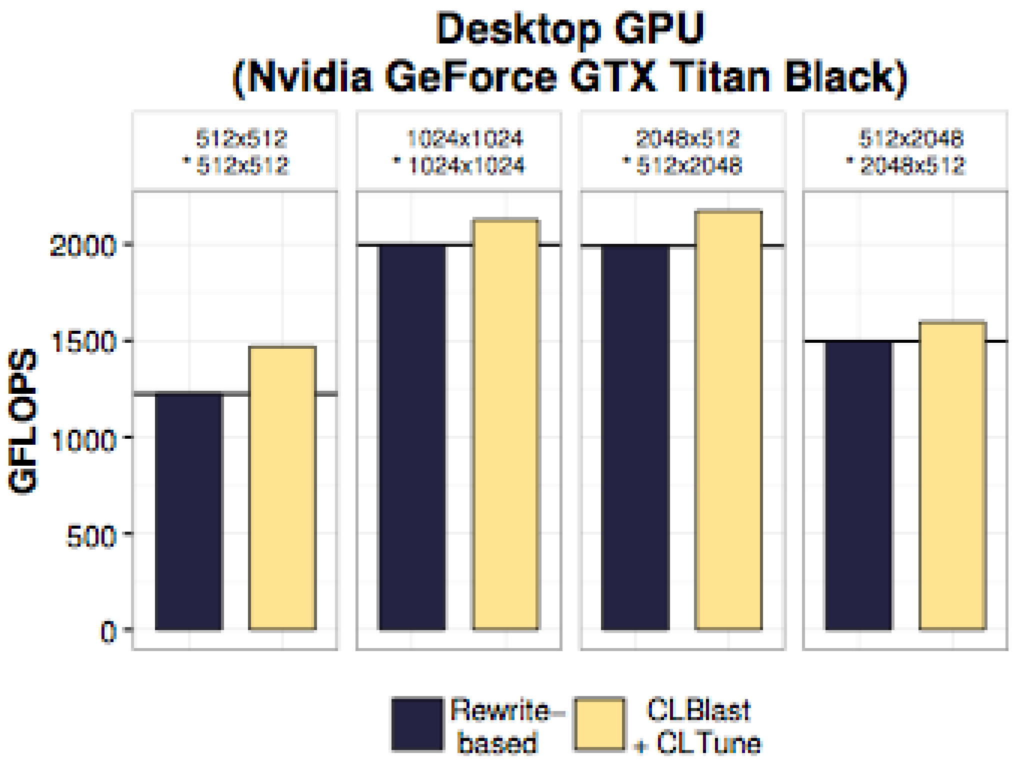
```
λ (A, B) ↦
 A >> split(m) >> mapGlb0(λ nRowsOfA ↦
  B >> split(n) >> mapGlb1(λ mColsOfB ↦
    zip( transpose(nRowsOfA) >> split(k),
         transpose(mColsOfB) >> split(k)  ) >>
    reduceSeq(init = make2DArray(n,m, 0.0f) >>
              toPrivate(mapSeq(mapSeq(id))),
     λ (accTile, (tileOfA, 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
```

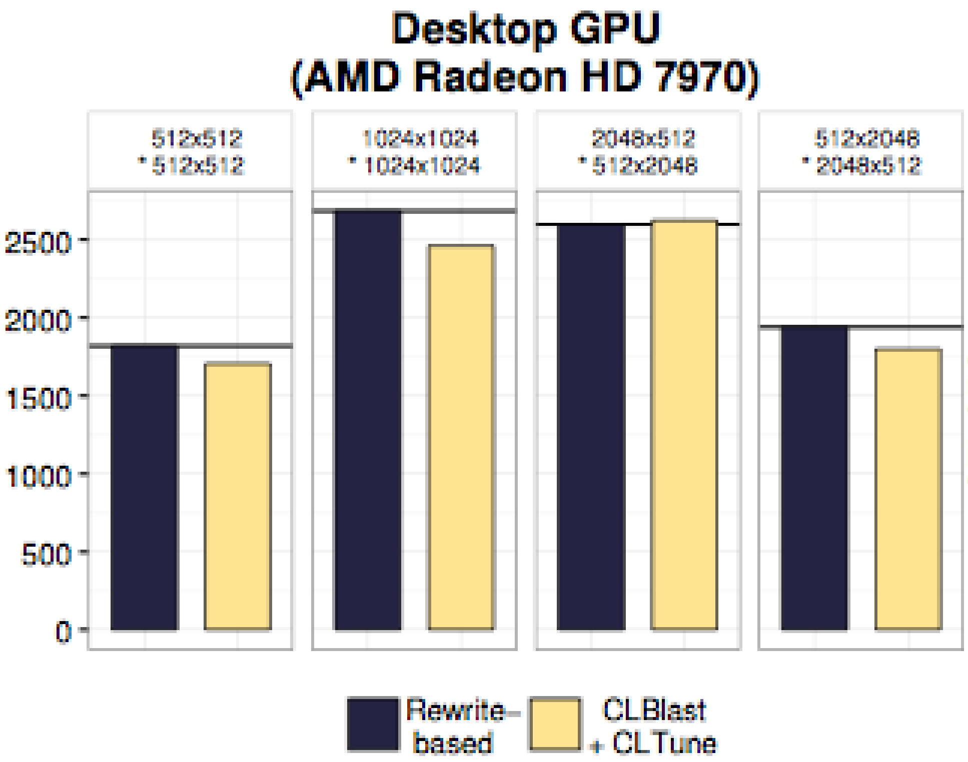
```
int i = get_global_id(0);
int j = get_global_id(1);

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);
```

# Performance Portability Achieved

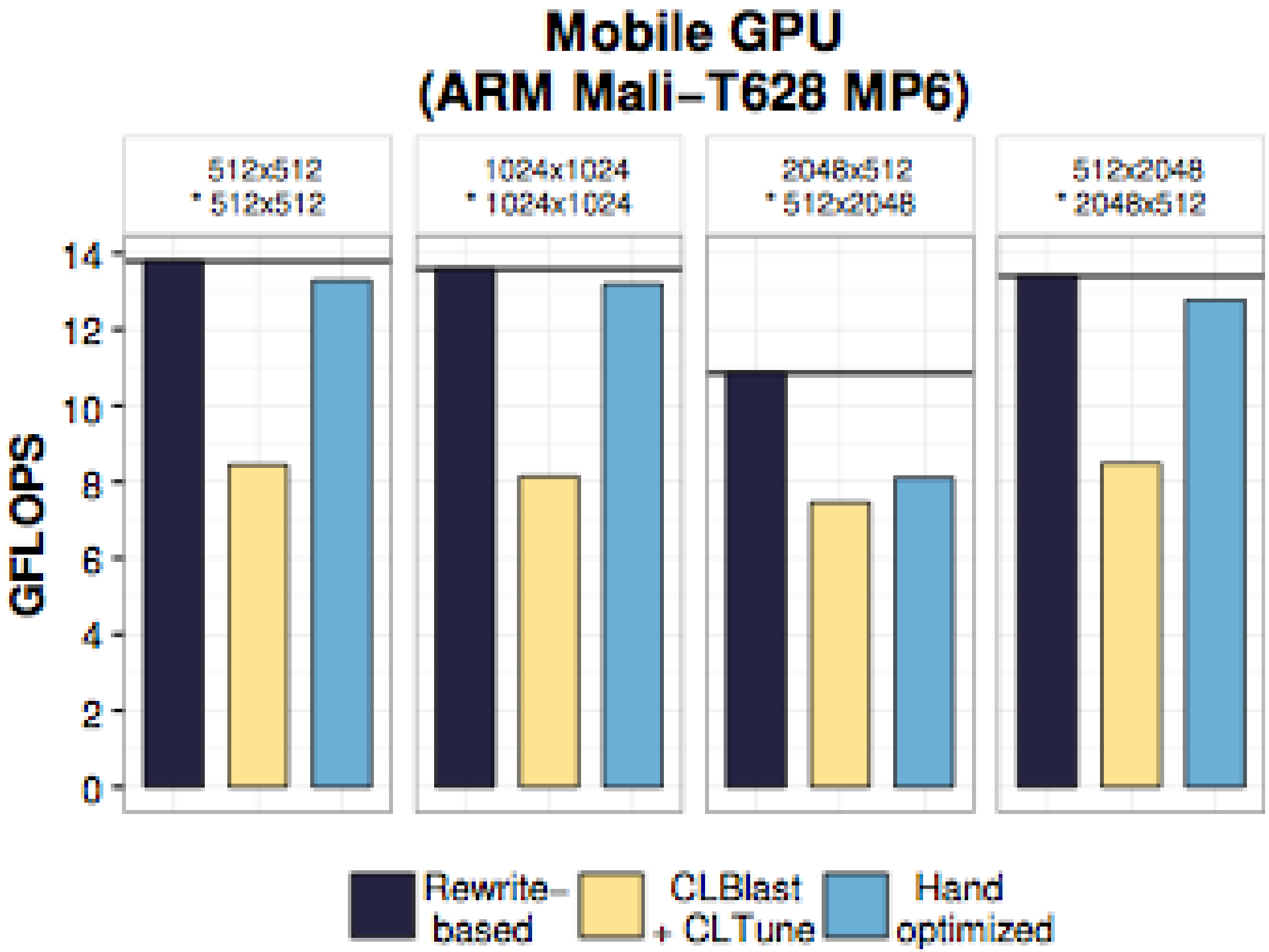
**Compiler input:**

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



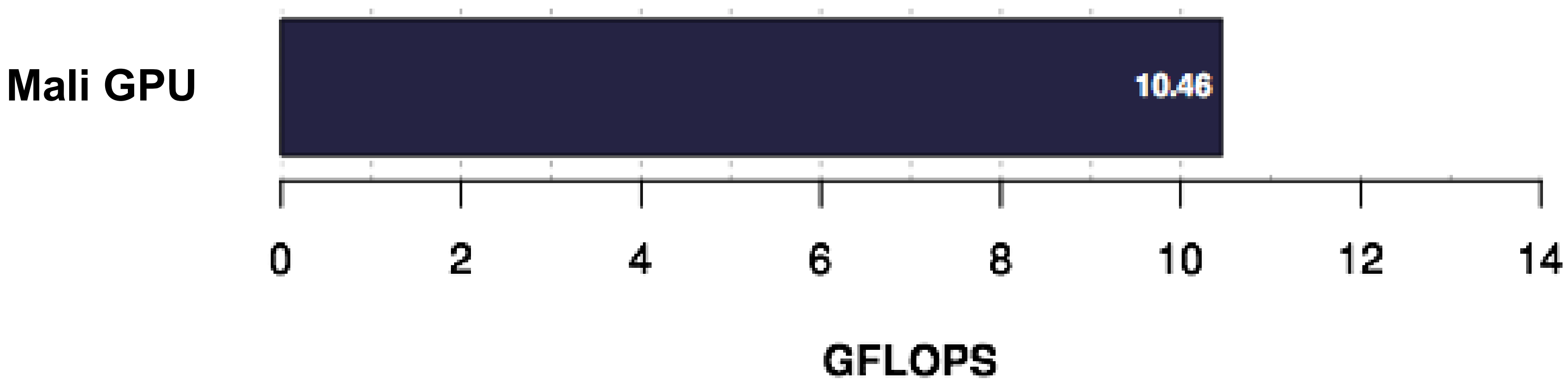



# Also works on mobile GPUs



# Easily Extensible

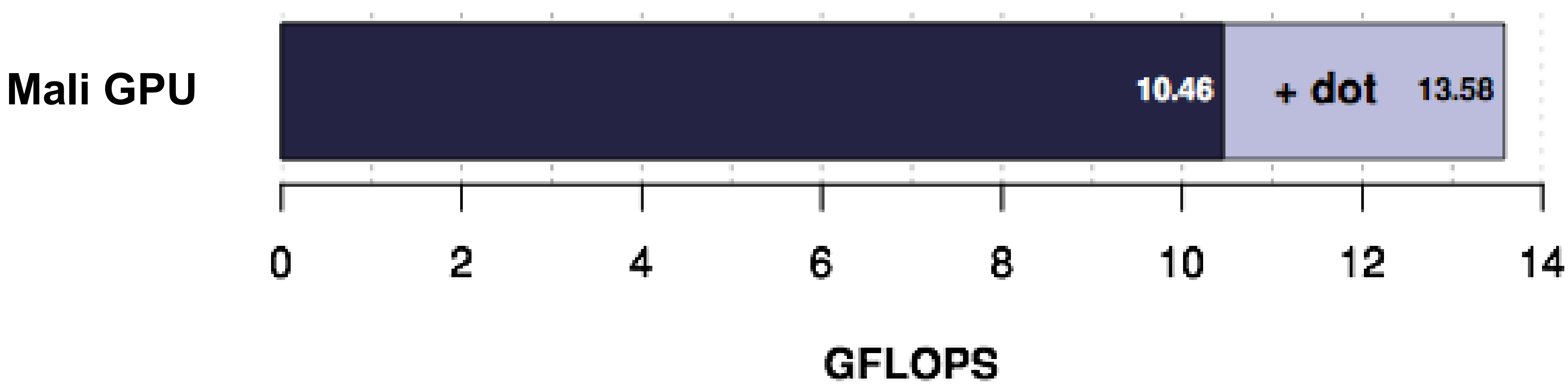
- New rules can be added
- E.g. dot built-in



# Easily Extensible

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

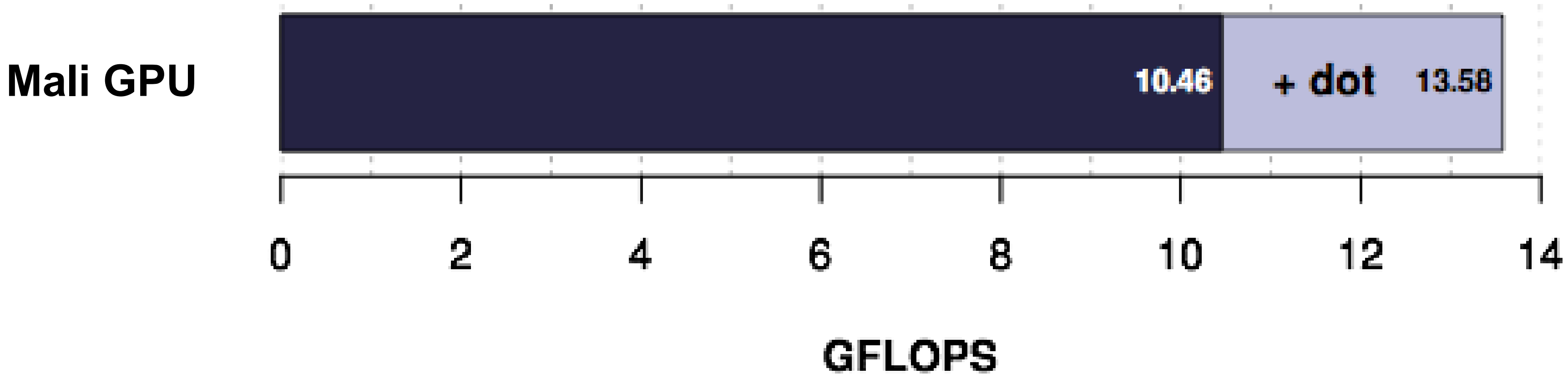
```
zip(x,y) >> mapSeq(mult4) >> reduceSeq(z, add4) → dot(x, y)
```



# Easily Extensible

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

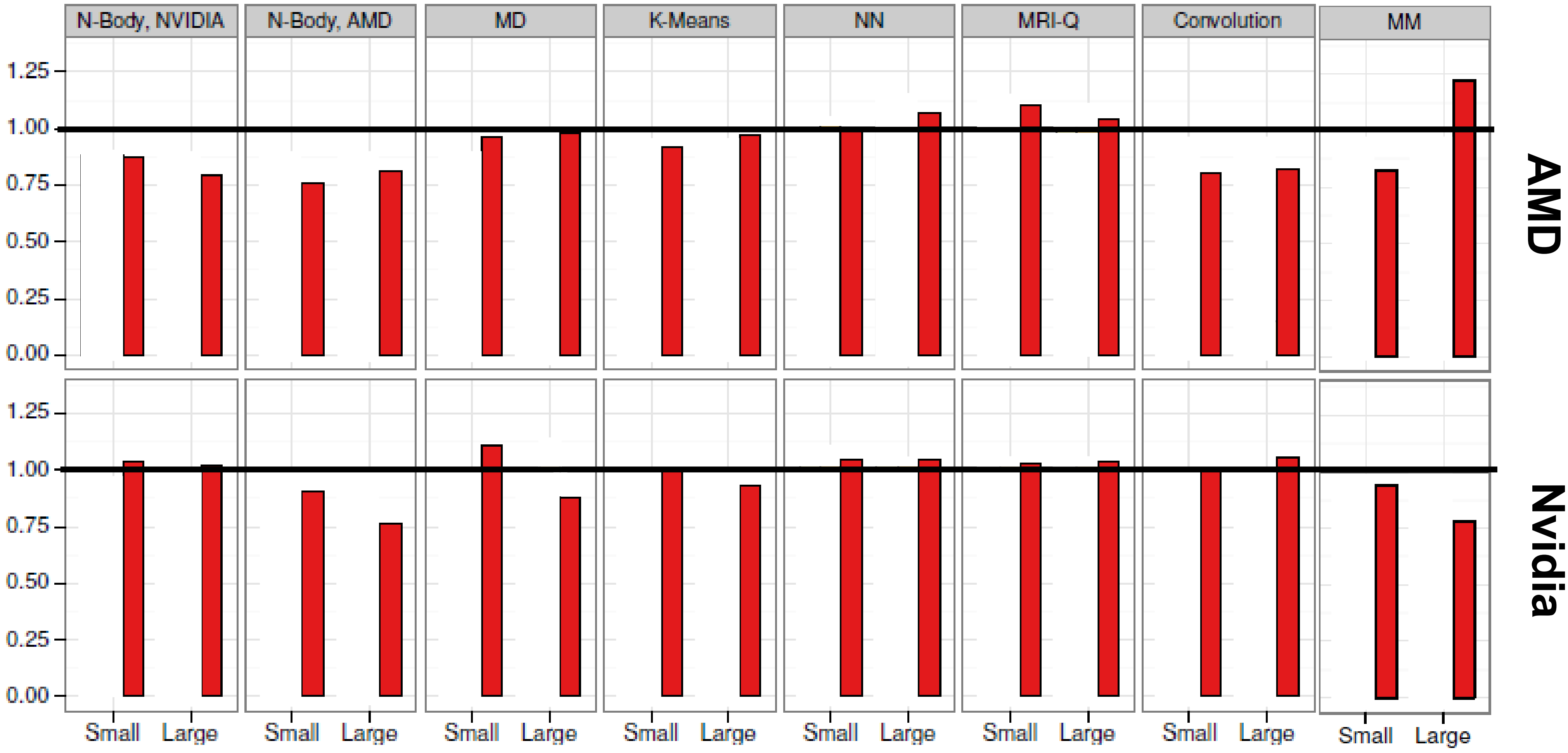
```
zip(x,y) >> mapSeq(mult4) >> reduceSeq(z, add4) → dot(x, y)
```



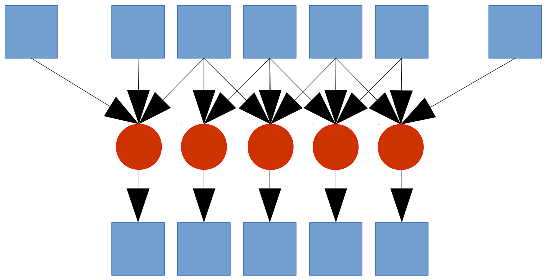

**No need to modify original application!**

**=> Performance portable**

# Works for other programs too



# Stencil Computation

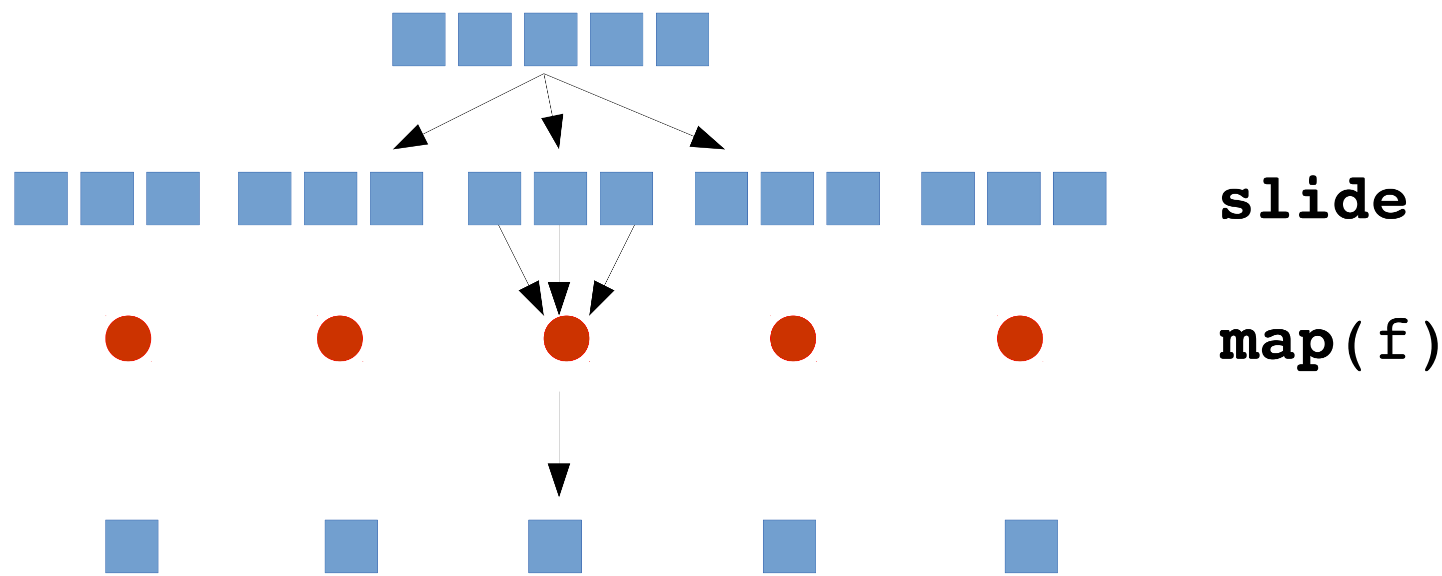


- important application
  - image processing
  - physic simulations
  - neural networks
  - pde solvers
- large opportunity for parallelism
  - GPUs perfect fit

# Stencil Computation in Lift

- use a “slide” primitive

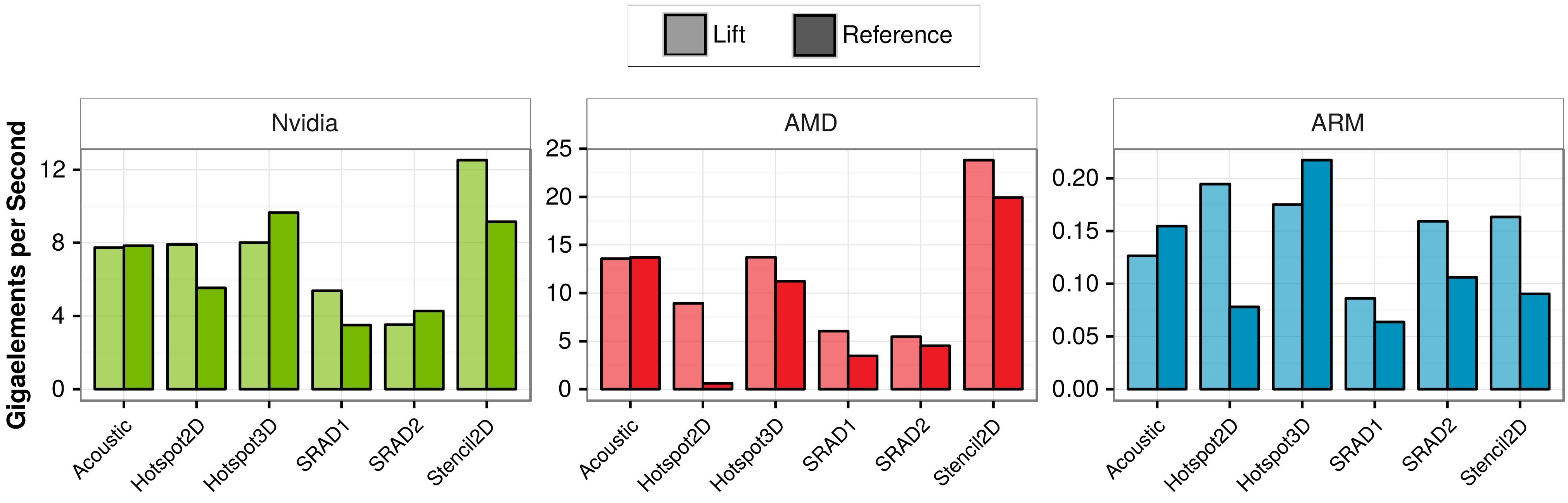
```
stencil = slide(step,size) >> map(f)
```




- can work in 2D too
- leverage all existing rewrites

# Stencil Computation Results

- No conceptual changes in the compiler
  - just two new primitives (slide,pad)
  - one extra rewrite rules



# Summary

- Rewrite rules define a search space
  - formalisation of algorithmic and optimisation choices
- High performance
  - on par with highly-tuned code

# Future Directions

- How to search efficiently the space
  - machine-learning
- Look at other applications
  - e.g. neural network (convolution, recurrent)
- Support for different target programming models
  - MPI, FPGAs, ...

# Purpose of the Tutorial

- Convince you to take up some of the ideas with you
  - functional IR
  - rewriting for optimisation
- Show you what Lift can do
  - OpenCL code generation
  - Rewriting exploration
- Find new potential collaboration
  - FPGA backend
  - MPI work
  - new application domain

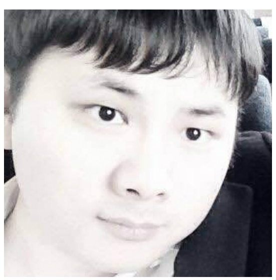
# www.lift-project.org

**Michel Steuwer**
Lectuer
Glasgow Uni.

**Toomas Remmelg**
PhD student
Edinburgh Uni.

**Adam Harris**
PhD student
Edinburgh Uni.

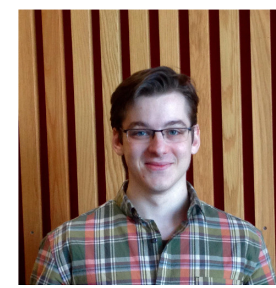
**Bastian Hagedorn**
PhD student
Muenster Uni



**Lu Li**
Postdoc
Edinburgh Uni.

**Federico Pizzuti**
PhD student
Edinburgh Uni.

**Larisa Stolztfus**
PhD student
Edinburgh Uni.



**Naums Mogers**
PhD student
Edinburgh Uni.


supported by:
**Oracle Labs**
**Huawei** Innovation Research Progam FLAGSHIP
**Google** Faculty Rsearch Award
**Microsoft Research**