University of Glasgow

# The LIFT Project

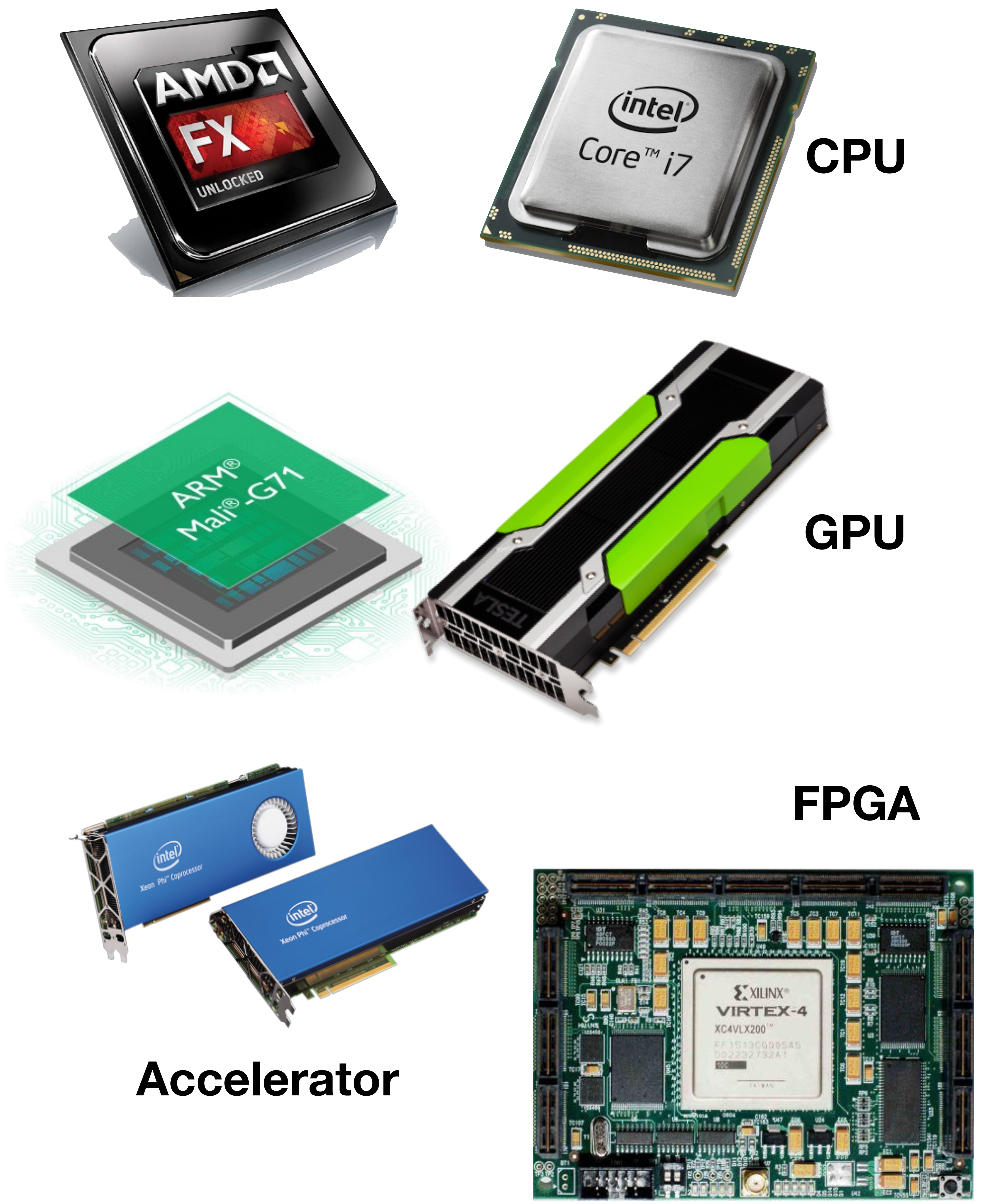
## Performance Portable Parallel Code Generation via Rewrite Rules

Michel Steuwer — *michel.steuwer@glasgow.ac.uk*

INSPIRING PEOPLE

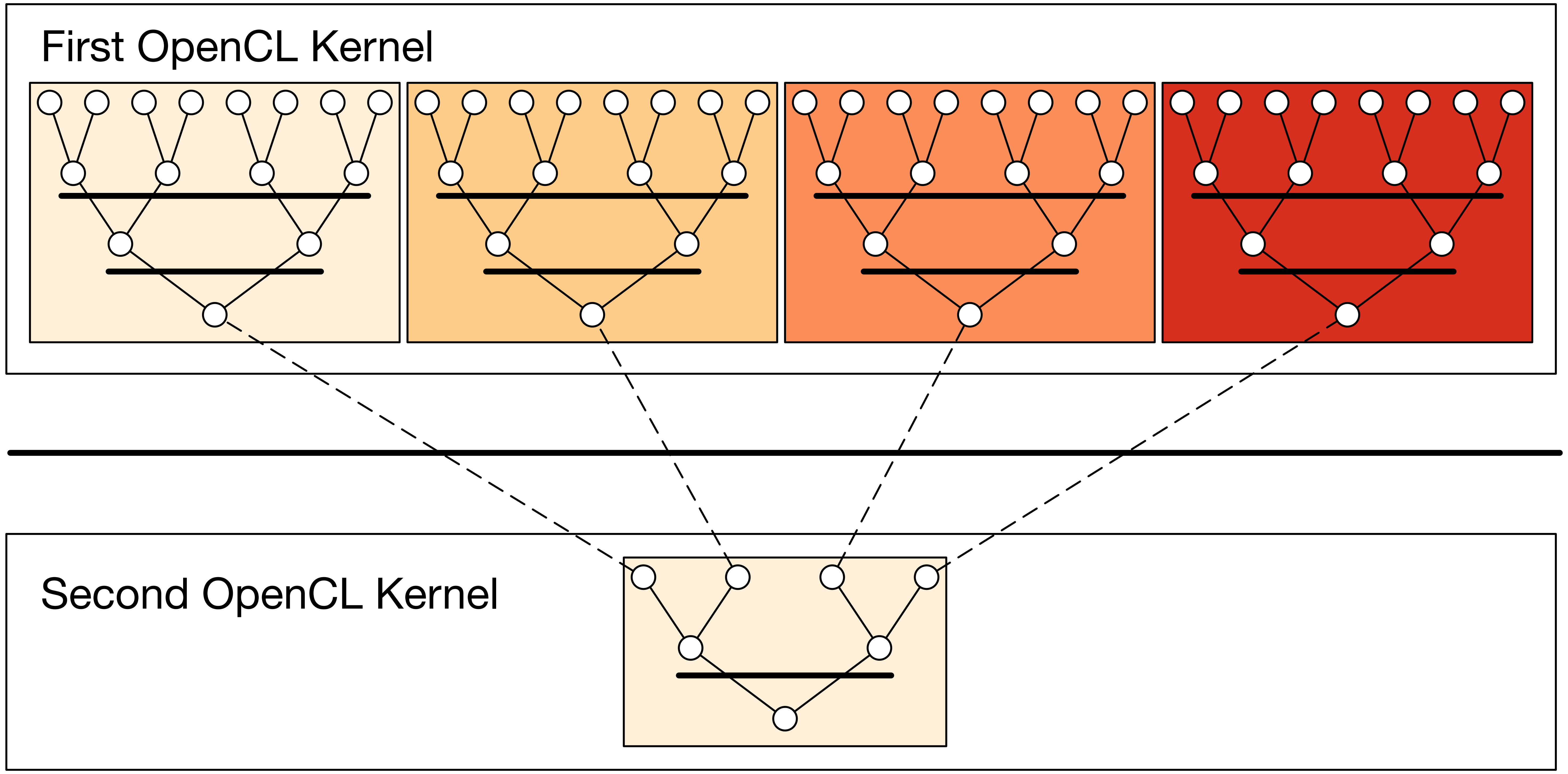
# What are the problems LIFT tries to tackle?

- Parallel processors everywhere
- Many different types: CPUs, GPUs, …
- Parallel programming is hard
- Optimising is even harder
- **Problem**:
  No portability of performance!



# Case Study: Parallel Reduction in OpenCL

- Summing up all values of an array
- Comparison of 7 implementations by Nvidia
- Investigating complexity and efficiency of optimisations



# Parallel reduction with OpenCL

```
kernel void reduce(global float* g_idata, global float* g_odata,
                   unsigned int n, local float* l_data) {
  unsigned int tid = get_local_id(0);
  unsigned int i   = get_global_id(0);
  l_data[tid] = (i < n) ? g_idata[i] : 0;
  barrier(CLK_LOCAL_MEM_FENCE);
  // do reduction in local memory
  for (unsigned int s=1; s < get_local_size(0); s*= 2) {
    if ((tid % (2*s)) == 0) {
      l_data[tid] += l_data[tid + s];
      barrier(CLK_LOCAL_MEM_FENCE);
    }
  }
  // write result for this work-group to global memory
  if (tid == 0) g_odata[get_group_id(0)] = l_data[0];
}
```

# Parallel reduction with OpenCL

**Kernel** function executed in parallel by multiple **work-items**

```
kernel void reduce(global float* g_idata, global float* g_odata,
                   unsigned int n, local float* l_data) {
  unsigned int tid = get_local_id(0);
  unsigned int i   = get_global_id(0);
  l_data[tid] = (i < n) ? g_idata[i] : 0;
  barrier(CLK_LOCAL_MEM_FENCE);
  // do reduction in local memory
  for (unsigned int s=1; s < get_local_size(0); s*= 2) {
    if ((tid % (2*s)) == 0) {
      l_data[tid] += l_data[tid + s];
      barrier(CLK_LOCAL_MEM_FENCE);
    }
  }
  // write result for this work-group to global memory
  if (tid == 0) g_odata[get_group_id(0)] = l_data[0];
}
```

**Work-items** are identified by a unique **global id**

# Parallel reduction with OpenCL

**Work-items** are grouped into **work-groups**

**Local id** within work-group

```
kernel void reduce(global float* g_idata, global float* g_odata,
                   unsigned int n, local float* l_data) {
  unsigned int tid = get_local_id(0);
  unsigned int i   = get_global_id(0);
  l_data[tid] = (i < n) ? g_idata[i] : 0;
  barrier(CLK_LOCAL_MEM_FENCE);
  // do reduction in local memory
  for (unsigned int s=1; s < get_local_size(0); s*= 2) {
    if ((tid % (2*s)) == 0) {
      l_data[tid] += l_data[tid + s];
      barrier(CLK_LOCAL_MEM_FENCE);
    }
  }
  // write result for this work-group to global memory
  if (tid == 0) g_odata[get_group_id(0)] = l_data[0];
}
```

# Parallel reduction with OpenCL

Small, but fast **local** memory

Big, but slow **global** memory

```
kernel void reduce(global float* g_idata, global float* g_odata,
                   unsigned int n, local float* l_data) {
  unsigned int tid = get_local_id(0);
  unsigned int i   = get_global_id(0);
  l_data[tid] = (i < n) ? g_idata[i] : 0;
  barrier(CLK_LOCAL_MEM_FENCE);
  // do reduction in local memory
  for (unsigned int s=1; s < get_local_size(0); s*= 2) {
    if ((tid % (2*s)) == 0) {
      l_data[tid] += l_data[tid + s];
      barrier(CLK_LOCAL_MEM_FENCE);
    }
  }
  // write result for this work-group to global memory
  if (tid == 0) g_odata[get_group_id(0)] = l_data[0];
}
```

Memory **barriers** for consistency

# Parallel reduction with OpenCL

```
kernel void reduce(global float* g_idata, global float* g_odata,
                   unsigned int n, local float* l_data) {
  unsigned int tid = get_local_id(0);
  unsigned int i   = get_global_id(0);
  l_data[tid] = (i < n) ? g_idata[i] : 0;
  barrier(CLK_LOCAL_MEM_FENCE);
  // do reduction in local memory
  for (unsigned int s=1; s < get_local_size(0); s*= 2) {
    if ((tid % (2*s)) == 0) {
      l_data[tid] += l_data[tid + s];
    }
    barrier(CLK_LOCAL_MEM_FENCE);
  }
  // write result for this work-group to global memory
  if (tid == 0) g_odata[get_group_id(0)] = l_data[0];
}
```

Potential **Deadlock**!

**Functionally correct implementations in OpenCL are hard!**

# 1. Version: Unoptimised Implementation Parallel Reduction

```
kernel void reduce0(global float* g_idata, global float* g_odata,
                    unsigned int n, local float* l_data) {
  unsigned int tid = get_local_id(0);
  unsigned int i   = get_global_id(0);
  l_data[tid] = (i < n) ? g_idata[i] : 0;
  barrier(CLK_LOCAL_MEM_FENCE);
  // do reduction in local memory
  for (unsigned int s=1; s < get_local_size(0); s*= 2) {
    if ((tid % (2*s)) == 0) {
      l_data[tid] += l_data[tid + s];
    barrier(CLK_LOCAL_MEM_FENCE);
    }
  }
  // write result for this work-group to global memory
  if (tid == 0) g_odata[get_group_id(0)] = l_data[0];
}
```

# 2. Version: Avoid Divergent Branching

```
kernel void reduce1(global float* g_idata, global float* g_odata,
                    unsigned int n, local float* l_data) {
  unsigned int tid = get_local_id(0);
  unsigned int i   = get_global_id(0);
  l_data[tid] = (i < n) ? g_idata[i] : 0;
  barrier(CLK_LOCAL_MEM_FENCE);

  for (unsigned int s=1; s < get_local_size(0); s*= 2) {
    // continuous work-items remain active
    int index = 2 * s * tid;
    if (index < get_local_size(0)) {
      l_data[index] += l_data[index + s];
    }
    barrier(CLK_LOCAL_MEM_FENCE);
  }
  if (tid == 0) g_odata[get_group_id(0)] = l_data[0];
}
```

# 3. Version: Avoid Interleaved Addressing

```
kernel void reduce2(global float* g_idata, global float* g_odata,
                    unsigned int n, local float* l_data) {
  unsigned int tid = get_local_id(0);
  unsigned int i   = get_global_id(0);
  l_data[tid] = (i < n) ? g_idata[i] : 0;
  barrier(CLK_LOCAL_MEM_FENCE);

  // process elements in different order
  // requires commutativity
  for (unsigned int s=get_local_size(0)/2; s>0; s>>=1) {
    if (tid < s) {
      l_data[tid] += l_data[tid + s];
    }
    barrier(CLK_LOCAL_MEM_FENCE);
  }
  if (tid == 0) g_odata[get_group_id(0)] = l_data[0];
}
```

# 4. Version: Increase Computational Intensity per Work-Item

```
kernel void reduce3(global float* g_idata, global float* g_odata,
                    unsigned int n, local float* l_data) {
  unsigned int tid = get_local_id(0);
  unsigned int i = get_group_id(0) * (get_local_size(0)*2)
                                   + get_local_id(0);
  l_data[tid] = (i < n) ? g_idata[i] : 0;
  // performs first addition during loading
  if (i + get_local_size(0) < n)
    l_data[tid] += g_idata[i+get_local_size(0)];
  barrier(CLK_LOCAL_MEM_FENCE);

  for (unsigned int s=get_local_size(0)/2; s>0; s>>=1) {
    if (tid < s) {
      l_data[tid] += l_data[tid + s];
    }
    barrier(CLK_LOCAL_MEM_FENCE);
  }
  if (tid == 0) g_odata[get_group_id(0)] = l_data[0];
}
```

# 5. Version: Avoid Synchronisation inside a Warp

```
kernel void reduce4(global float* g_idata, global float* g_odata,
                    unsigned int n, local volatile float* l_data) {
  unsigned int tid = get_local_id(0);
  unsigned int i = get_group_id(0) * (get_local_size(0)*2)
                                   + get_local_id(0);
  l_data[tid] = (i < n) ? g_idata[i] : 0;
  if (i + get_local_size(0) < n)
    l_data[tid] += g_idata[i+get_local_size(0)];
  barrier(CLK_LOCAL_MEM_FENCE);

  # pragma unroll 1
  for (unsigned int s=get_local_size(0)/2; s>32; s>>=1) {
    if (tid < s) { l_data[tid] += l_data[tid + s]; }
    barrier(CLK_LOCAL_MEM_FENCE); }

  // this is not portable OpenCL code!
  if (tid < 32) {
    if (WG_SIZE >= 64) { l_data[tid] += l_data[tid+32]; }
    if (WG_SIZE >= 32) { l_data[tid] += l_data[tid+16]; }
    if (WG_SIZE >= 16) { l_data[tid] += l_data[tid+ 8]; }
    if (WG_SIZE >=  8) { l_data[tid] += l_data[tid+ 4]; }
    if (WG_SIZE >=  4) { l_data[tid] += l_data[tid+ 2]; }
    if (WG_SIZE >=  2) { l_data[tid] += l_data[tid+ 1]; } }
  if (tid == 0) g_odata[get_group_id(0)] = l_data[0]; }
```

# 6. Version: Complete Loop Unrolling

```
kernel void reduce5(global float* g_idata, global float* g_odata,
                    unsigned int n, local volatile float* l_data) {
  unsigned int tid = get_local_id(0);
  unsigned int i = get_group_id(0) * (get_local_size(0)*2)
                                    + get_local_id(0);
  l_data[tid] = (i < n) ? g_idata[i] : 0;
  if (i + get_local_size(0) < n)
    l_data[tid] += g_idata[i+get_local_size(0)];
  barrier(CLK_LOCAL_MEM_FENCE);

  if (WG_SIZE >= 256) {
    if (tid < 128) { l_data[tid] += l_data[tid+128]; }
    barrier(CLK_LOCAL_MEM_FENCE); }
  if (WG_SIZE >= 128) {
    if (tid <  64) { l_data[tid] += l_data[tid+ 64]; }
    barrier(CLK_LOCAL_MEM_FENCE); }
  if (tid < 32) {
    if (WG_SIZE >= 64) { l_data[tid] += l_data[tid+32]; }
    if (WG_SIZE >= 32) { l_data[tid] += l_data[tid+16]; }
    if (WG_SIZE >= 16) { l_data[tid] += l_data[tid+ 8]; }
    if (WG_SIZE >=  8) { l_data[tid] += l_data[tid+ 4]; }
    if (WG_SIZE >=  4) { l_data[tid] += l_data[tid+ 2]; }
    if (WG_SIZE >=  2) { l_data[tid] += l_data[tid+ 1]; } }
  if (tid == 0) g_odata[get_group_id(0)] = l_data[0]; }
```

# 7. Version: Fully Optimised Implementation

```
kernel void reduce6(global float* g_idata, global float* g_odata,
                    unsigned int n, local volatile float* l_data) {
  unsigned int tid = get_local_id(0);
  unsigned int i = get_group_id(0) * (get_local_size(0)*2)
                                   + get_local_id(0);
  unsigned int gridSize = WG_SIZE * get_num_groups(0);
  l_data[tid] = 0;
  while (i < n) { l_data[tid] += g_idata[i];
                  if (i + WG_SIZE < n)
                    l_data[tid] += g_idata[i+WG_SIZE];
                  i += gridSize; }
  barrier(CLK_LOCAL_MEM_FENCE);

  if (WG_SIZE >= 256) {
    if (tid < 128) { l_data[tid] += l_data[tid+128]; }
    barrier(CLK_LOCAL_MEM_FENCE); }
  if (WG_SIZE >= 128) {
    if (tid <  64) { l_data[tid] += l_data[tid+ 64]; }
    barrier(CLK_LOCAL_MEM_FENCE); }
  if (tid < 32) {
    if (WG_SIZE >= 64) { l_data[tid] += l_data[tid+32]; }
    if (WG_SIZE >= 32) { l_data[tid] += l_data[tid+16]; }
    if (WG_SIZE >= 16) { l_data[tid] += l_data[tid+ 8]; }
    if (WG_SIZE >=  8) { l_data[tid] += l_data[tid+ 4]; }
    if (WG_SIZE >=  4) { l_data[tid] += l_data[tid+ 2]; }
    if (WG_SIZE >=  2) { l_data[tid] += l_data[tid+ 1]; } }
  if (tid == 0) g_odata[get_group_id(0)] = l_data[0]; }
```

# Reduction Case Study Conclusions

- Optimising OpenCL is complex
  - Understanding of target hardware required
- Program changes not obvious
- Is it worth it? …

```
kernel
void reduce0(global float* g_idata,
             global float* g_odata,
             unsigned int n,
             local float* l_data) {
  unsigned int tid = get_local_id(0);
  unsigned int i   = get_global_id(0);
  l_data[tid] = (i < n) ? g_idata[i] : 0;
  barrier(CLK_LOCAL_MEM_FENCE);

  for (unsigned int s=1;
       s < get_local_size(0); s*= 2) {
    if ((tid % (2*s)) == 0) {
      l_data[tid] += l_data[tid + s];
    }
    barrier(CLK_LOCAL_MEM_FENCE);
  }
  if (tid == 0)
    g_odata[get_group_id(0)] = l_data[0];
}
```

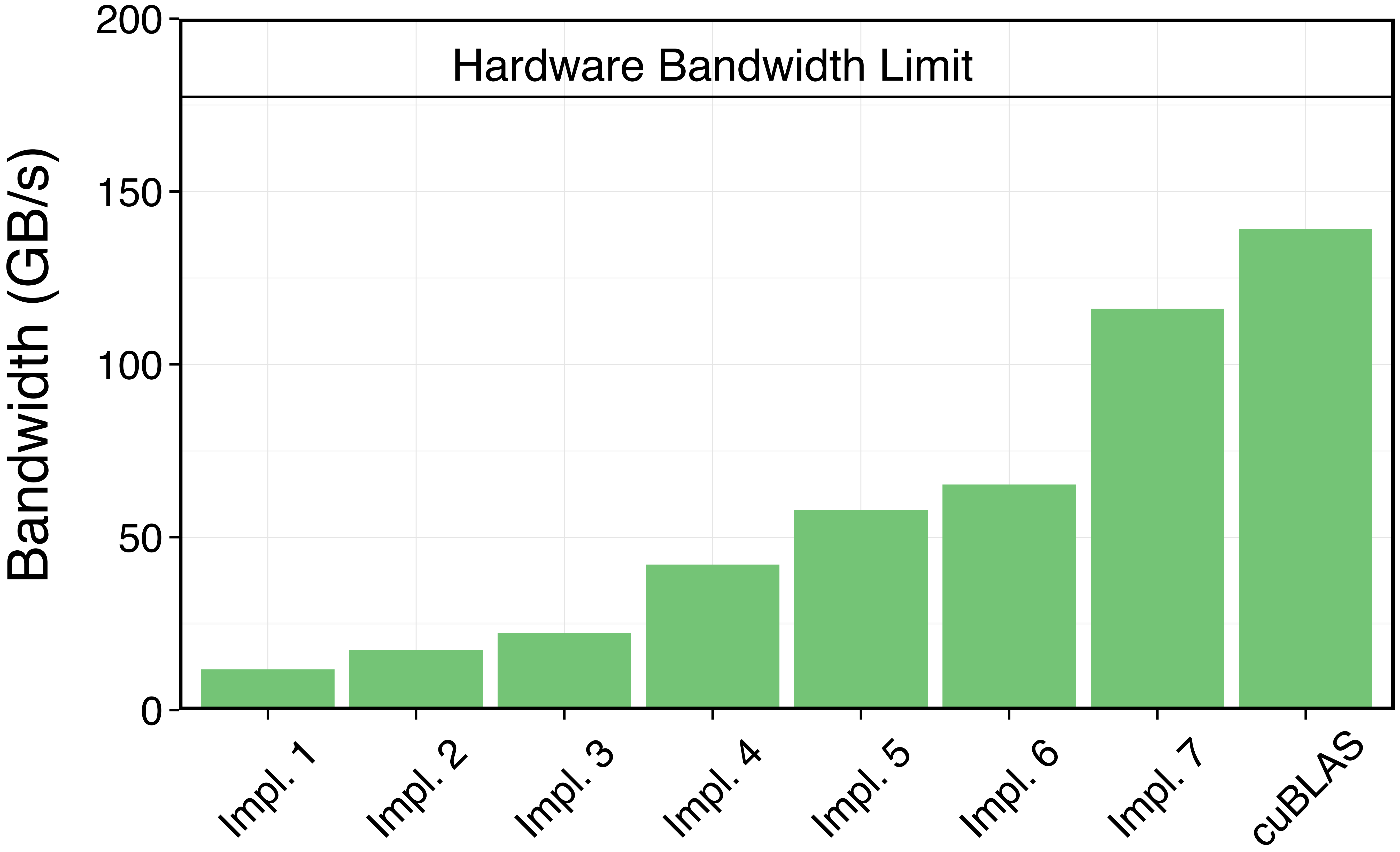
Unoptimized Implementation

```
kernel
void reduce6(global float* g_idata,
             global float* g_odata,
             unsigned int n,
             local volatile float* l_data) {
  unsigned int tid = get_local_id(0);
  unsigned int i =
    get_group_id(0) * (get_local_size(0)*2)
                    + get_local_id(0);
  unsigned int gridSize =
    WG_SIZE * get_num_groups(0);
  l_data[tid] = 0;
  while (i < n) {
    l_data[tid] += g_idata[i];
    if (i + WG_SIZE < n)
      l_data[tid] += g_idata[i+WG_SIZE];
    i += gridSize; }
  barrier(CLK_LOCAL_MEM_FENCE);

  if (WG_SIZE >= 256) {
    if (tid < 128) {
      l_data[tid] += l_data[tid+128]; }
    barrier(CLK_LOCAL_MEM_FENCE); }
  if (WG_SIZE >= 128) {
    if (tid <  64) {
      l_data[tid] += l_data[tid+ 64]; }
    barrier(CLK_LOCAL_MEM_FENCE); }
  if (tid < 32) {
    if (WG_SIZE >= 64) {
      l_data[tid] += l_data[tid+32]; }
    if (WG_SIZE >= 32) {
      l_data[tid] += l_data[tid+16]; }
    if (WG_SIZE >= 16) {
      l_data[tid] += l_data[tid+ 8]; }
    if (WG_SIZE >=  8) {
      l_data[tid] += l_data[tid+ 4]; }
    if (WG_SIZE >=  4) {
      l_data[tid] += l_data[tid+ 2]; }
    if (WG_SIZE >=  2) {
      l_data[tid] += l_data[tid+ 1]; } }
  if (tid == 0)
    g_odata[get_group_id(0)] = l_data[0];
}
```

Fully Optimized Implementation

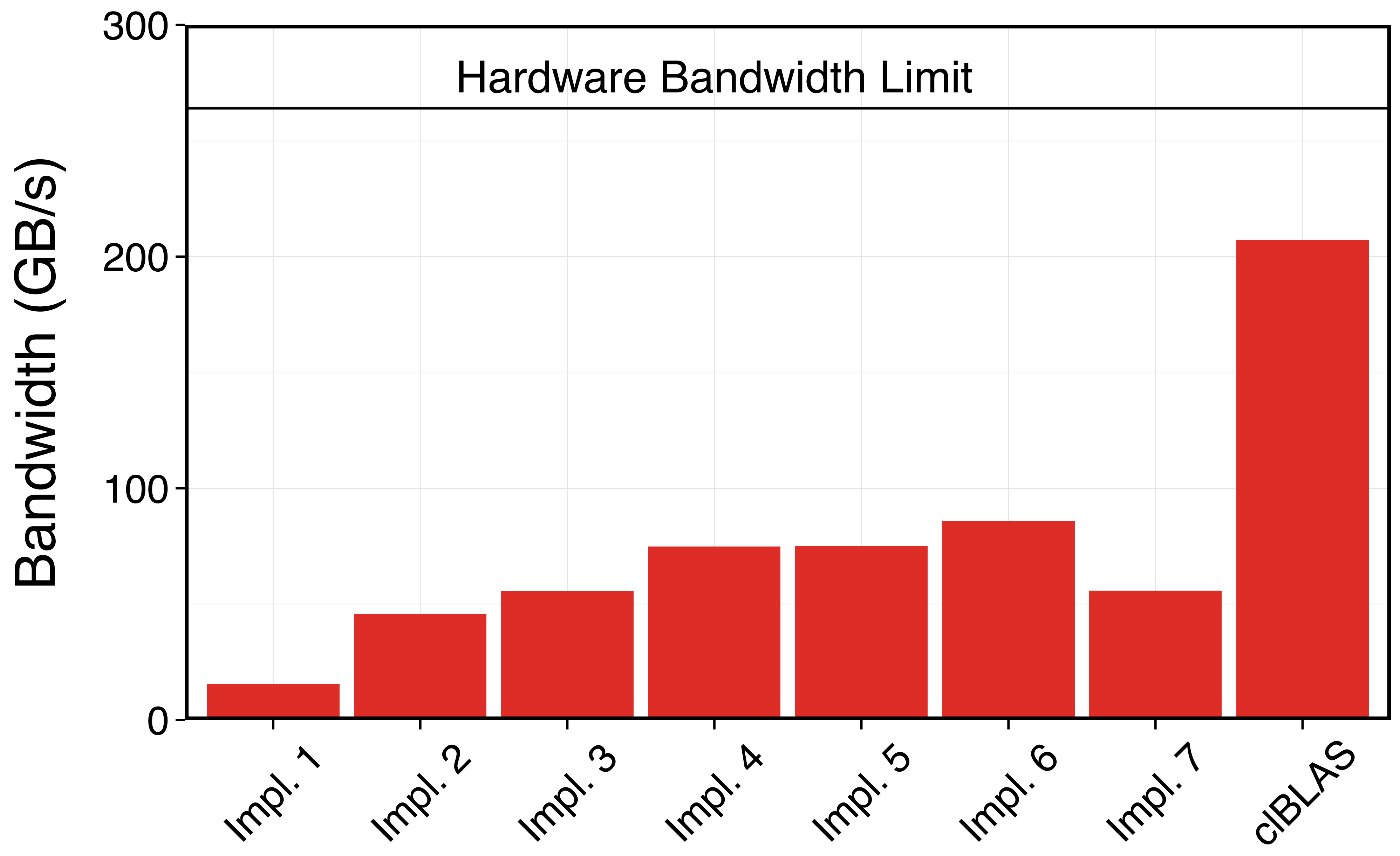
# Performance Results Nvidia



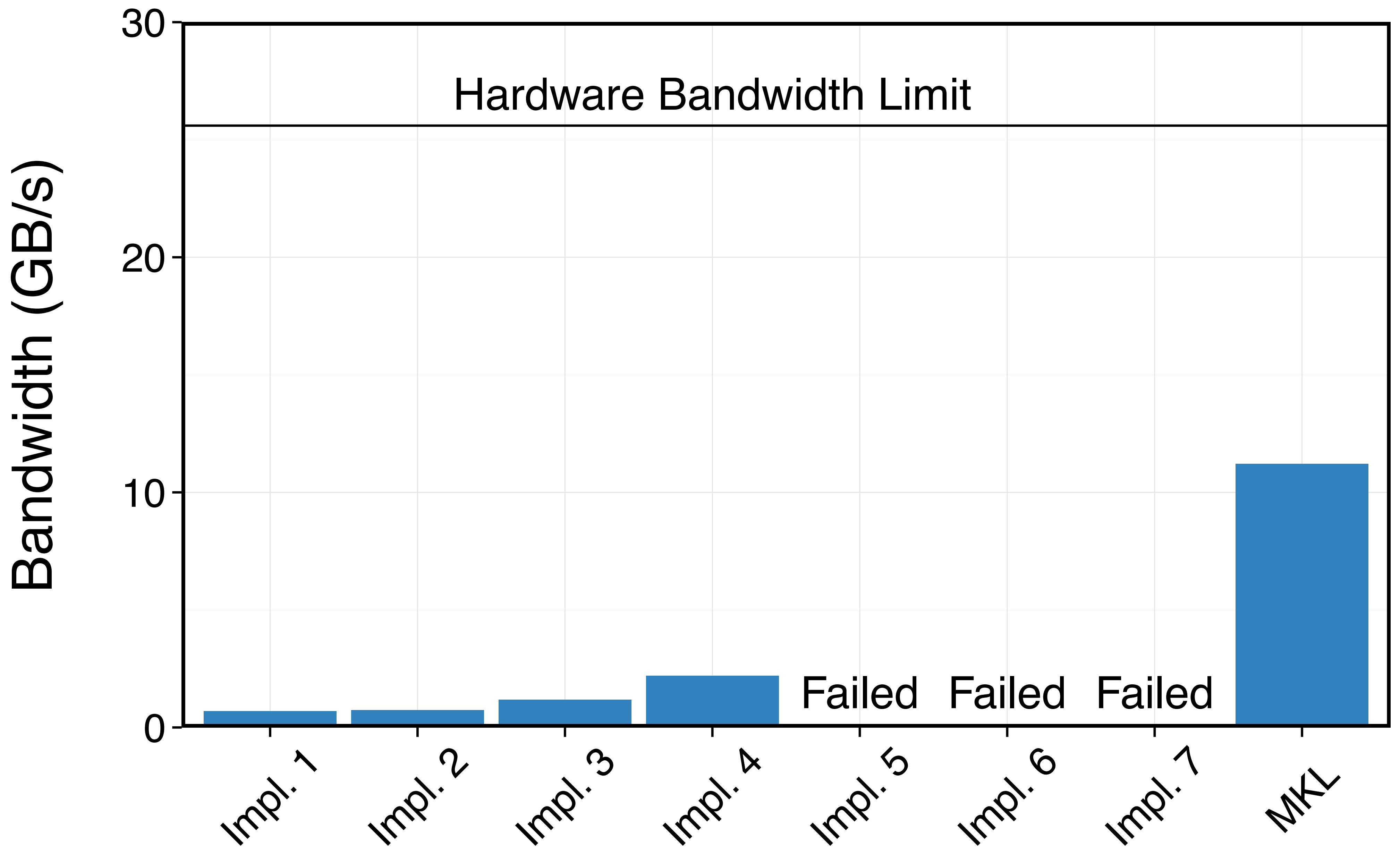

(a) Nvidia's GTX 480 GPU.

- … Yes! Optimising improves performance by a factor of **10**!
- Optimising is important, but …

# Performance Results AMD and Intel



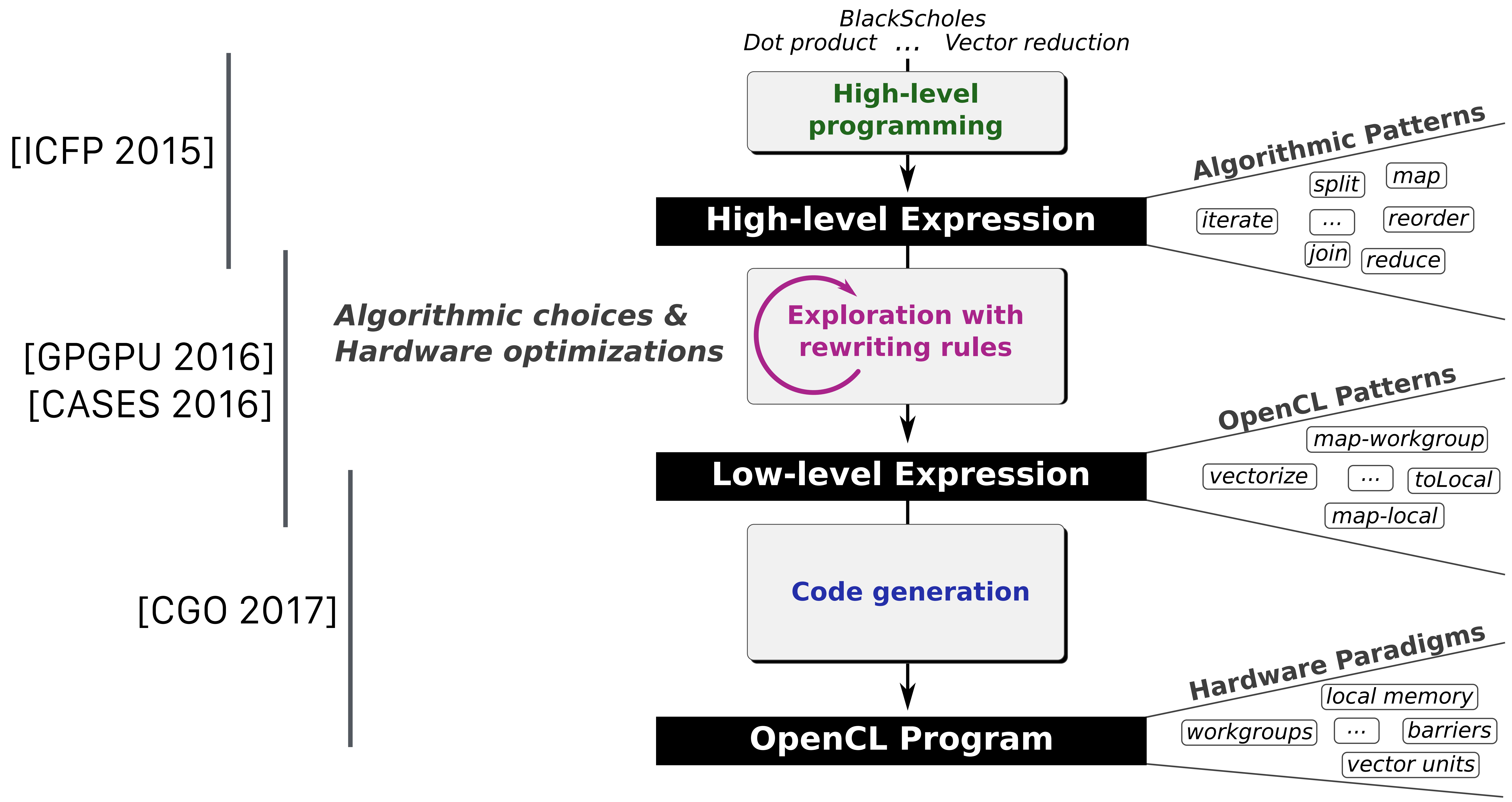

(b) AMD's HD 7970 GPU.




(c) Intel's E5530 dual-socket CPU.

- … unfortunately, optimisations in OpenCL are not portable!
- **Challenge**: how to achieving portable performance?

# LIFT: Performance Portable GPU Code Generation via Rewrite Rules




**Ambition**: automatic generation of *Performance Portable* code

# Walkthrough

① $sum(vec) = reduce(+, 0, vec)$

rewrite rules

code generation

②

$vecSum = reduce \circ join \circ map\text{-}workgroup\ ($
$\quad join \circ toGlobal\ (map\text{-}local\ (map\text{-}seq\ id)) \circ split\ 1 \circ$
$\quad join \circ map\text{-}warp\ ($
$\quad\quad join \circ map\text{-}lane\ (reduce\text{-}seq\ (+)\ 0) \circ split\ 2\ \circ reorder\text{-}stride\ 1 \circ$
$\quad\quad join \circ map\text{-}lane\ (reduce\text{-}seq\ (+)\ 0) \circ split\ 2\ \circ reorder\text{-}stride\ 2 \circ$
$\quad\quad join \circ map\text{-}lane\ (reduce\text{-}seq\ (+)\ 0) \circ split\ 2\ \circ reorder\text{-}stride\ 4 \circ$
$\quad\quad join \circ map\text{-}lane\ (reduce\text{-}seq\ (+)\ 0) \circ split\ 2\ \circ reorder\text{-}stride\ 8 \circ$
$\quad\quad join \circ map\text{-}lane\ (reduce\text{-}seq\ (+)\ 0) \circ split\ 2\ \circ reorder\text{-}stride\ 16 \circ$
$\quad\quad join \circ map\text{-}lane\ (reduce\text{-}seq\ (+)\ 0) \circ split\ 2\ \circ reorder\text{-}stride\ 32$
$\quad ) \circ split\ 64 \circ$
$\quad join \circ map\text{-}local\ (reduce\text{-}seq\ (+)\ 0) \circ split\ 2\ \circ reorder\text{-}stride\ 64 \circ$
$\quad join \circ toLocal\ (map\text{-}local\ (reduce\text{-}seq\ (+)\ 0)) \circ$
$\quad split\ (blockSize/128)\ \circ reorder\text{-}stride\ 128$
$) \circ split\ blockSize$

③

```
kernel
void reduce6(global float* g_idata,
             global float* g_odata,
             unsigned int n,
             local volatile float* l_data) {
  unsigned int tid = get_local_id(0);
  unsigned int i =
    get_group_id(0) * (get_local_size(0)*2)
                    + get_local_id(0);
  unsigned int gridSize =
    WG_SIZE * get_num_groups(0);
  l_data[tid] = 0;
  while (i < n) {
    l_data[tid] += g_idata[i];
    if (i + WG_SIZE < n)
      l_data[tid] += g_idata[i+WG_SIZE];
    i += gridSize; }
  barrier(CLK_LOCAL_MEM_FENCE);

  if (WG_SIZE >= 256) {
    if (tid < 128) {
      l_data[tid] += l_data[tid+128]; }
    barrier(CLK_LOCAL_MEM_FENCE); }
  if (WG_SIZE >= 128) {
    if (tid <  64) {
      l_data[tid] += l_data[tid+ 64]; }
    barrier(CLK_LOCAL_MEM_FENCE); }
  if (tid < 32) {
    if (WG_SIZE >= 64) {
      l_data[tid] += l_data[tid+32]; }
    if (WG_SIZE >= 32) {
      l_data[tid] += l_data[tid+16]; }
    if (WG_SIZE >= 16) {
      l_data[tid] += l_data[tid+ 8]; }
    if (WG_SIZE >=  8) {
      l_data[tid] += l_data[tid+ 4]; }
    if (WG_SIZE >=  4) {
      l_data[tid] += l_data[tid+ 2]; }
    if (WG_SIZE >=  2) {
      l_data[tid] += l_data[tid+ 1]; } }
  if (tid == 0)
    g_odata[get_group_id(0)] = l_data[0];
}
```

# Walkthrough

① $sum(vec) = reduce(+, 0, vec)$

rewrite rules

code generation

②

$vecSum = reduce \circ join \circ map\text{-}workgroup\ ($
$join \circ toGlobal\ (map\text{-}local\ (map\text{-}seq\ id)) \circ split\ 1\ \circ$
$join \circ map\text{-}warp\ ($
$join \circ map\text{-}lane\ (reduce\text{-}seq\ (+)\ 0) \circ split\ 2\ \circ reorder\text{-}stride\ 1\ \circ$
$join \circ map\text{-}lane\ (reduce\text{-}seq\ (+)\ 0) \circ split\ 2\ \circ reorder\text{-}stride\ 2\ \circ$
$join \circ map\text{-}lane\ (reduce\text{-}seq\ (+)\ 0) \circ split\ 2\ \circ reorder\text{-}stride\ 4\ \circ$
$join \circ map\text{-}lane\ (reduce\text{-}seq\ (+)\ 0) \circ split\ 2\ \circ reorder\text{-}stride\ 8\ \circ$
$join \circ map\text{-}lane\ (reduce\text{-}seq\ (+)\ 0) \circ split\ 2\ \circ reorder\text{-}stride\ 16\ \circ$
$join \circ map\text{-}lane\ (reduce\text{-}seq\ (+)\ 0) \circ split\ 2\ \circ reorder\text{-}stride\ 32$
$) \circ split\ 64\ \circ$
$join \circ map\text{-}local\ (reduce\text{-}seq\ (+)\ 0) \circ split\ 2\ \circ reorder\text{-}stride\ 64\ \circ$
$join \circ toLocal\ (map\text{-}local\ (reduce\text{-}seq\ (+)\ 0))\ \circ$
$split\ (blockSize/128)\ \circ reorder\text{-}stride\ 128$
$) \circ split\ blockSize$

③

```
kernel
void reduce6(global float* g_idata,
             global float* g_odata,
             unsigned int n,
             local volatile float* l_data) {
  unsigned int tid = get_local_id(0);
  unsigned int i =
    get_group_id(0) * (get_local_size(0)*2)
                    + get_local_id(0);
  unsigned int gridSize =
    WG_SIZE * get_num_groups(0);
  l_data[tid] = 0;
  while (i < n) {
    l_data[tid] += g_idata[i];
    if (i + WG_SIZE < n)
      l_data[tid] += g_idata[i+WG_SIZE];
    i += gridSize; }
  barrier(CLK_LOCAL_MEM_FENCE);

  if (WG_SIZE >= 256) {
    if (tid < 128) {
      l_data[tid] += l_data[tid+128]; }
    barrier(CLK_LOCAL_MEM_FENCE); }
  if (WG_SIZE >= 128) {
    if (tid <  64) {
      l_data[tid] += l_data[tid+ 64]; }
    barrier(CLK_LOCAL_MEM_FENCE); }
  if (tid < 32) {
    if (WG_SIZE >= 64) {
      l_data[tid] += l_data[tid+32]; }
    if (WG_SIZE >= 32) {
      l_data[tid] += l_data[tid+16]; }
    if (WG_SIZE >= 16) {
      l_data[tid] += l_data[tid+ 8]; }
    if (WG_SIZE >=  8) {
      l_data[tid] += l_data[tid+ 4]; }
    if (WG_SIZE >=  4) {
      l_data[tid] += l_data[tid+ 2]; }
    if (WG_SIZE >=  2) {
      l_data[tid] += l_data[tid+ 1]; } }
  if (tid == 0)
    g_odata[get_group_id(0)] = l_data[0];
}
```

# ① Algorithmic Primitives (a.k.a. algorithmic skeletons)

| | | | |
|---|---|---|---|
| map(f, x): | $[x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8]$ | $\longmapsto$ | $[f(x_1), f(x_2), f(x_3), f(x_4), f(x_5), f(x_6), f(x_7), f(x_8)]$ |
| zip(x, y): | $[x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8]$ <br> $[y_1, y_2, y_3, y_4, y_5, y_6, y_7, y_8]$ | $\longmapsto$ | $[(x_1, y_1), (x_2, y_2), (x_3, y_3), (x_4, y_4), (x_5, y_5), (x_6, y_6), (x_7, y_7), (x_8, y_8)]$ |
| reduce(+, 0, x): | $[x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8]$ | $\longmapsto$ | $[x_1+x_2+x_3+x_4+x_5+x_6+x_7+x_8]$ |
| split(n, x): | $[x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8]$ | $\longmapsto$ | $[[x_1, x_2], [x_3, x_4], [x_5, x_6], [x_7, x_8]]$ |
| join(x): | $[[x_1, x_2], [x_3, x_4], [x_5, x_6], [x_7, x_8]]$ | $\longmapsto$ | $[x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8]$ |
| iterate(f, n, x): | $[x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8]$ | $\longmapsto$ | $f(\ \ldots\ f([x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8])\ldots)$ |
| reorder(σ, x): | $[x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8]$ | $\longmapsto$ | $[x_{\sigma(1)}, x_{\sigma(2)}, x_{\sigma(3)}, x_{\sigma(4)}, x_{\sigma(5)}, x_{\sigma(6)}, x_{\sigma(7)}, x_{\sigma(8)}]$ |

# ① High-Level Programs

```
scal(a, vec) = map(λ x ↦ x*a, vec)

asum(vec) = reduce(+, 0, map(abs, vec))

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

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

# Walkthrough

① $sum(vec) = reduce(+, 0, vec)$

rewrite rules

code generation

②

$vecSum = reduce \circ join \circ map\text{-}workgroup\ ($
$\quad join \circ toGlobal\ (map\text{-}local\ (map\text{-}seq\ id)) \circ split\ 1 \circ$
$\quad join \circ map\text{-}warp\ ($
$\quad\quad join \circ map\text{-}lane\ (reduce\text{-}seq\ (+)\ 0) \circ split\ 2\ \circ reorder\text{-}stride\ 1 \circ$
$\quad\quad join \circ map\text{-}lane\ (reduce\text{-}seq\ (+)\ 0) \circ split\ 2\ \circ reorder\text{-}stride\ 2 \circ$
$\quad\quad join \circ map\text{-}lane\ (reduce\text{-}seq\ (+)\ 0) \circ split\ 2\ \circ reorder\text{-}stride\ 4 \circ$
$\quad\quad join \circ map\text{-}lane\ (reduce\text{-}seq\ (+)\ 0) \circ split\ 2\ \circ reorder\text{-}stride\ 8 \circ$
$\quad\quad join \circ map\text{-}lane\ (reduce\text{-}seq\ (+)\ 0) \circ split\ 2\ \circ reorder\text{-}stride\ 16 \circ$
$\quad\quad join \circ map\text{-}lane\ (reduce\text{-}seq\ (+)\ 0) \circ split\ 2\ \circ reorder\text{-}stride\ 32$
$\quad ) \circ split\ 64 \circ$
$\quad join \circ map\text{-}local\ (reduce\text{-}seq\ (+)\ 0) \circ split\ 2\ \circ reorder\text{-}stride\ 64 \circ$
$\quad join \circ toLocal\ (map\text{-}local\ (reduce\text{-}seq\ (+)\ 0)) \circ$
$\quad split\ (blockSize/128)\ \circ reorder\text{-}stride\ 128$
$) \circ split\ blockSize$

③

```
kernel
void reduce6(global float* g_idata,
             global float* g_odata,
             unsigned int n,
             local volatile float* l_data) {
  unsigned int tid = get_local_id(0);
  unsigned int i =
    get_group_id(0) * (get_local_size(0)*2)
                    + get_local_id(0);
  unsigned int gridSize =
    WG_SIZE * get_num_groups(0);
  l_data[tid] = 0;
  while (i < n) {
    l_data[tid] += g_idata[i];
    if (i + WG_SIZE < n)
      l_data[tid] += g_idata[i+WG_SIZE];
    i += gridSize; }
  barrier(CLK_LOCAL_MEM_FENCE);

  if (WG_SIZE >= 256) {
    if (tid < 128) {
      l_data[tid] += l_data[tid+128]; }
    barrier(CLK_LOCAL_MEM_FENCE); }
  if (WG_SIZE >= 128) {
    if (tid <  64) {
      l_data[tid] += l_data[tid+ 64]; }
    barrier(CLK_LOCAL_MEM_FENCE); }
  if (tid < 32) {
    if (WG_SIZE >= 64) {
      l_data[tid] += l_data[tid+32]; }
    if (WG_SIZE >= 32) {
      l_data[tid] += l_data[tid+16]; }
    if (WG_SIZE >= 16) {
      l_data[tid] += l_data[tid+ 8]; }
    if (WG_SIZE >=  8) {
      l_data[tid] += l_data[tid+ 4]; }
    if (WG_SIZE >=  4) {
      l_data[tid] += l_data[tid+ 2]; }
    if (WG_SIZE >=  2) {
      l_data[tid] += l_data[tid+ 1]; } }
  if (tid == 0)
    g_odata[get_group_id(0)] = l_data[0];
}
```

# Walkthrough

① $\text{sum}(vec) = \text{reduce}(+, 0, vec)$

rewrite rules

code generation

②

$vecSum = \textit{reduce} \circ \textit{join} \circ \textit{map-workgroup}\ ($
$\quad \textit{join} \circ \textit{toGlobal}\ (\textit{map-local}\ (\textit{map-seq id})) \circ \textit{split}\ 1 \circ$
$\quad \textit{join} \circ \textit{map-warp}\ ($
$\quad\quad \textit{join} \circ \textit{map-lane}\ (\textit{reduce-seq}\ (+)\ 0) \circ \textit{split}\ 2 \circ \textit{reorder-stride}\ 1 \circ$
$\quad\quad \textit{join} \circ \textit{map-lane}\ (\textit{reduce-seq}\ (+)\ 0) \circ \textit{split}\ 2 \circ \textit{reorder-stride}\ 2 \circ$
$\quad\quad \textit{join} \circ \textit{map-lane}\ (\textit{reduce-seq}\ (+)\ 0) \circ \textit{split}\ 2 \circ \textit{reorder-stride}\ 4 \circ$
$\quad\quad \textit{join} \circ \textit{map-lane}\ (\textit{reduce-seq}\ (+)\ 0) \circ \textit{split}\ 2 \circ \textit{reorder-stride}\ 8 \circ$
$\quad\quad \textit{join} \circ \textit{map-lane}\ (\textit{reduce-seq}\ (+)\ 0) \circ \textit{split}\ 2 \circ \textit{reorder-stride}\ 16 \circ$
$\quad\quad \textit{join} \circ \textit{map-lane}\ (\textit{reduce-seq}\ (+)\ 0) \circ \textit{split}\ 2 \circ \textit{reorder-stride}\ 32$
$\quad ) \circ \textit{split}\ 64 \circ$
$\quad \textit{join} \circ \textit{map-local}\ (\textit{reduce-seq}\ (+)\ 0) \circ \textit{split}\ 2 \circ \textit{reorder-stride}\ 64 \circ$
$\quad \textit{join} \circ \textit{toLocal}\ (\textit{map-local}\ (\textit{reduce-seq}\ (+)\ 0)) \circ$
$\quad \textit{split}\ (blockSize/128) \circ \textit{reorder-stride}\ 128$
$) \circ \textit{split}\ blockSize$

③

```
kernel
void reduce6(global float* g_idata,
             global float* g_odata,
             unsigned int n,
             local volatile float* l_data) {
  unsigned int tid = get_local_id(0);
  unsigned int i =
    get_group_id(0) * (get_local_size(0)*2)
                    + get_local_id(0);
  unsigned int gridSize =
    WG_SIZE * get_num_groups(0);
  l_data[tid] = 0;
  while (i < n) {
    l_data[tid] += g_idata[i];
    if (i + WG_SIZE < n)
      l_data[tid] += g_idata[i+WG_SIZE];
    i += gridSize; }
  barrier(CLK_LOCAL_MEM_FENCE);

  if (WG_SIZE >= 256) {
    if (tid < 128) {
      l_data[tid] += l_data[tid+128]; }
    barrier(CLK_LOCAL_MEM_FENCE); }
  if (WG_SIZE >= 128) {
    if (tid <  64) {
      l_data[tid] += l_data[tid+ 64]; }
    barrier(CLK_LOCAL_MEM_FENCE); }
  if (tid < 32) {
    if (WG_SIZE >= 64) {
      l_data[tid] += l_data[tid+32]; }
    if (WG_SIZE >= 32) {
      l_data[tid] += l_data[tid+16]; }
    if (WG_SIZE >= 16) {
      l_data[tid] += l_data[tid+ 8]; }
    if (WG_SIZE >=  8) {
      l_data[tid] += l_data[tid+ 4]; }
    if (WG_SIZE >=  4) {
      l_data[tid] += l_data[tid+ 2]; }
    if (WG_SIZE >=  2) {
      l_data[tid] += l_data[tid+ 1]; } }
  if (tid == 0)
    g_odata[get_group_id(0)] = l_data[0];
}
```

# ② Algorithmic Rewrite Rules

- **Provably correct** rewrite rules
- Express algorithmic implementation 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:

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

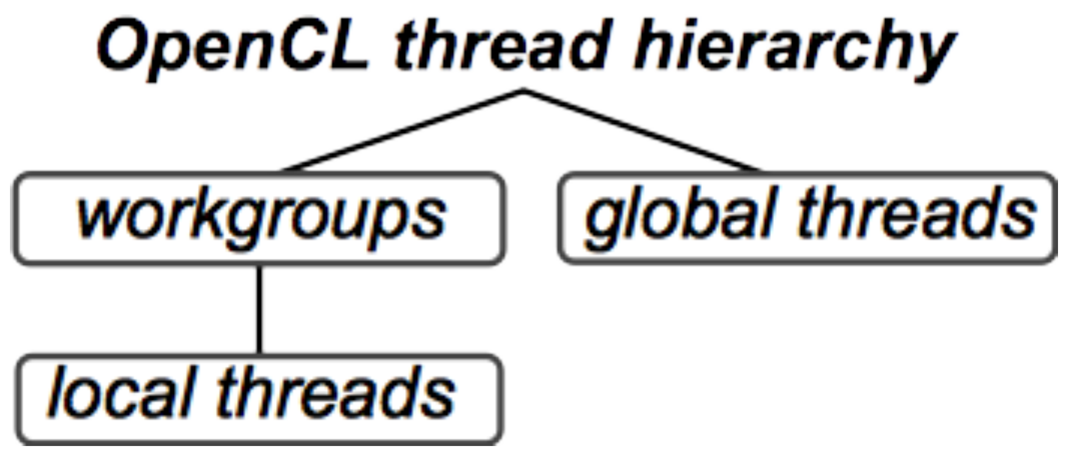
$$reducePart\ f\ \ z \rightarrow reducePart\ f\ \ z \circ reorder$$

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

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

# ② OpenCL Primitives

| **Primitive** | **OpenCL concept** |
| --- | --- |
| $mapGlobal$ | Work-items |
| $mapWorkgroup$<br>$mapLocal$ | Work-groups |
| $mapSeq$<br>$reduceSeq$ | Sequential implementations |
| $toLocal$ , $toGlobal$ | Memory areas |
| $mapVec$,<br>$splitVec$, $joinVec$ | Vectorisation |



# ② OpenCL Rewrite Rules

- Express low-level implementation and optimisation choices

Map rules:

$$map\ f \rightarrow mapWorkgroup\ f \mid mapLocal\ f \mid mapGlobal\ f \mid mapSeq\ f$$

Local/ global memory rules:

$$mapLocal\ f \rightarrow toLocal\ (mapLocal\ f) \qquad mapLocal\ f \rightarrow toGlobal\ (mapLocal\ f)$$

Vectorisation rule:

$$map\ f \rightarrow joinVec \circ map\ (mapVec\ f) \circ splitVec\ n$$

Fusion rule:

$$reduceSeq\ f\ \ z \circ mapSeq\ g \rightarrow reduceSeq\ (\lambda\ (acc, x).\ \ f\ \ (acc, g\ \ x))\ \ z$$

# Walkthrough

① $vecSum = reduce\ (+)\ 0$

rewrite rules ↓ code generation ↗

②

$vecSum = reduce \circ join \circ map\text{-}workgroup\ ($
$\quad join \circ toGlobal\ (map\text{-}local\ (map\text{-}seq\ id)) \circ split\ 1\ \circ$
$\quad join \circ map\text{-}warp\ ($
$\quad\quad join \circ map\text{-}lane\ (reduce\text{-}seq\ (+)\ 0) \circ split\ 2\ \circ reorder\text{-}stride\ 1\ \circ$
$\quad\quad join \circ map\text{-}lane\ (reduce\text{-}seq\ (+)\ 0) \circ split\ 2\ \circ reorder\text{-}stride\ 2\ \circ$
$\quad\quad join \circ map\text{-}lane\ (reduce\text{-}seq\ (+)\ 0) \circ split\ 2\ \circ reorder\text{-}stride\ 4\ \circ$
$\quad\quad join \circ map\text{-}lane\ (reduce\text{-}seq\ (+)\ 0) \circ split\ 2\ \circ reorder\text{-}stride\ 8\ \circ$
$\quad\quad join \circ map\text{-}lane\ (reduce\text{-}seq\ (+)\ 0) \circ split\ 2\ \circ reorder\text{-}stride\ 16\ \circ$
$\quad\quad join \circ map\text{-}lane\ (reduce\text{-}seq\ (+)\ 0) \circ split\ 2\ \circ reorder\text{-}stride\ 32$
$\quad ) \circ split\ 64\ \circ$
$\quad join \circ map\text{-}local\ (reduce\text{-}seq\ (+)\ 0) \circ split\ 2\ \circ reorder\text{-}stride\ 64\ \circ$
$\quad join \circ toLocal\ (map\text{-}local\ (reduce\text{-}seq\ (+)\ 0))\ \circ$
$\quad split\ (blockSize/128)\ \circ reorder\text{-}stride\ 128$
$) \circ split\ blockSize$

③

```
kernel
void reduce6(global float* g_idata,
             global float* g_odata,
             unsigned int n,
             local volatile float* l_data) {
  unsigned int tid = get_local_id(0);
  unsigned int i =
    get_group_id(0) * (get_local_size(0)*2)
                    + get_local_id(0);
  unsigned int gridSize =
    WG_SIZE * get_num_groups(0);
  l_data[tid] = 0;
  while (i < n) {
    l_data[tid] += g_idata[i];
    if (i + WG_SIZE < n)
      l_data[tid] += g_idata[i+WG_SIZE];
    i += gridSize; }
  barrier(CLK_LOCAL_MEM_FENCE);

  if (WG_SIZE >= 256) {
    if (tid < 128) {
      l_data[tid] += l_data[tid+128]; }
    barrier(CLK_LOCAL_MEM_FENCE); }
  if (WG_SIZE >= 128) {
    if (tid <  64) {
      l_data[tid] += l_data[tid+ 64]; }
    barrier(CLK_LOCAL_MEM_FENCE); }
  if (tid < 32) {
    if (WG_SIZE >= 64) {
      l_data[tid] += l_data[tid+32]; }
    if (WG_SIZE >= 32) {
      l_data[tid] += l_data[tid+16]; }
    if (WG_SIZE >= 16) {
      l_data[tid] += l_data[tid+ 8]; }
    if (WG_SIZE >=  8) {
      l_data[tid] += l_data[tid+ 4]; }
    if (WG_SIZE >=  4) {
      l_data[tid] += l_data[tid+ 2]; }
    if (WG_SIZE >=  2) {
      l_data[tid] += l_data[tid+ 1]; } }
  if (tid == 0)
    g_odata[get_group_id(0)] = l_data[0];
}
```

# Walkthrough

① $vecSum = reduce\ (+)\ 0$

rewrite rules

code generation

②

$vecSum = reduce \circ join \circ map\text{-}workgroup\ ($
$\quad join \circ toGlobal\ (map\text{-}local\ (map\text{-}seq\ id)) \circ split\ 1 \circ$
$\quad join \circ map\text{-}warp\ ($
$\quad\quad join \circ map\text{-}lane\ (reduce\text{-}seq\ (+)\ 0) \circ split\ 2\ \circ reorder\text{-}stride\ 1 \circ$
$\quad\quad join \circ map\text{-}lane\ (reduce\text{-}seq\ (+)\ 0) \circ split\ 2\ \circ reorder\text{-}stride\ 2 \circ$
$\quad\quad join \circ map\text{-}lane\ (reduce\text{-}seq\ (+)\ 0) \circ split\ 2\ \circ reorder\text{-}stride\ 4 \circ$
$\quad\quad join \circ map\text{-}lane\ (reduce\text{-}seq\ (+)\ 0) \circ split\ 2\ \circ reorder\text{-}stride\ 8 \circ$
$\quad\quad join \circ map\text{-}lane\ (reduce\text{-}seq\ (+)\ 0) \circ split\ 2\ \circ reorder\text{-}stride\ 16 \circ$
$\quad\quad join \circ map\text{-}lane\ (reduce\text{-}seq\ (+)\ 0) \circ split\ 2\ \circ reorder\text{-}stride\ 32$
$\quad ) \circ split\ 64 \circ$
$\quad join \circ map\text{-}local\ (reduce\text{-}seq\ (+)\ 0) \circ split\ 2\ \circ reorder\text{-}stride\ 64 \circ$
$\quad join \circ toLocal\ (map\text{-}local\ (reduce\text{-}seq\ (+)\ 0)) \circ$
$\quad split\ (blockSize/128)\ \circ reorder\text{-}stride\ 128$
$) \circ split\ blockSize$

③

```
kernel
void reduce6(global float* g_idata,
             global float* g_odata,
             unsigned int n,
             local volatile float* l_data) {
  unsigned int tid = get_local_id(0);
  unsigned int i =
    get_group_id(0) * (get_local_size(0)*2)
                    + get_local_id(0);
  unsigned int gridSize =
    WG_SIZE * get_num_groups(0);
  l_data[tid] = 0;
  while (i < n) {
    l_data[tid] += g_idata[i];
    if (i + WG_SIZE < n)
      l_data[tid] += g_idata[i+WG_SIZE];
    i += gridSize; }
  barrier(CLK_LOCAL_MEM_FENCE);

  if (WG_SIZE >= 256) {
    if (tid < 128) {
      l_data[tid] += l_data[tid+128]; }
    barrier(CLK_LOCAL_MEM_FENCE); }
  if (WG_SIZE >= 128) {
    if (tid <  64) {
      l_data[tid] += l_data[tid+ 64]; }
    barrier(CLK_LOCAL_MEM_FENCE); }
  if (tid < 32) {
    if (WG_SIZE >= 64) {
      l_data[tid] += l_data[tid+32]; }
    if (WG_SIZE >= 32) {
      l_data[tid] += l_data[tid+16]; }
    if (WG_SIZE >= 16) {
      l_data[tid] += l_data[tid+ 8]; }
    if (WG_SIZE >=  8) {
      l_data[tid] += l_data[tid+ 4]; }
    if (WG_SIZE >=  4) {
      l_data[tid] += l_data[tid+ 2]; }
    if (WG_SIZE >=  2) {
      l_data[tid] += l_data[tid+ 1]; } }
  if (tid == 0)
    g_odata[get_group_id(0)] = l_data[0];
}
```

# ③ Pattern based OpenCL Code Generation

- Generate OpenCL code for each OpenCL primitive

$mapGlobal\ f\ xs$ →

```
for (int g_id = get_global_id(0); g_id < n;
      g_id += get_global_size(0)) {
  output[g_id]  = f(xs[g_id]);
}
```

$reduceSeq\ f\ z\ xs$ →

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

⋮ ⋮

- A lot more details about the code generation implementation can be found in our CGO 2017 paper

# Walkthrough

① $vecSum = reduce\ (+)\ 0$

rewrite rules ↓ code generation ↗

②

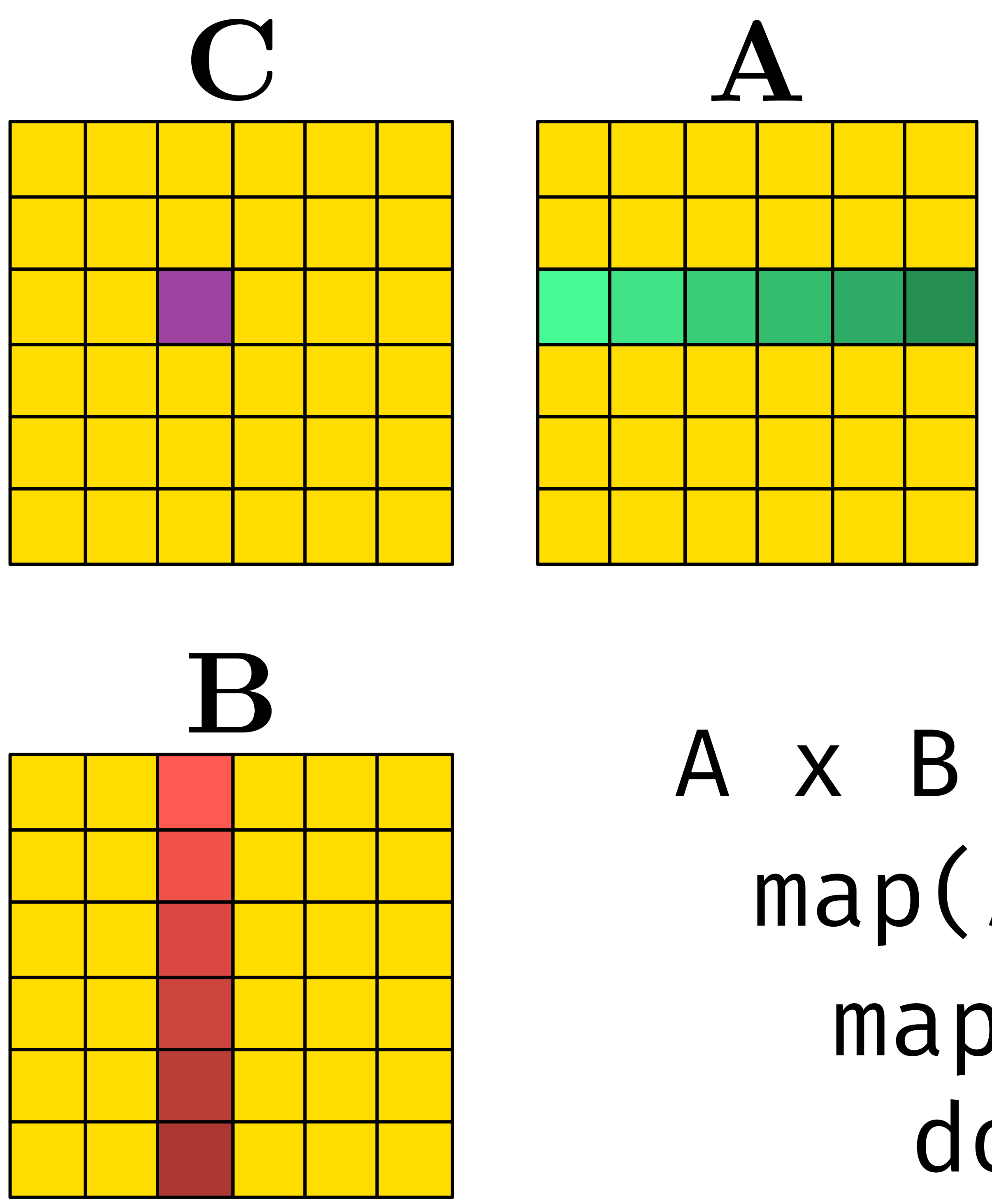
$$\begin{aligned}
&vecSum = reduce \circ join \circ map\text{-}workgroup\ ( \\
&\quad join \circ toGlobal\ (map\text{-}local\ (map\text{-}seq\ id)) \circ split\ 1 \circ \\
&\quad join \circ map\text{-}warp\ ( \\
&\qquad join \circ map\text{-}lane\ (reduce\text{-}seq\ (+)\ 0) \circ split\ 2\ \circ reorder\text{-}stride\ 1 \circ \\
&\qquad join \circ map\text{-}lane\ (reduce\text{-}seq\ (+)\ 0) \circ split\ 2\ \circ reorder\text{-}stride\ 2 \circ \\
&\qquad join \circ map\text{-}lane\ (reduce\text{-}seq\ (+)\ 0) \circ split\ 2\ \circ reorder\text{-}stride\ 4 \circ \\
&\qquad join \circ map\text{-}lane\ (reduce\text{-}seq\ (+)\ 0) \circ split\ 2\ \circ reorder\text{-}stride\ 8 \circ \\
&\qquad join \circ map\text{-}lane\ (reduce\text{-}seq\ (+)\ 0) \circ split\ 2\ \circ reorder\text{-}stride\ 16 \circ \\
&\qquad join \circ map\text{-}lane\ (reduce\text{-}seq\ (+)\ 0) \circ split\ 2\ \circ reorder\text{-}stride\ 32 \\
&\quad ) \circ split\ 64 \circ \\
&\quad join \circ map\text{-}local\ (reduce\text{-}seq\ (+)\ 0) \circ split\ 2\ \circ reorder\text{-}stride\ 64 \circ \\
&\quad join \circ toLocal\ (map\text{-}local\ (reduce\text{-}seq\ (+)\ 0)) \circ \\
&\quad split\ (\mathrm{blockSize}/128)\ \circ reorder\text{-}stride\ 128 \\
&) \circ split\ \mathrm{blockSize}
\end{aligned}$$

③

```
kernel
void reduce6(global float* g_idata,
             global float* g_odata,
             unsigned int n,
             local volatile float* l_data) {
  unsigned int tid = get_local_id(0);
  unsigned int i =
    get_group_id(0) * (get_local_size(0)*2)
                    + get_local_id(0);
  unsigned int gridSize =
    WG_SIZE * get_num_groups(0);
  l_data[tid] = 0;
  while (i < n) {
    l_data[tid] += g_idata[i];
    if (i + WG_SIZE < n)
      l_data[tid] += g_idata[i+WG_SIZE];
    i += gridSize; }
  barrier(CLK_LOCAL_MEM_FENCE);

  if (WG_SIZE >= 256) {
    if (tid < 128) {
      l_data[tid] += l_data[tid+128]; }
    barrier(CLK_LOCAL_MEM_FENCE); }
  if (WG_SIZE >= 128) {
    if (tid <  64) {
      l_data[tid] += l_data[tid+ 64]; }
    barrier(CLK_LOCAL_MEM_FENCE); }
  if (tid < 32) {
    if (WG_SIZE >= 64) {
      l_data[tid] += l_data[tid+32]; }
    if (WG_SIZE >= 32) {
      l_data[tid] += l_data[tid+16]; }
    if (WG_SIZE >= 16) {
      l_data[tid] += l_data[tid+ 8]; }
    if (WG_SIZE >=  8) {
      l_data[tid] += l_data[tid+ 4]; }
    if (WG_SIZE >=  4) {
      l_data[tid] += l_data[tid+ 2]; }
    if (WG_SIZE >=  2) {
      l_data[tid] += l_data[tid+ 1]; } }
  if (tid == 0)
    g_odata[get_group_id(0)] = l_data[0];
}
```

# Case Study: Matrix Multiplication




```
A x B =
  map(λ rowA ↦
    map(λ colB ↦
      dotProduct(rowA, colB)
    , transpose(B))
  , A)
```

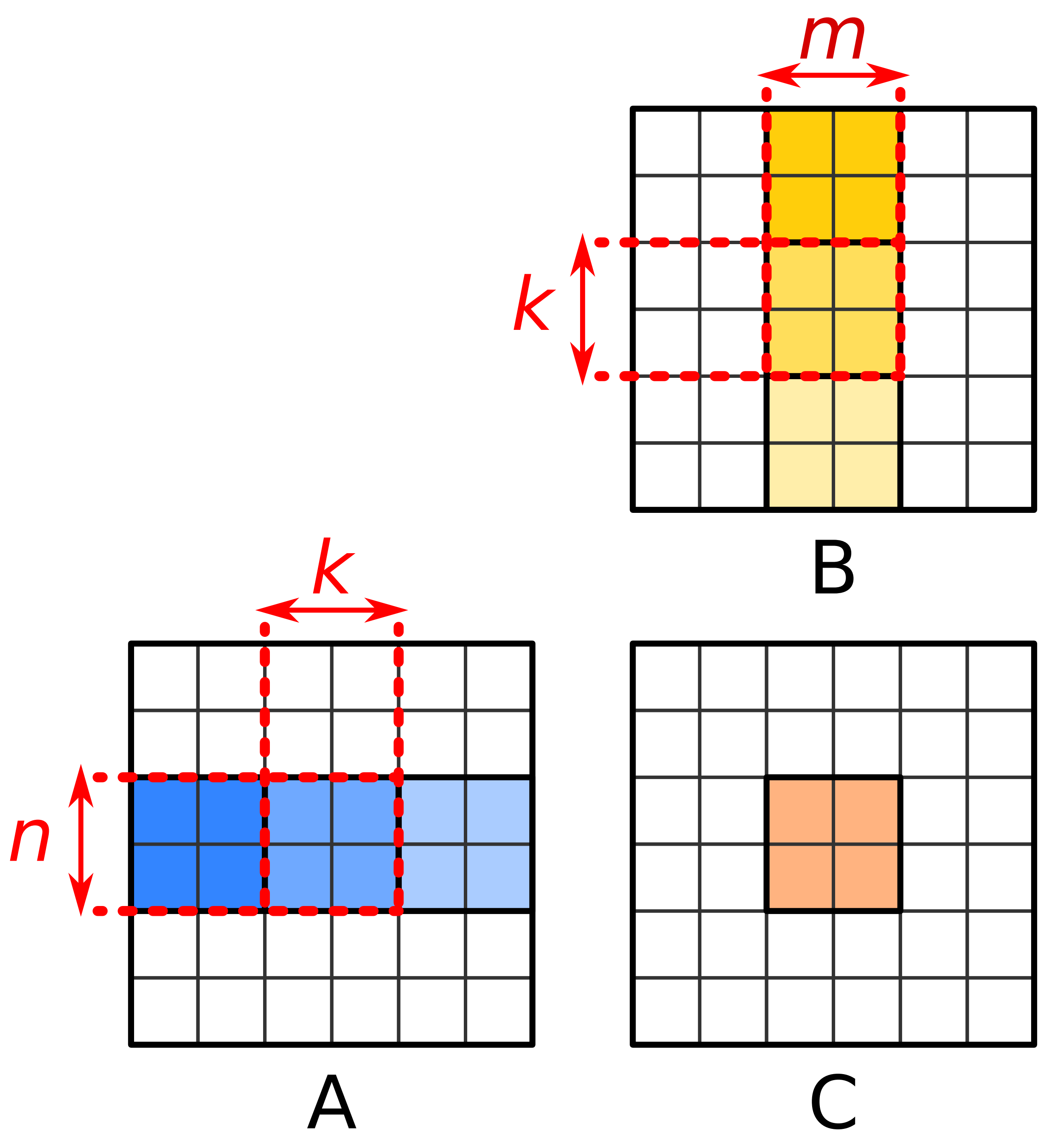
# Tiling as a Rewrite Rules

Naïve matrix multiplication

```
map(λ arow .
 map(λ bcol .
    reduce(+, 0) ∘ map(×) ∘ zip(arow, bcol)
 , transpose(B))
, A)
```

Apply tiling rules

```
untile ∘ map(λ rowOfTilesA .
 map(λ colOfTilesB .
  toGlobal(copy2D) ∘
  reduce(λ (tileAcc, (tileA, tileB)) .
    map(map(+)) ∘ zip(tileAcc) ∘
    map(λ as .
     map(λ bs .
       reduce(+, 0) ∘ map(×) ∘ zip(as, bs)
     , toLocal(copy2D(tileB)))
    , toLocal(copy2D(tileA)))
  ,0, zip(rowOfTilesA, colOfTilesB))
 ) ∘ tile(m, k, transpose(B))
) ∘ tile(n, k, A)
```

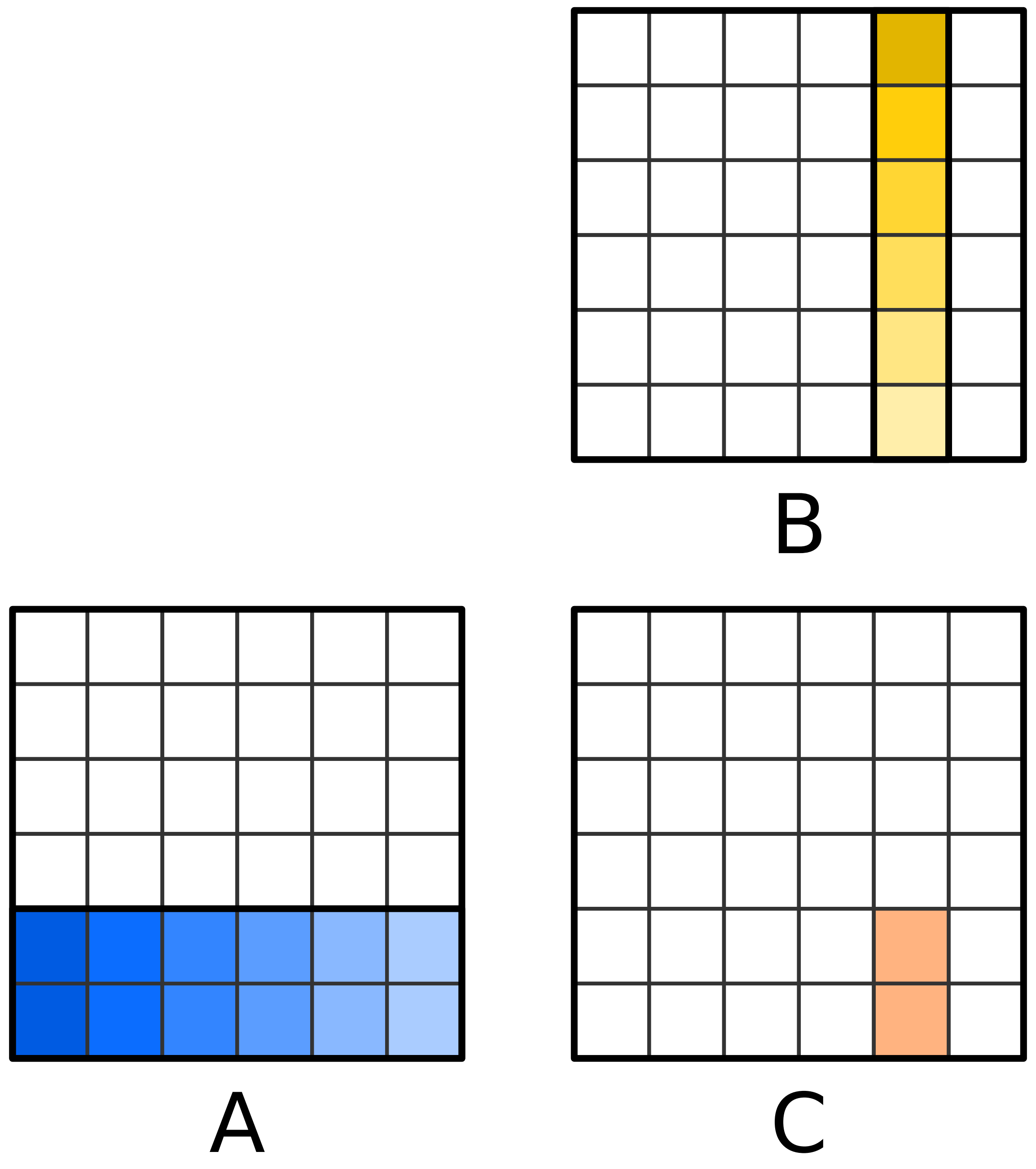


# Register Blocking as a Rewrite Rules

```
untile ∘ map(λ rowOfTilesA .
 map(λ colOfTilesB .
  toGlobal(copy2D) ∘
  reduce(λ (tileAcc, (tileA, tileB)) .
   map(map(+)) ∘ zip(tileAcc) ∘
   map(λ as .
    map(λ bs .
     reduce(+, 0) ∘ map(×) ∘ zip(as, bs)
    , toLocal(copy2D(tileB)))
   , toLocal(copy2D(tileA)))
  ,0, zip(rowOfTilesA, colOfTilesB))
 ) ∘ tile(m, k, transpose(B))
) ∘ tile(n, k, A)
```

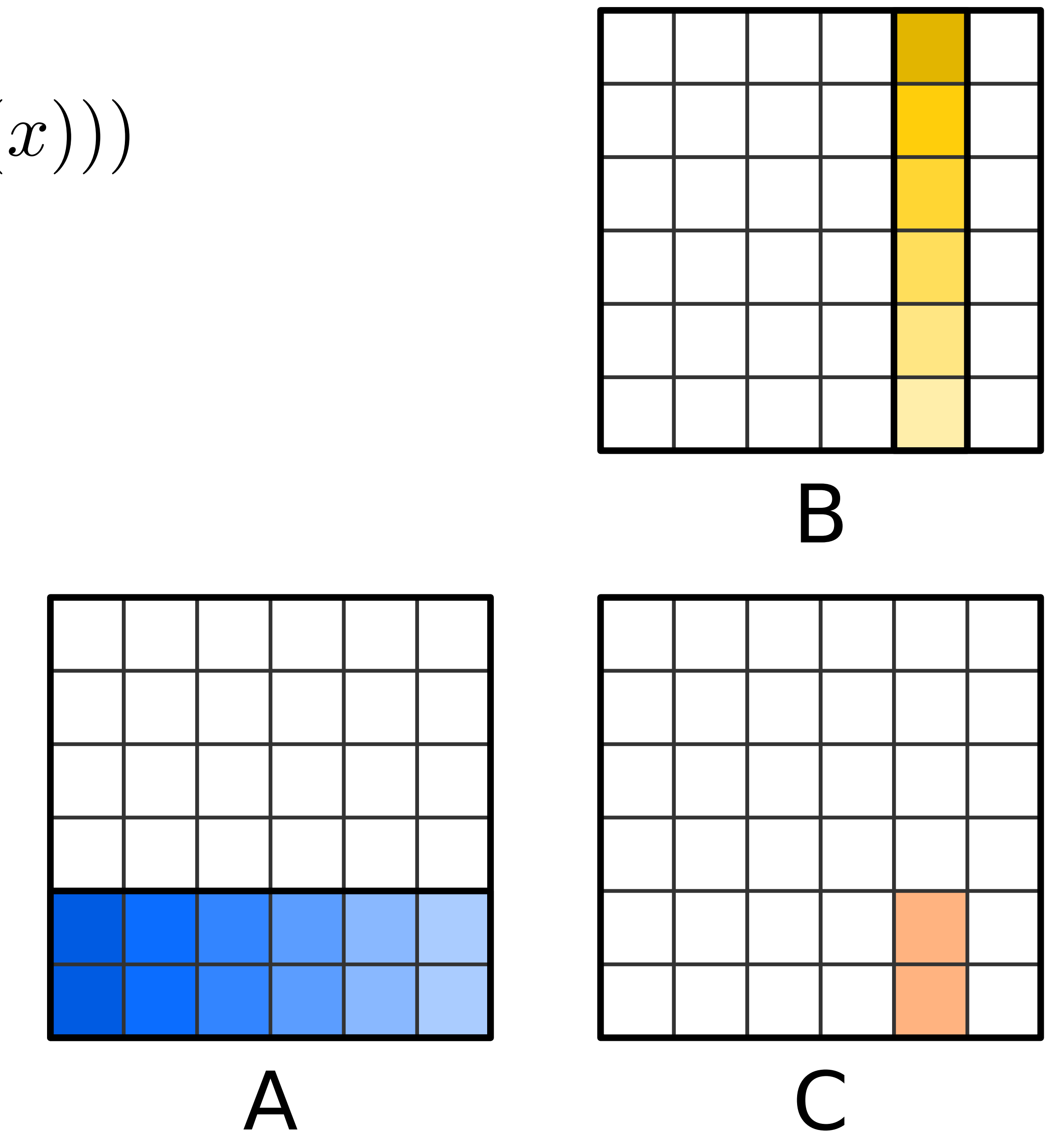
↓ Apply blocking rules

```
untile ∘ map(λ rowOfTilesA .
 map(λ colOfTilesB .
  toGlobal(copy2D) ∘
  reduce(λ (tileAcc, (tileA, tileB)) .
   map(map(+)) ∘ zip(tileAcc) ∘
   map(λ aBlocks .
    map(λ bs .
     reduce(+, 0) ∘
     map(λ (aBlock, b) .
       map(λ (a,bp) . a × bp
       , zip(aBlock, toPrivate(id(b))))
     ) ∘ zip(transpose(aBlocks), bs)
    , toLocal(copy2D(tileB)))
   , split(l, toLocal(copy2D(tileA))))
  ,0, zip(rowOfTilesA, colOfTilesB))
 ) ∘ tile(m, k, transpose(B))
) ∘ tile(n, k, A)
```

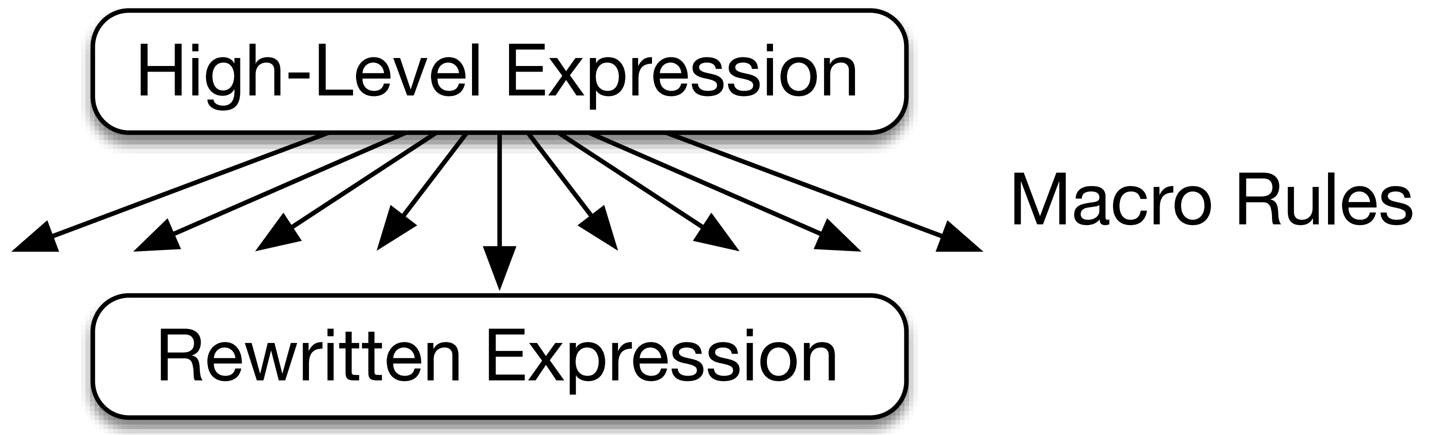


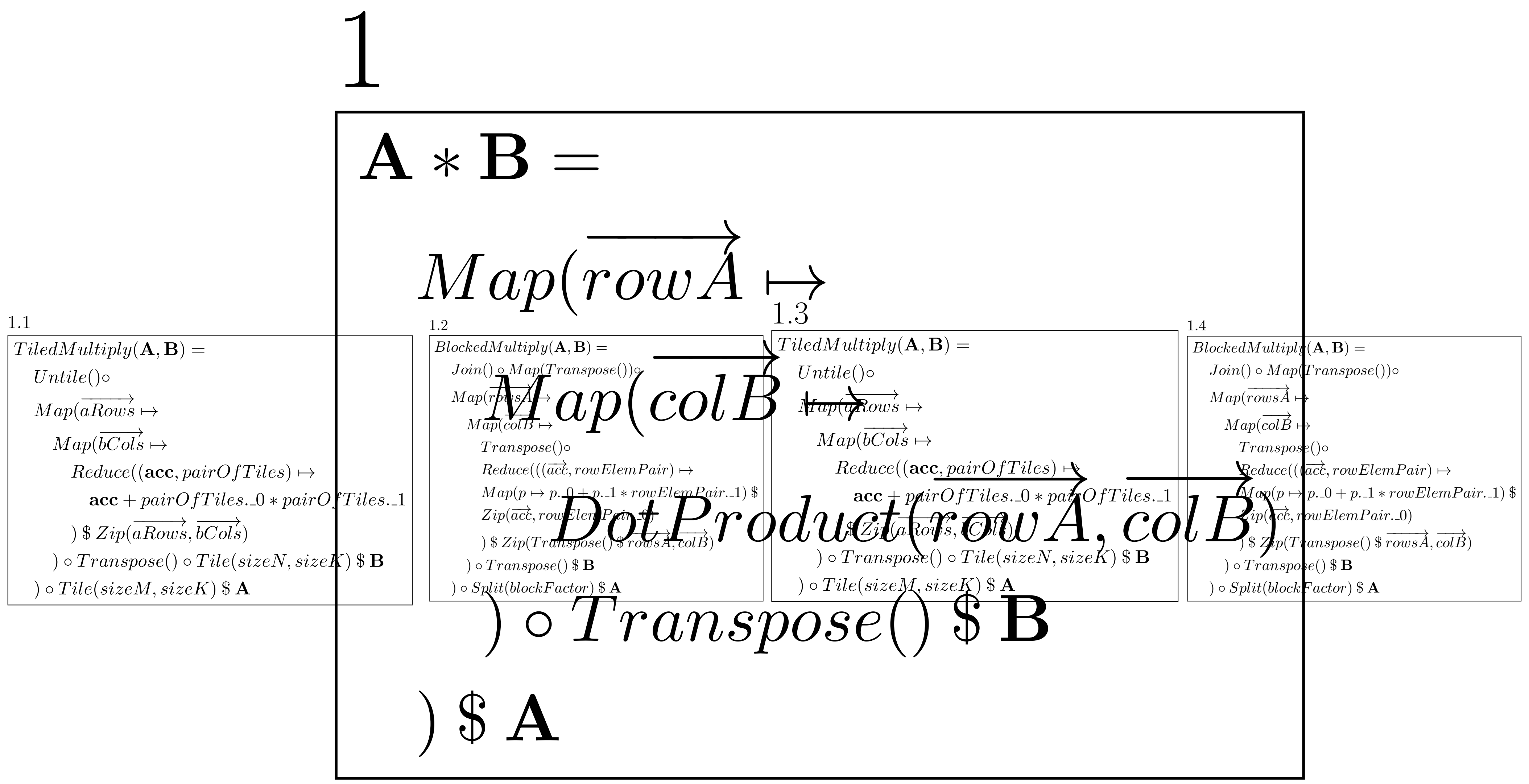
# Register Blocking as a Rewrite Rules

$$
\begin{aligned}
&registerBlocking = \\
&\quad Map(f) \Rightarrow Join() \circ Map(Map(f)) \circ Split(k) \\
&\quad Map(a \mapsto Map(b \mapsto f(a,b))) \Rightarrow Transpose() \circ Map(b \mapsto Map(a \mapsto f(a,b))) \\
&\quad Map(f \circ g) \Rightarrow Map(f) \circ Map(g) \\
&\quad Map(Reduce(f)) \Rightarrow Transpose() \circ Reduce((acc,x) \mapsto Map(f) \circ Zip(acc,x)) \\
&\quad Map(Map(f)) \Rightarrow Transpose() \circ Map(Map(f)) \circ Transpose() \\
&\quad Transpose() \circ Transpose() \Rightarrow id \\
&\quad Reduce(f) \circ Map(g) \Rightarrow Reduce((acc,x) \mapsto f(acc, g(x))) \\
&\quad Map(f) \circ Map(g) \Rightarrow Map(f \circ g)
\end{aligned}
$$

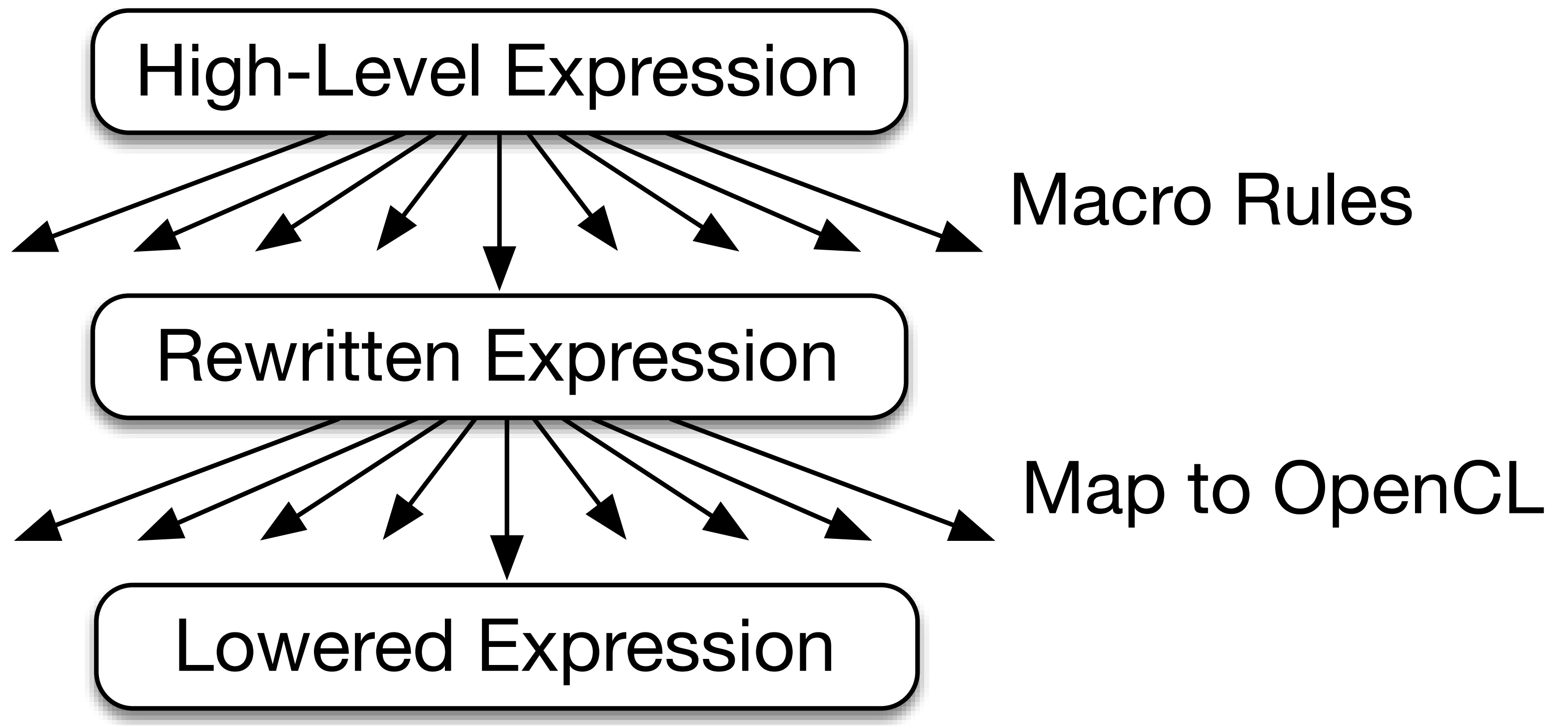


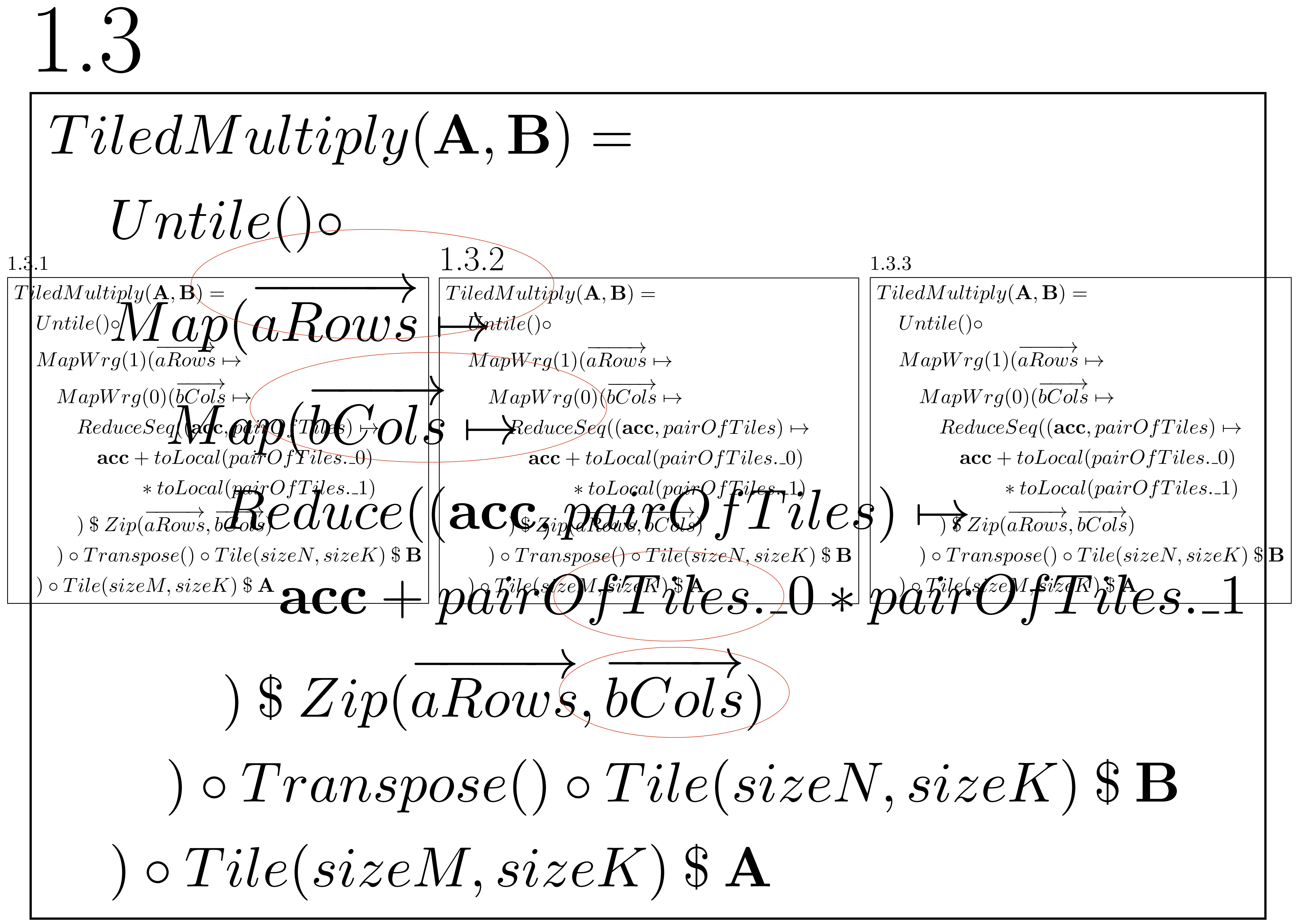
# Exploration Strategy



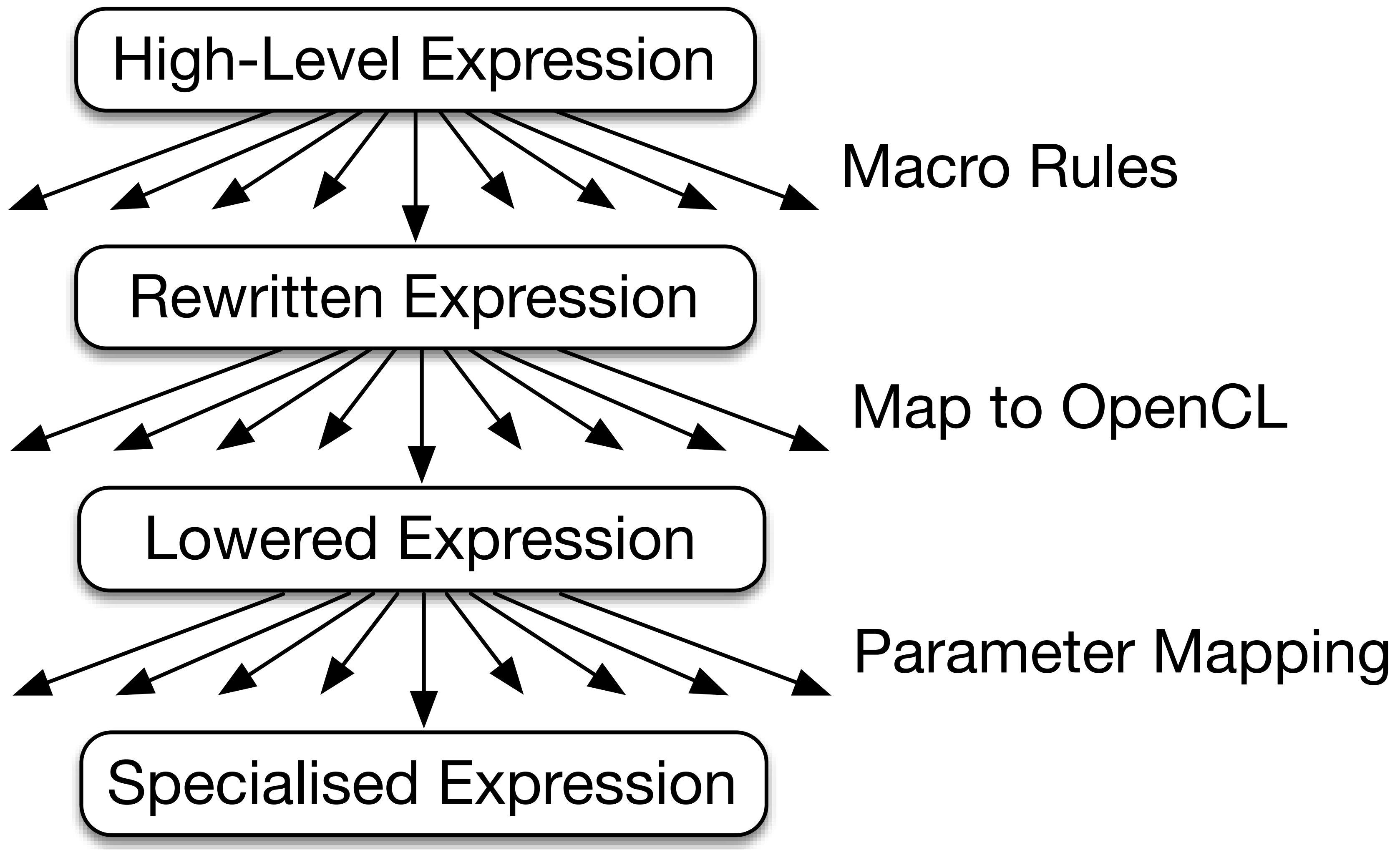



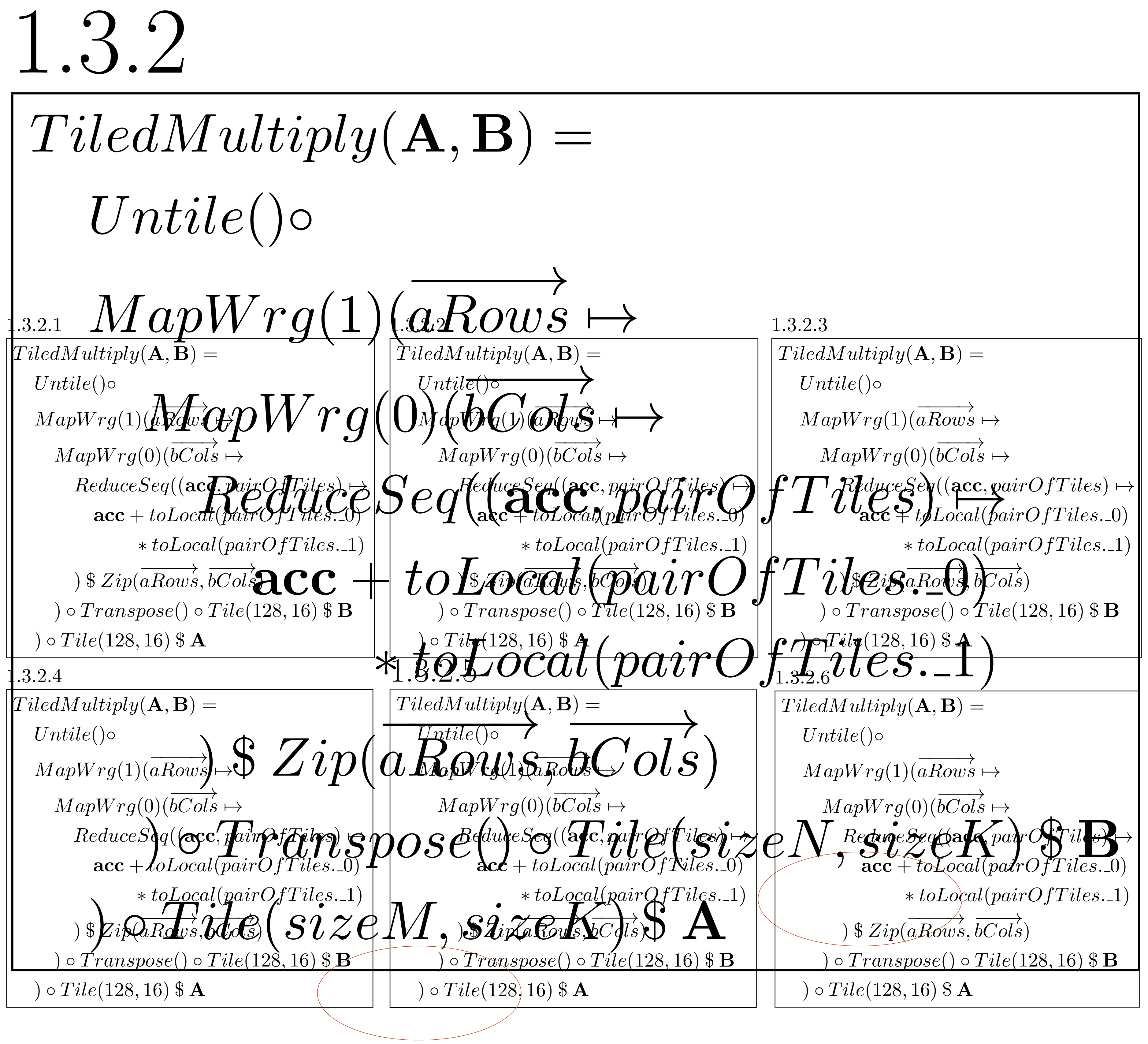
# Exploration Strategy



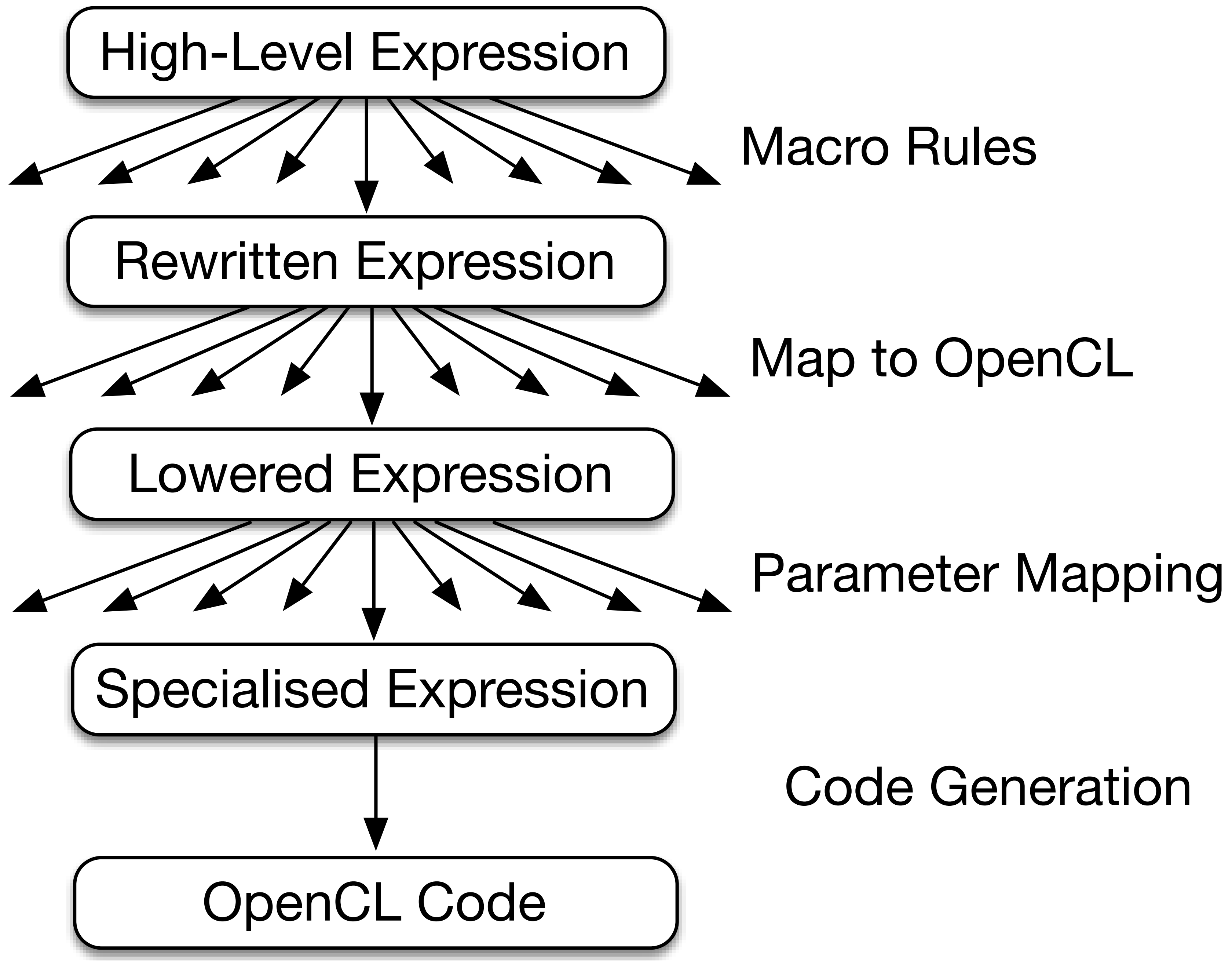



# Exploration Strategy

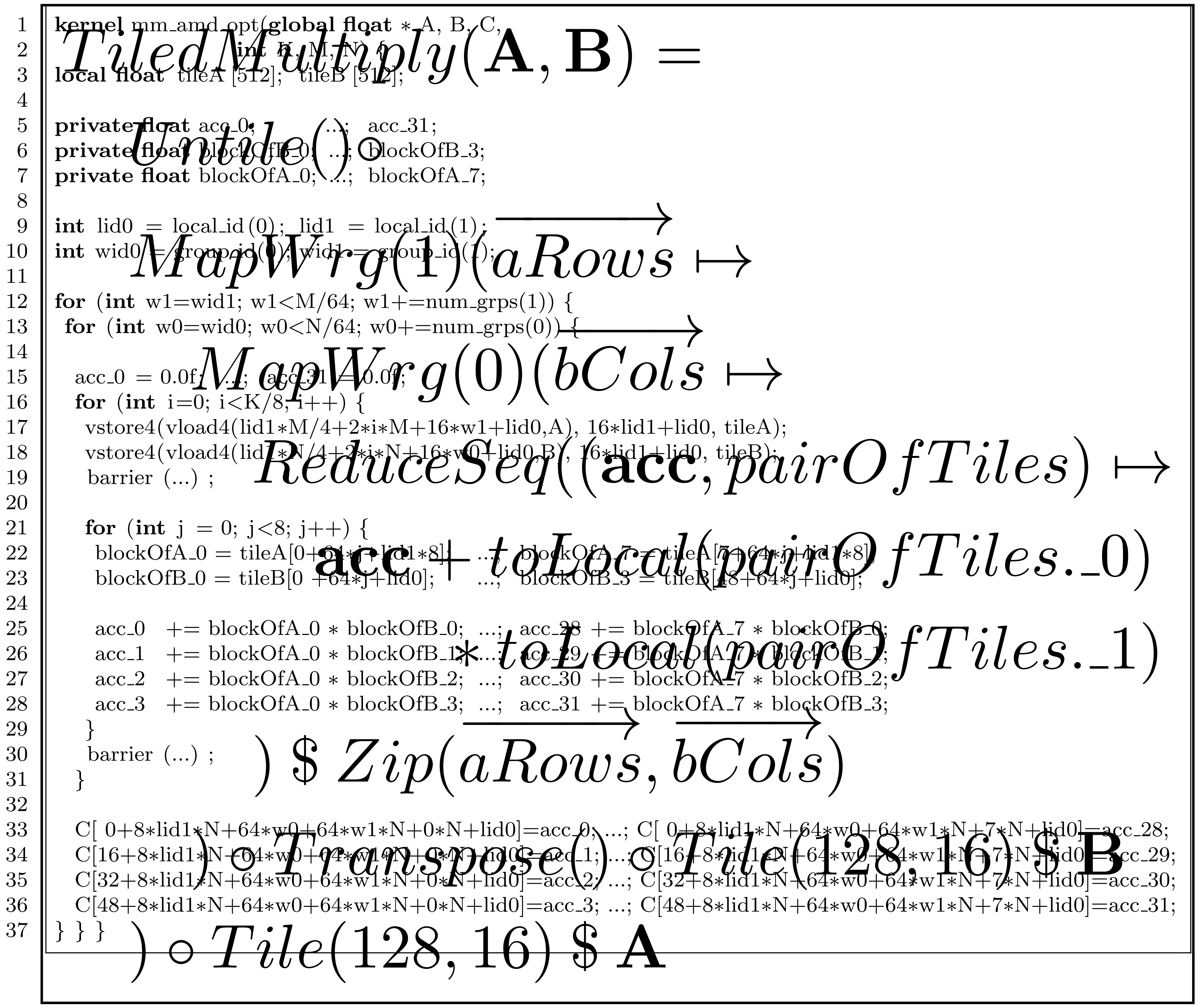


# Exploration Strategy




1.3.2.5



# Heuristics for Matrix Multiplication

**For Macro Rules:**

- Nesting depth
- Distance of addition and multiplication
- Number of times rules are applied

**For Map to OpenCL:**

- Fixed parallelism mapping
- Limited choices for mapping to local and global memory
- Follows best practice

**For Parameter Mapping:**

- Amount of memory used
  - Global
  - Local
  - Registers
- Amount of parallelism
  - Work-items
  - Workgroup

# Exploration in Numbers for Matrix Multiplication

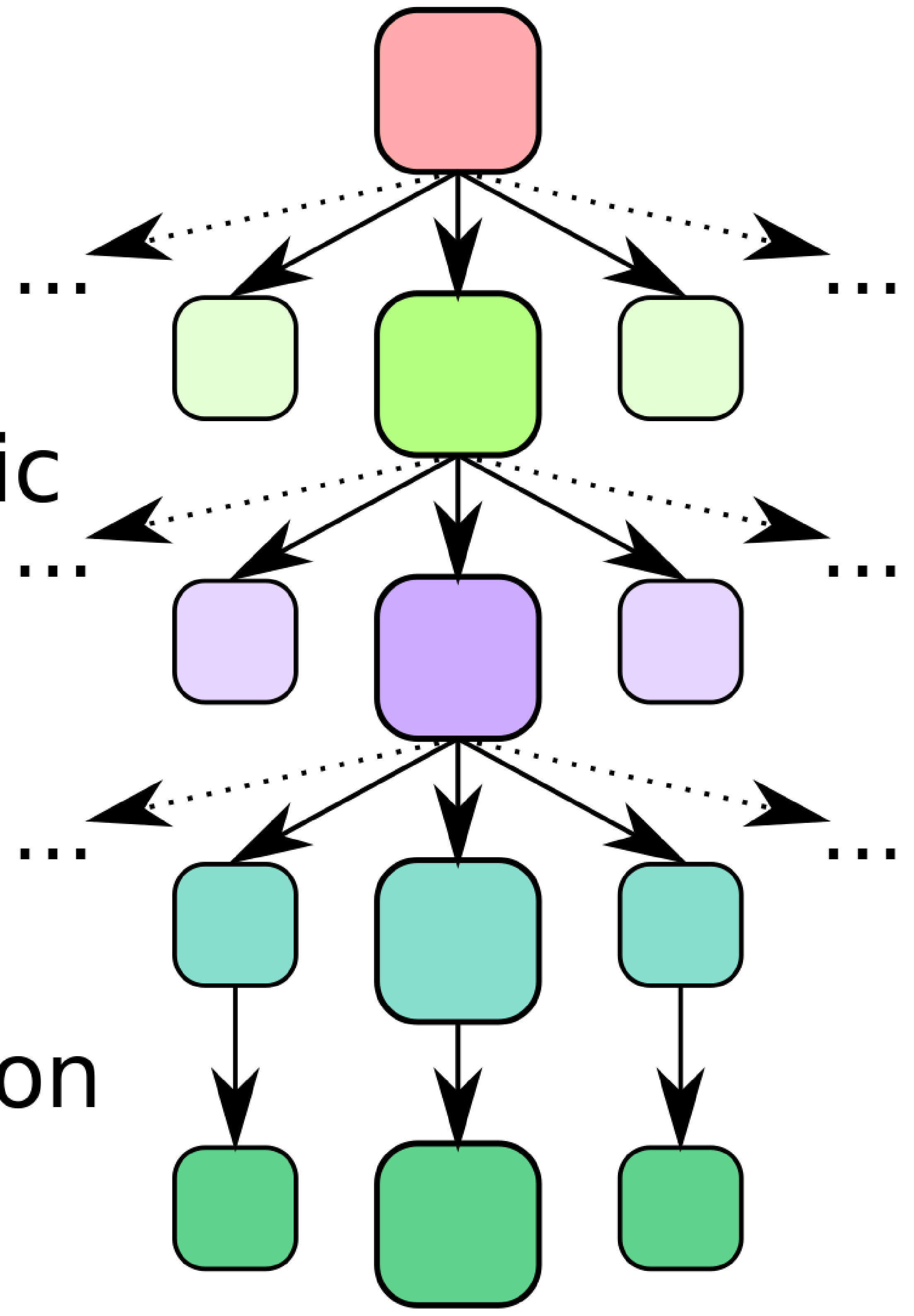
Phases:

Algorithmic Exploration

OpenCL specific Exploration

Parameter Exploration

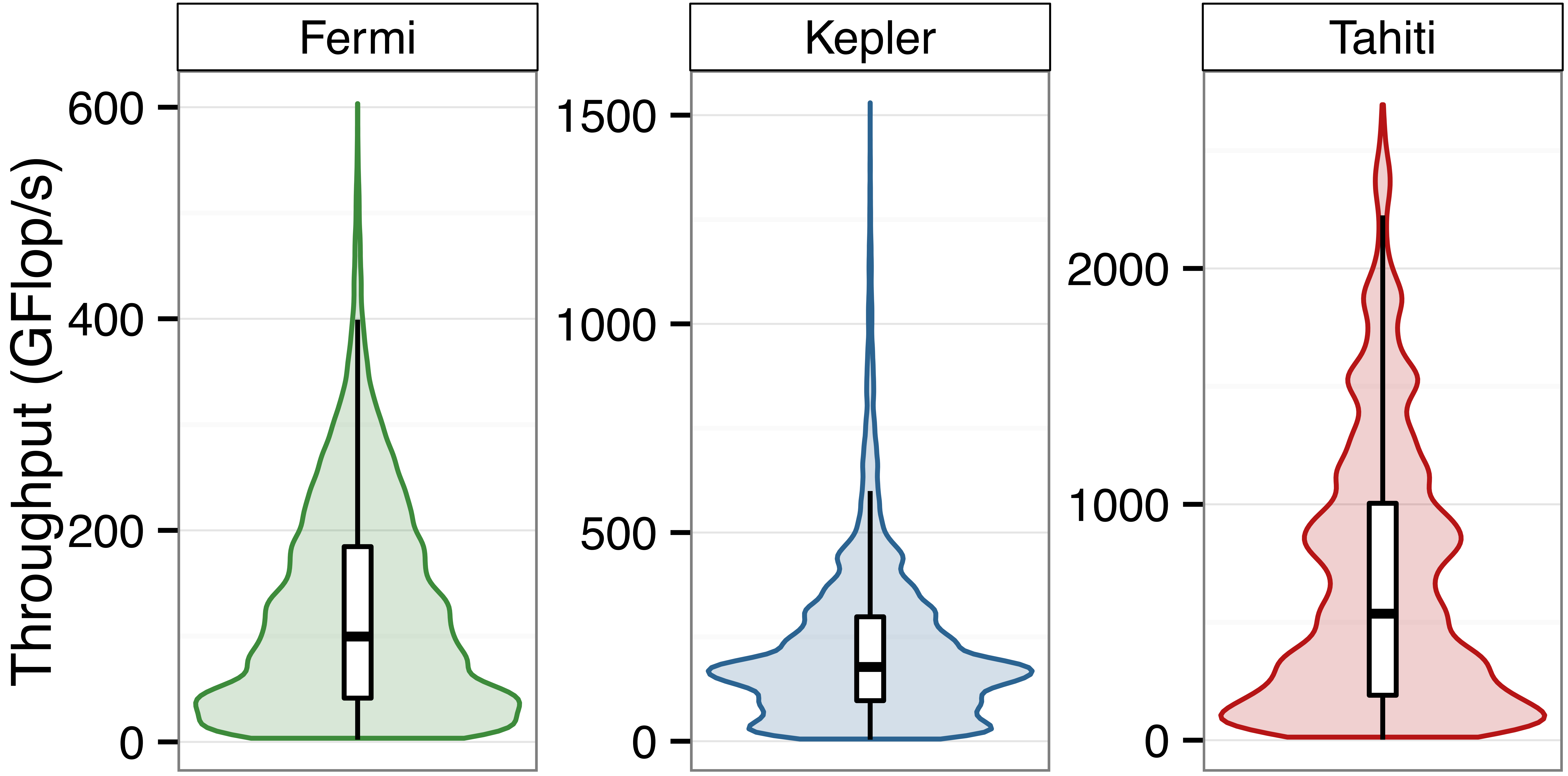
Code Generation




Program Variants:

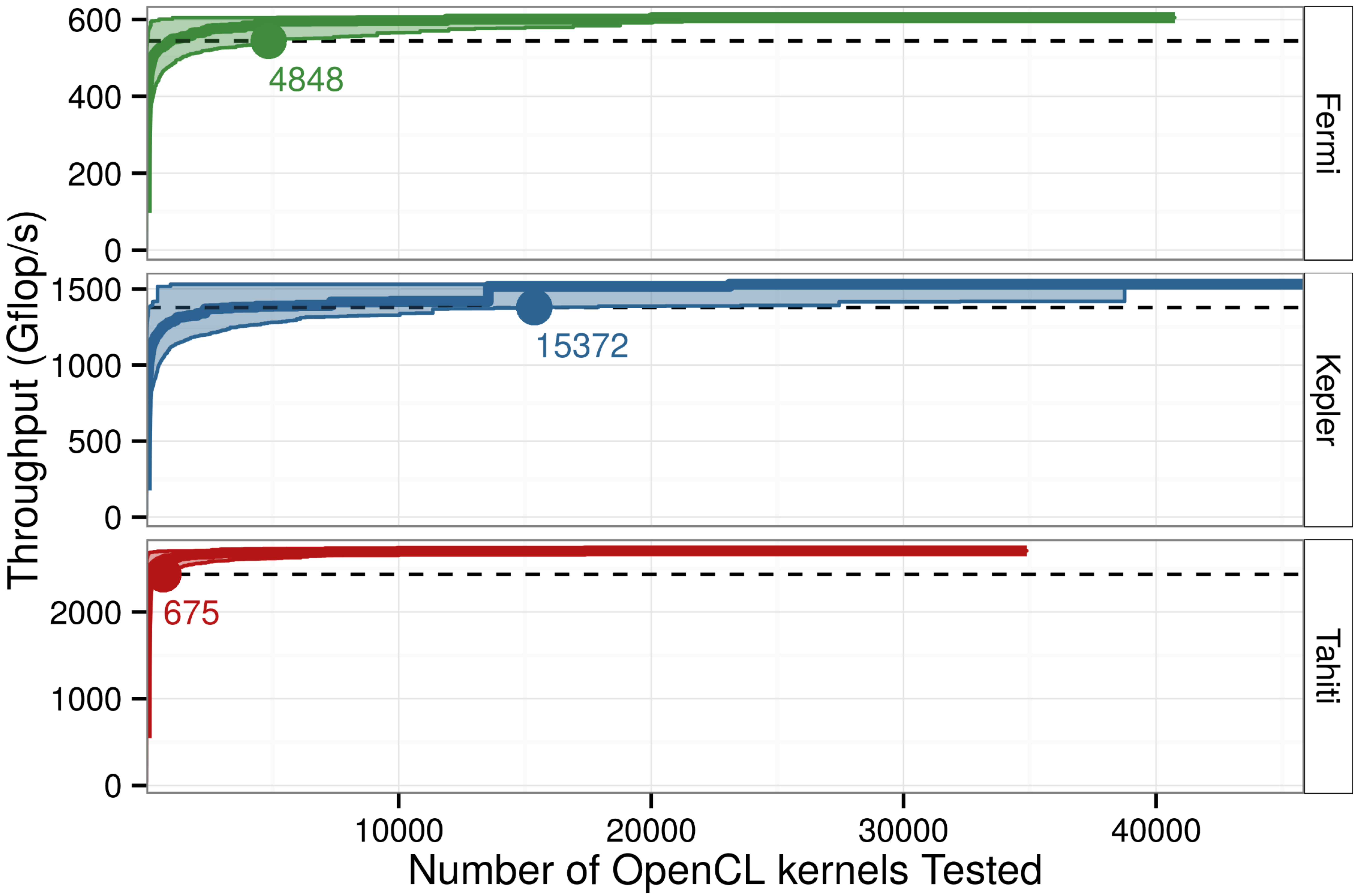
| | |
|---|---|
| High-Level Program | 1 |
| Algorithmic Rewritten Program | 8 |
| OpenCL Specific Program | 760 |
| Fully Specialized Program | 46,000 |
| OpenCL Code | 46,000 |

# Exploration Space for Matrix Multiplication




Only few OpenCL kernel with very good performance

# Performance Evolution for Randomised Search

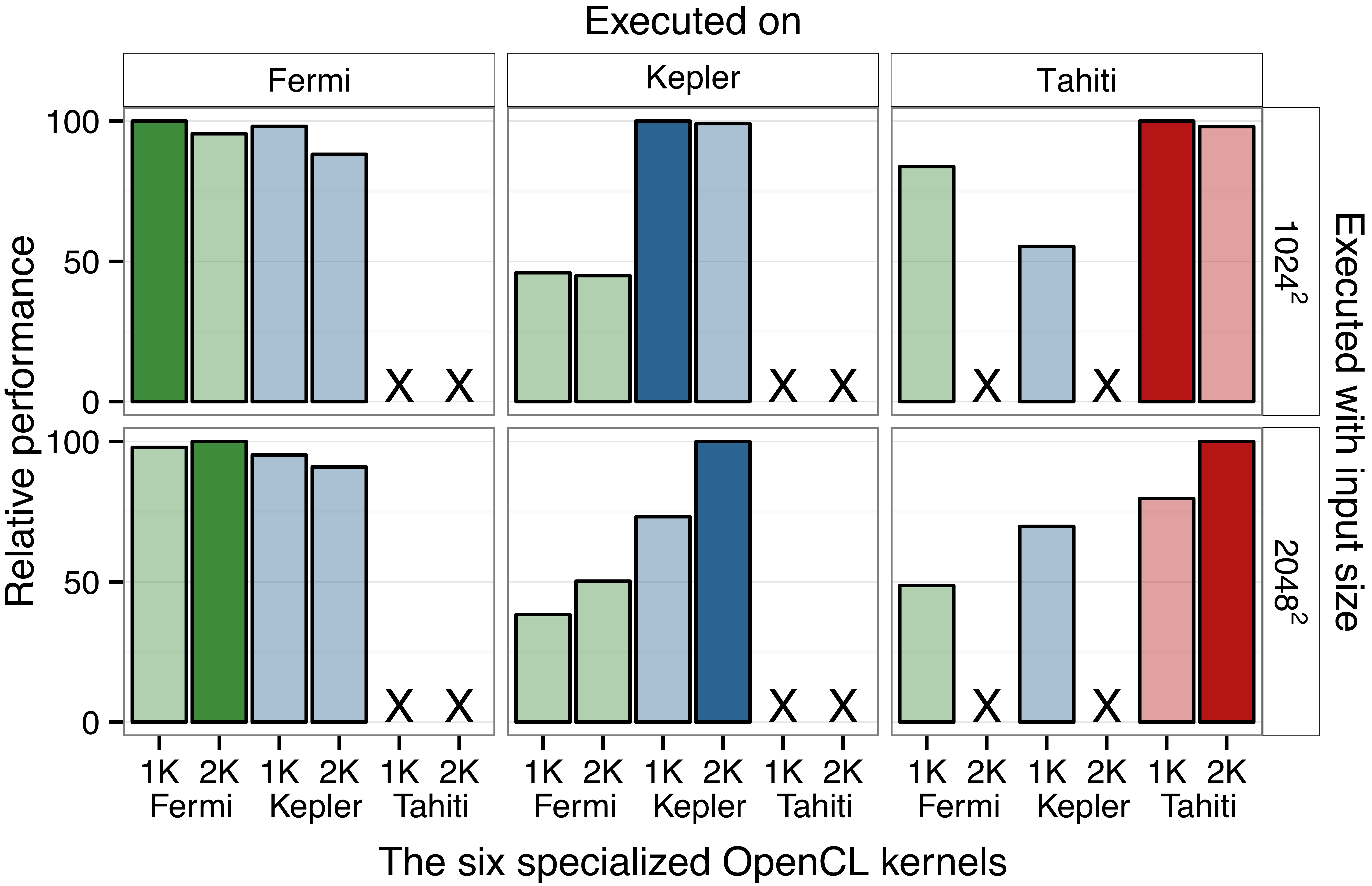



Even with a simple random search strategy one can expect to find a good performing kernel quickly

# Performance Results Matrix Multiplication

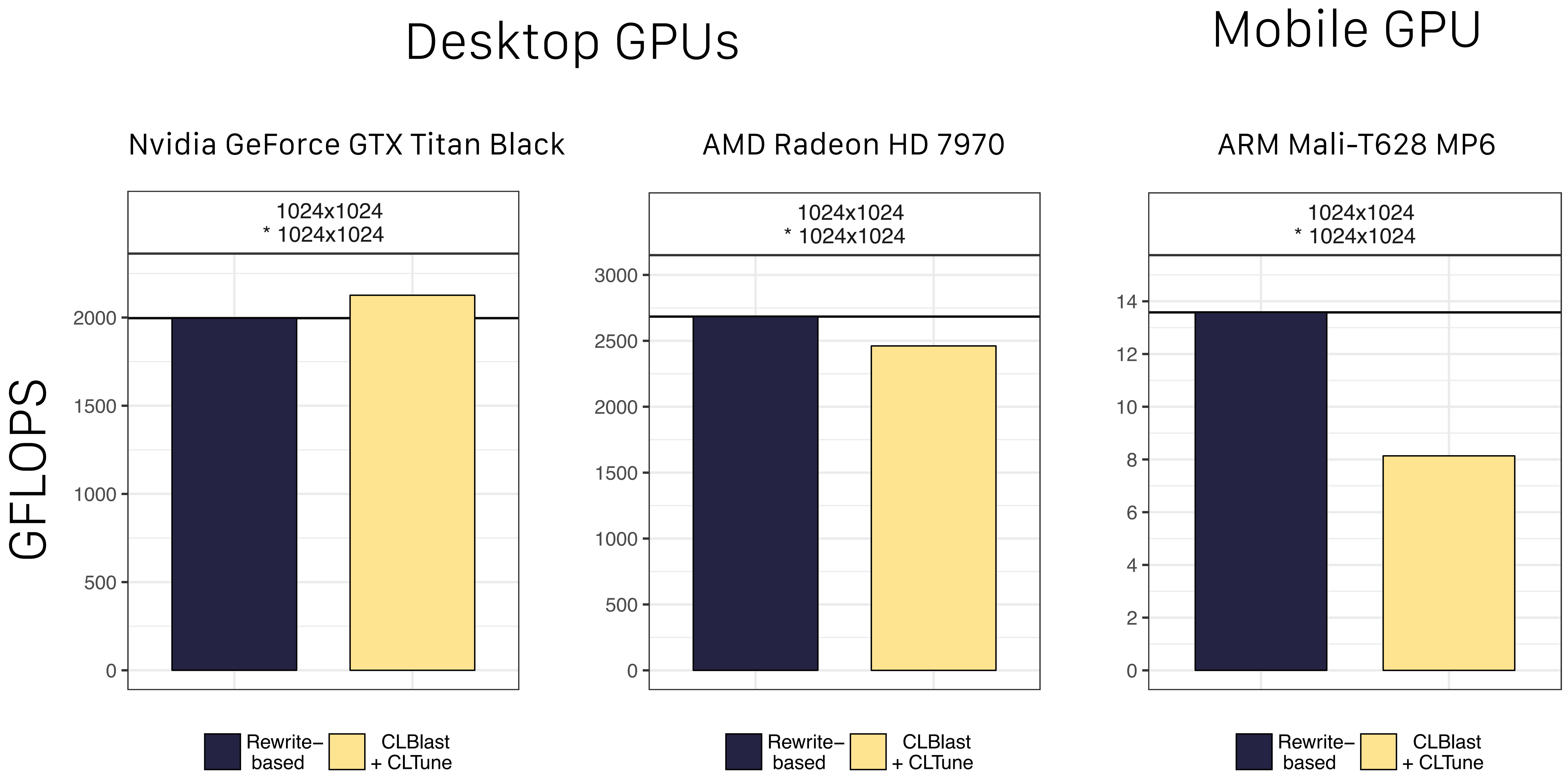

Performance close or better than hand-tuned MAGMA library

# Performance Portability Matrix Multiplication




Generated kernels are specialised for device and input size

# Desktop GPUs vs. Mobile GPU




Performance portable even for mobile GPU device!




# The LIFT Team




University of Glasgow

THE UNIVERSITY of EDINBURGH

WWU MÜNSTER

Federico Pizzuti
PhD Student
University of Edinburgh

# Identify *hidden parallelism* in LIFT programs

## Parallelising non-associative reductions

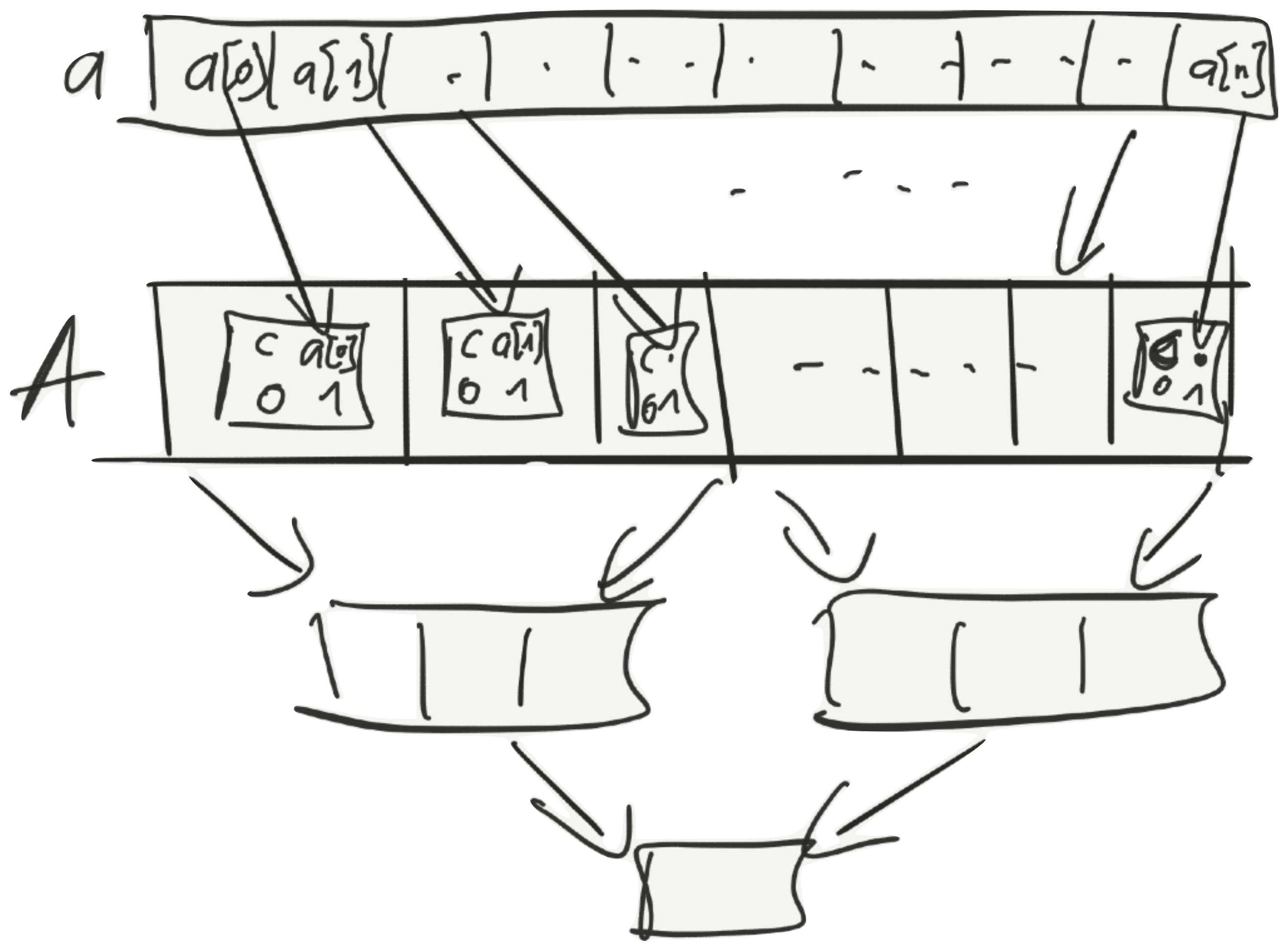
$$x \leftarrow 0;\ \textbf{for } i = 0 \textbf{ to } n \textbf{ do } x \leftarrow c \cdot x + a[i] \textbf{ done}.$$

↓

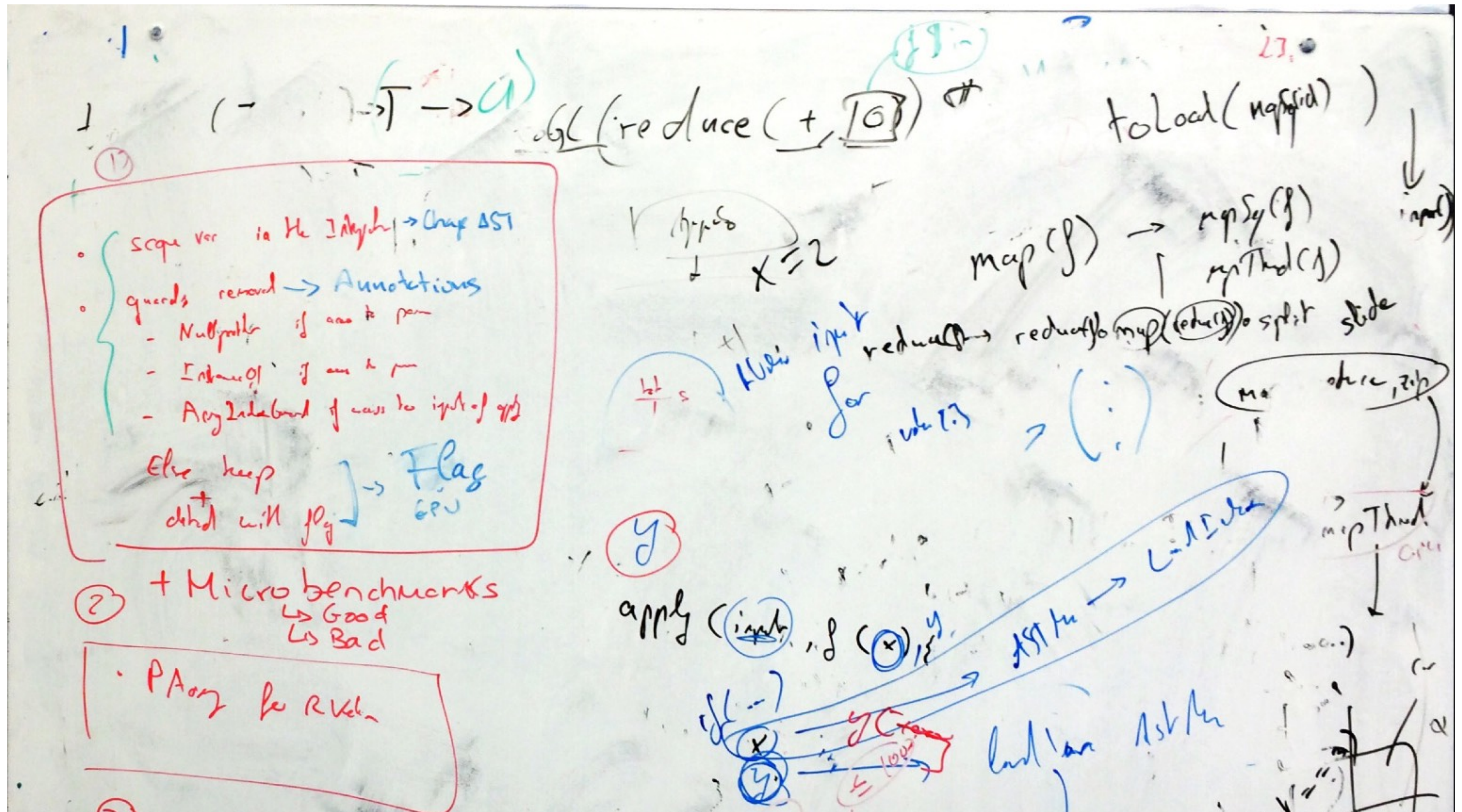
$$\boldsymbol{x} \leftarrow \boldsymbol{x}_0;\ \textbf{for } i = 0 \textbf{ to } n \textbf{ do } \boldsymbol{x} \leftarrow A_i \times \boldsymbol{x} \textbf{ done},$$

$$\textbf{where } \boldsymbol{x} = \begin{pmatrix} x \\ 1 \end{pmatrix},\ \boldsymbol{x}_0 = \begin{pmatrix} 0 \\ 1 \end{pmatrix},\ A_i = \begin{pmatrix} c & a[i] \\ 0 & 1 \end{pmatrix}.$$

↓

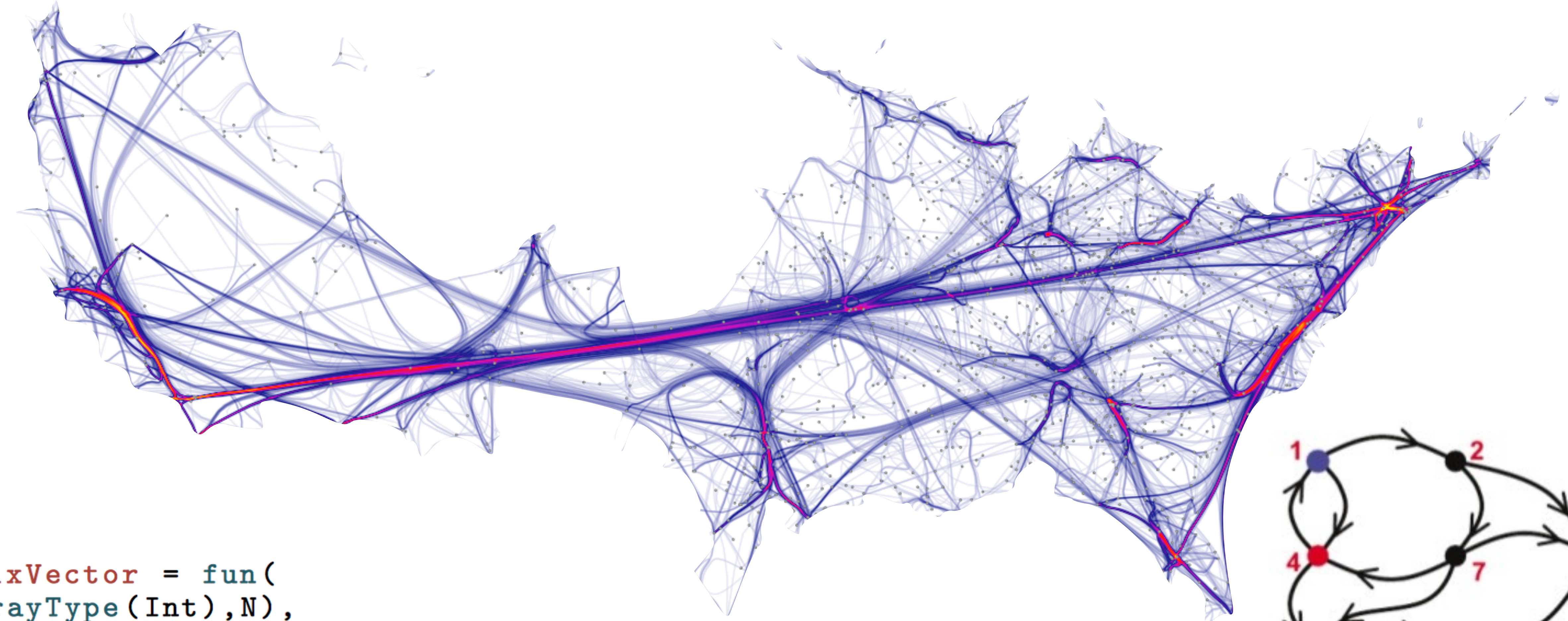


Key idea: Rearrange data as matrices to exploit associative matrix multiplication



# Graph Algorithms via Sparse Linear Algebra in LIFT

Adam Harries
PhD Student
University of Edinburgh



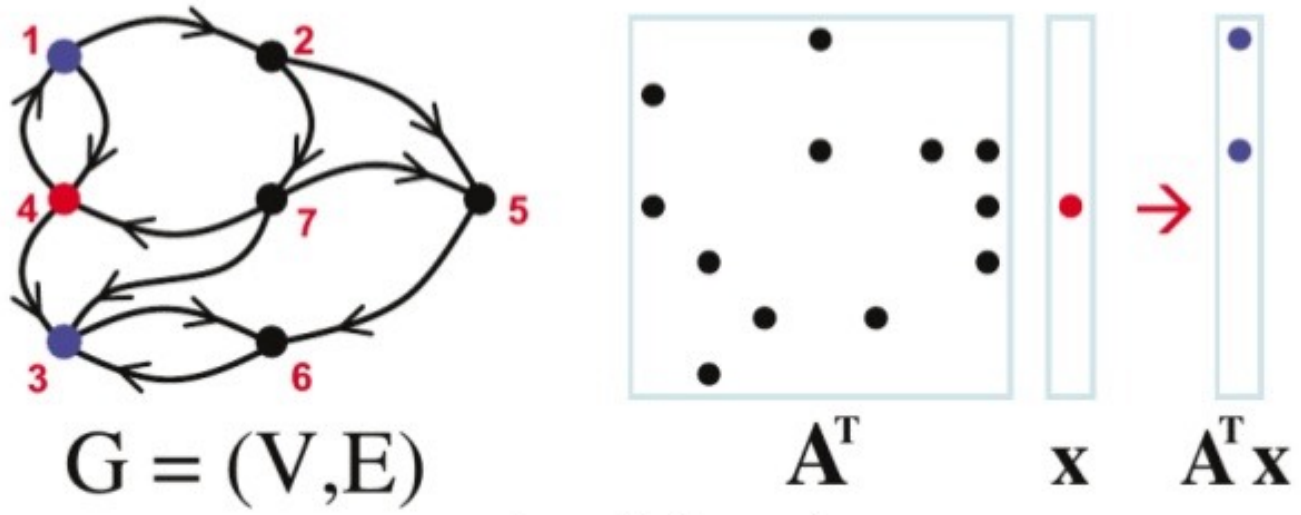
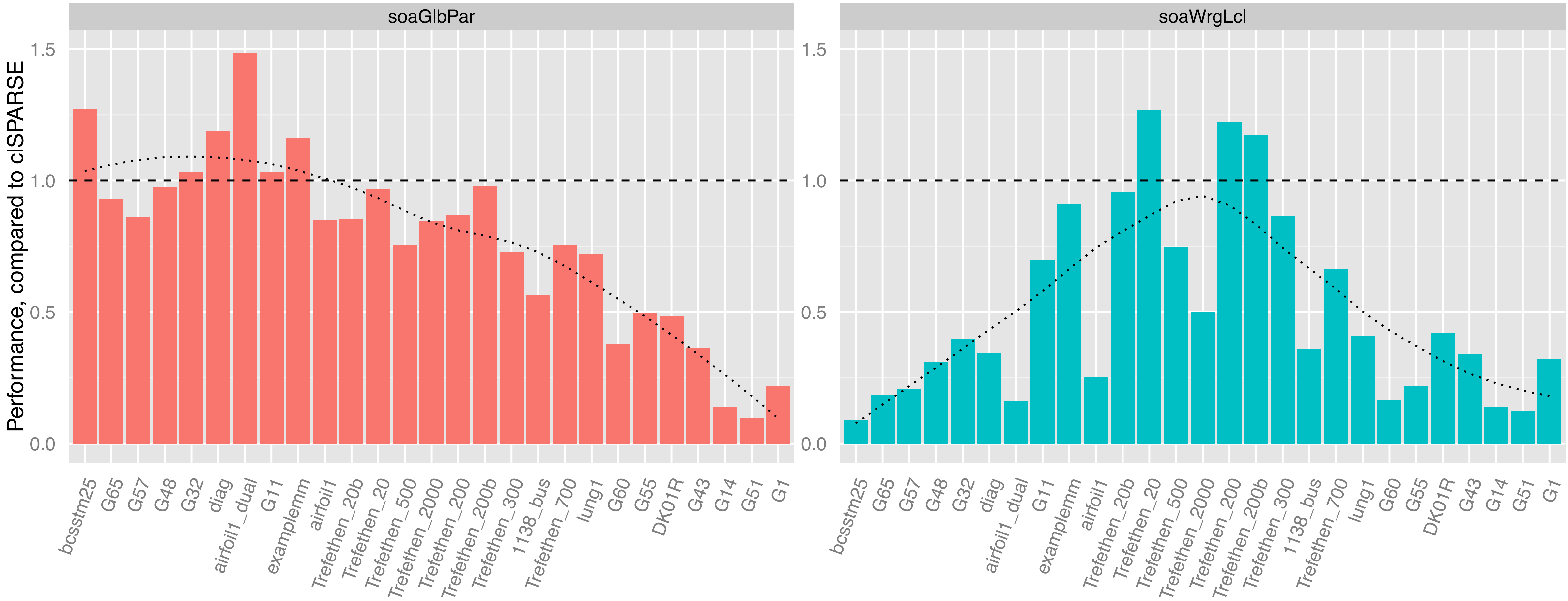



Image credit: [Kepner2011]

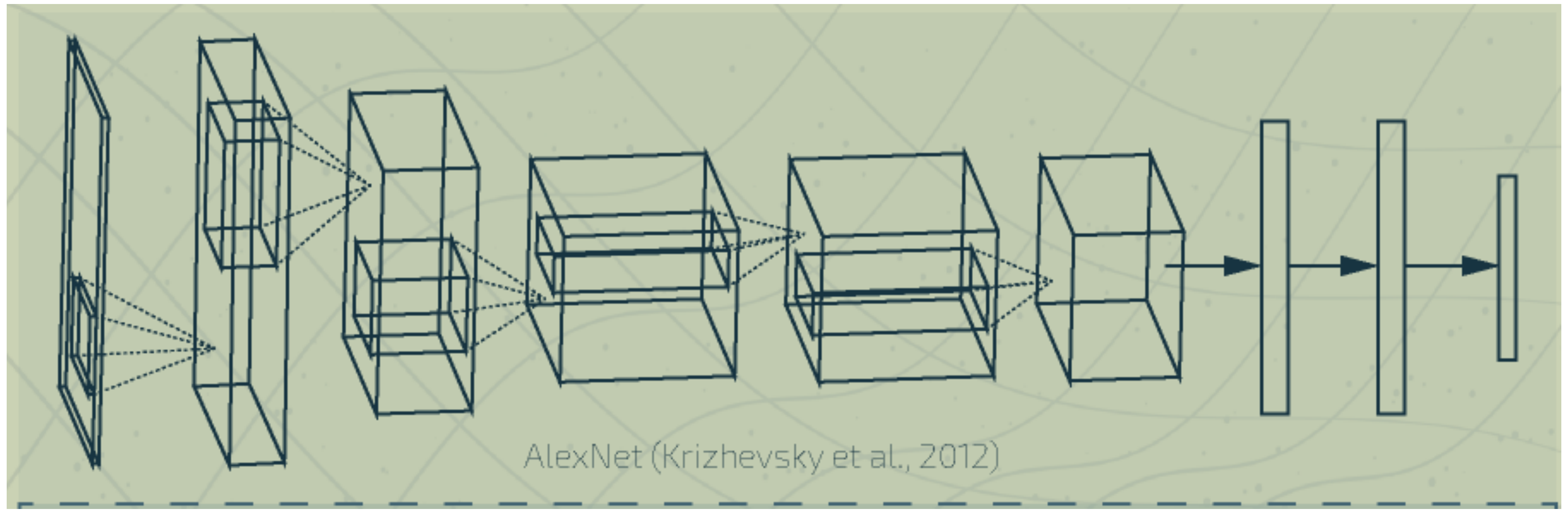
```
val sparseMatrixVector = fun(
  ArrayType(ArrayType(Int),N),
  ArrayType(ArrayType(ElemT),N),
  ArrayType(ElemT, M),
  (indices, values, vector) =>
    Map(fun(row =>
          sparseDotProduct(row, vector)),
      Map(Zip,Zip(indices, values))) )
```




Differently optimised kernels for different inputs

Naums Mogers
PhD Student
University of Edinburgh

# Optimising Deep Learning with LIFT



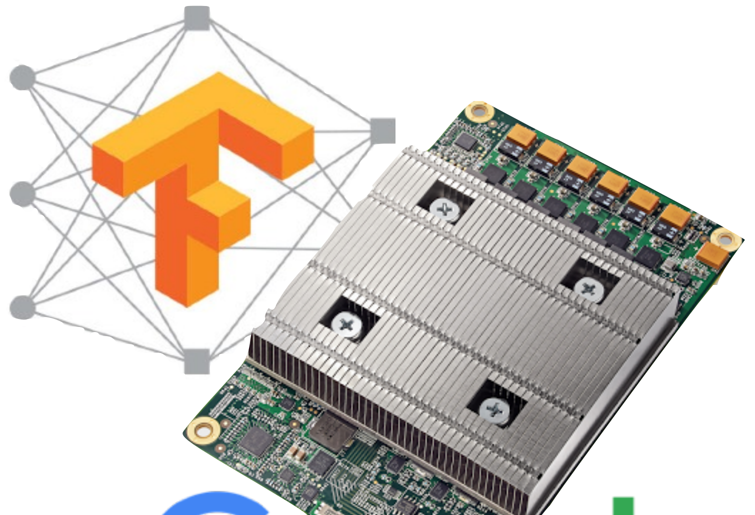

Express layers with LIFT primitives

```
fully_connected(f, weights, bias, inputs) :=
  Map((neuron_weights, neuron_bias) → f() o Reduce(add, neuron_bias) o
  Map(mult) $ Zip(inputs, neuron_weights)) $ Zip(weights, bias)
```

Optimise individual layers and across layers via rewrites

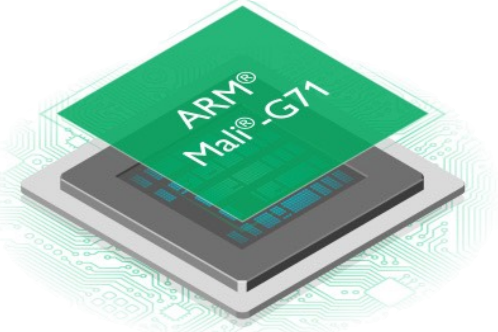
FPGAs

Microsoft Research



Google TPU

Low Power Devices

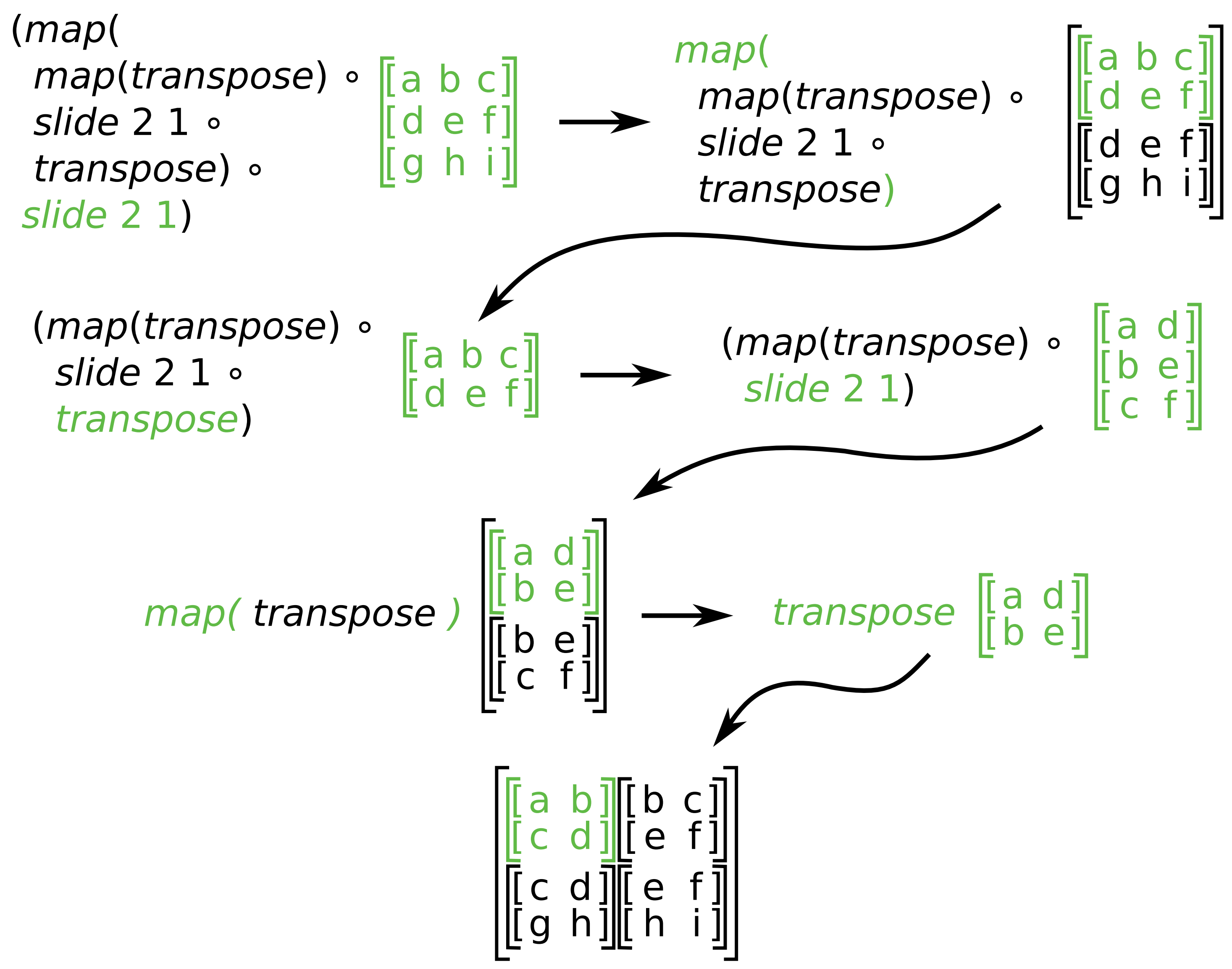


# Stencil Computations in LIFT

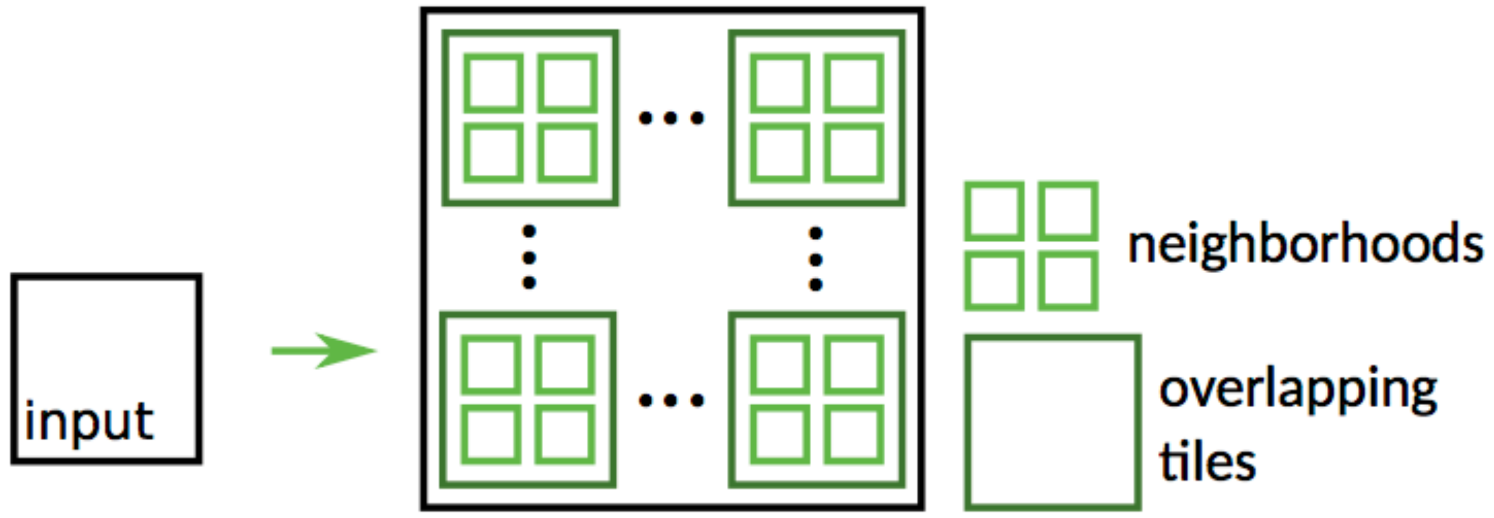
Bastian Hagedorn
PhD Student
University of Münster

Larisa Stoltzfus
PhD Student
University of Edinburgh

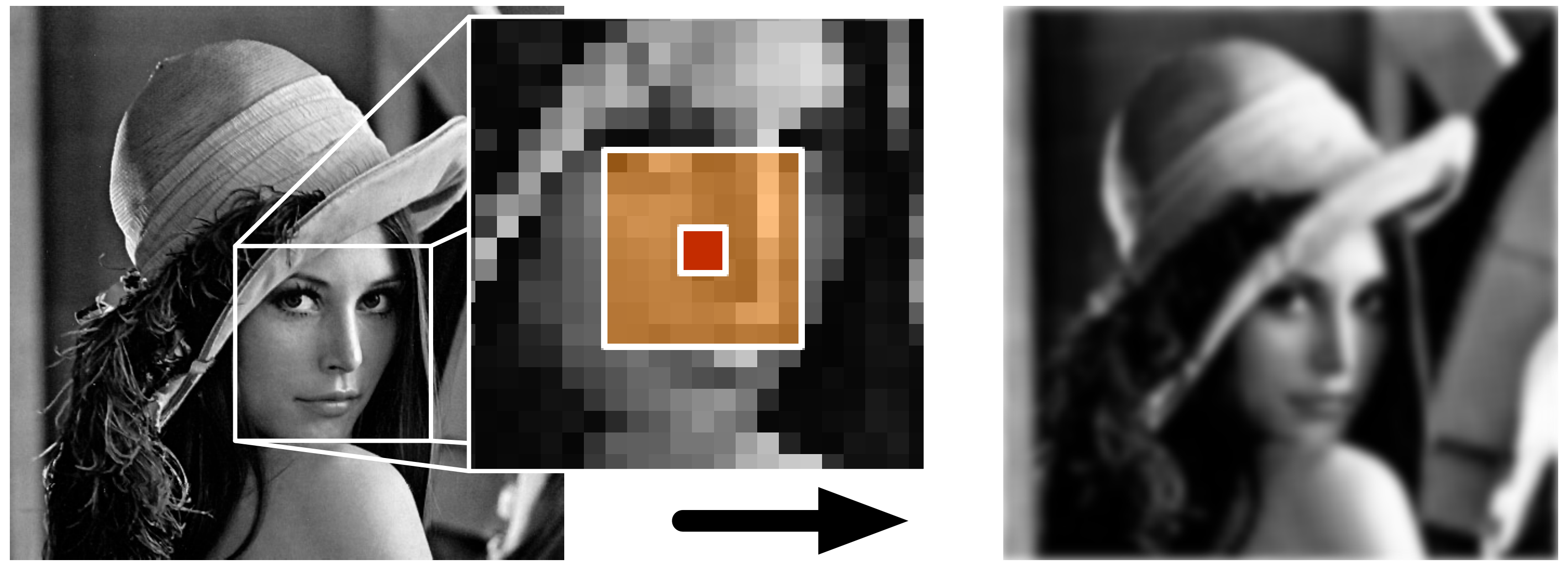
## Express Stencil with Skeletons



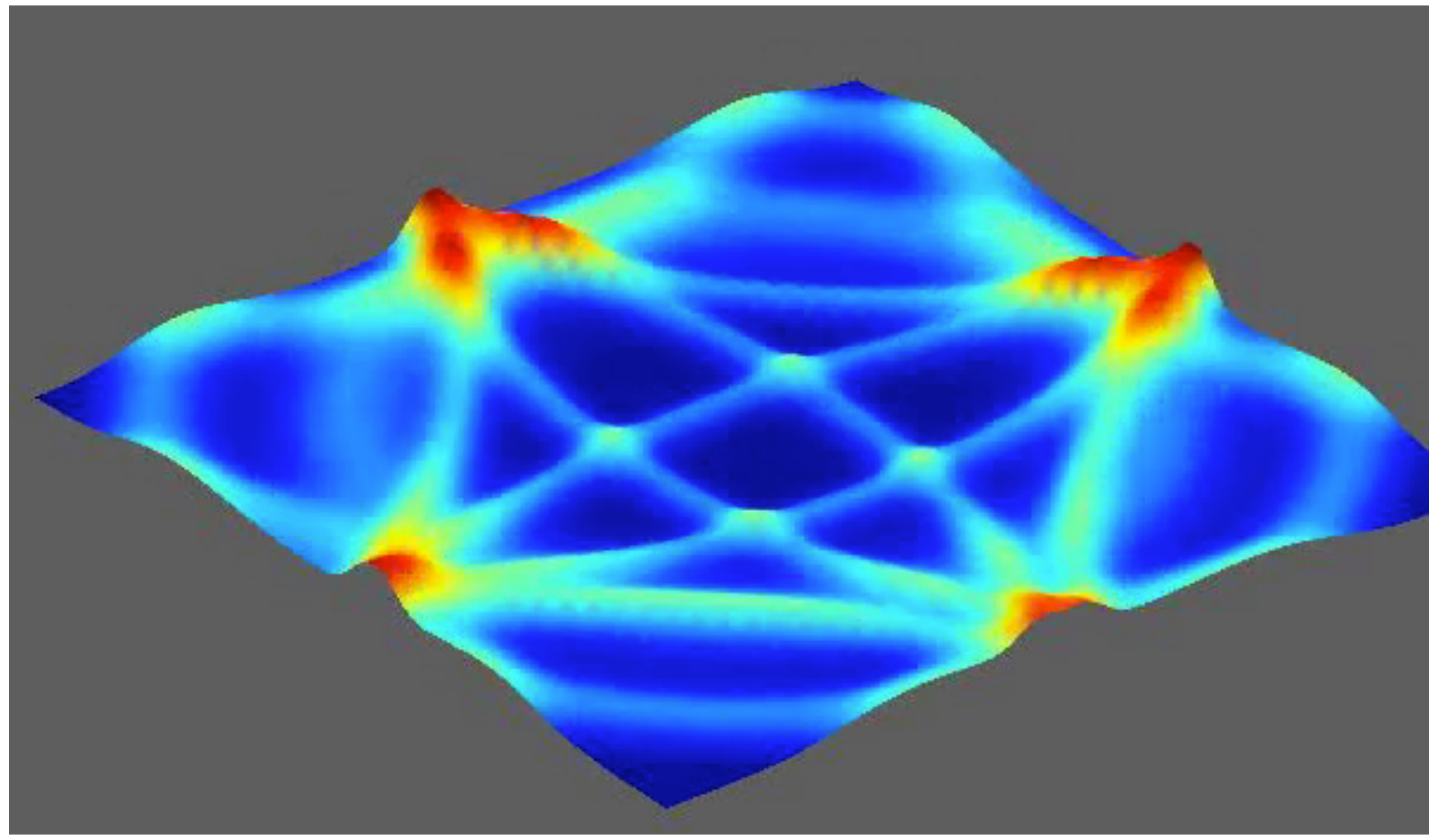

## Explore optimisations as rewrites




## Image Processing



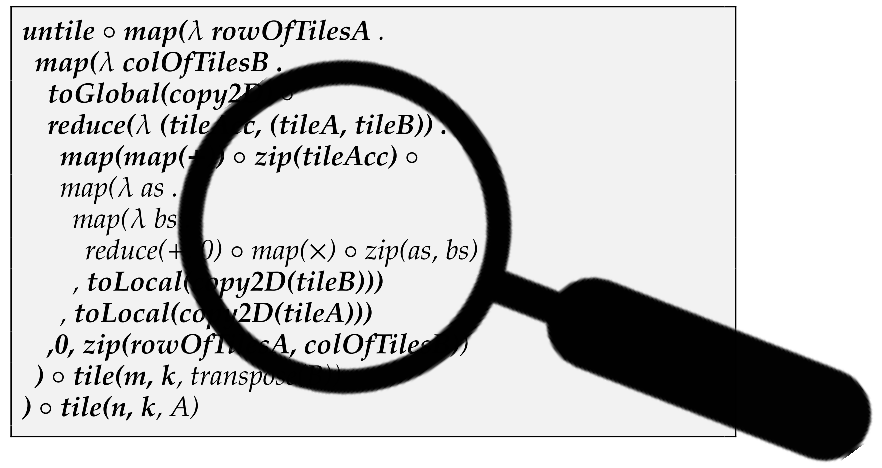
## Acoustics Simulation



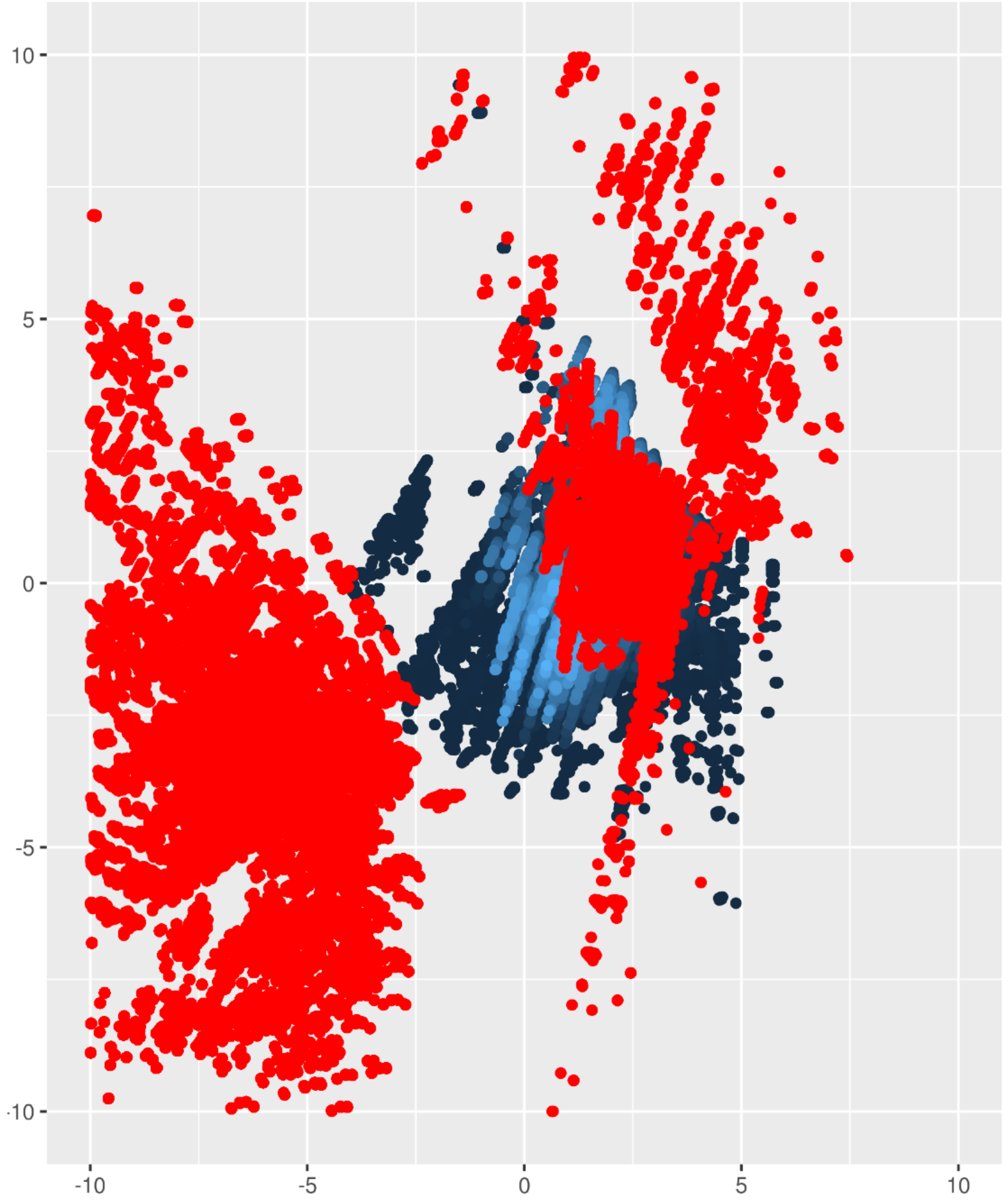
Video

# Performance Modeling of LIFT Programs

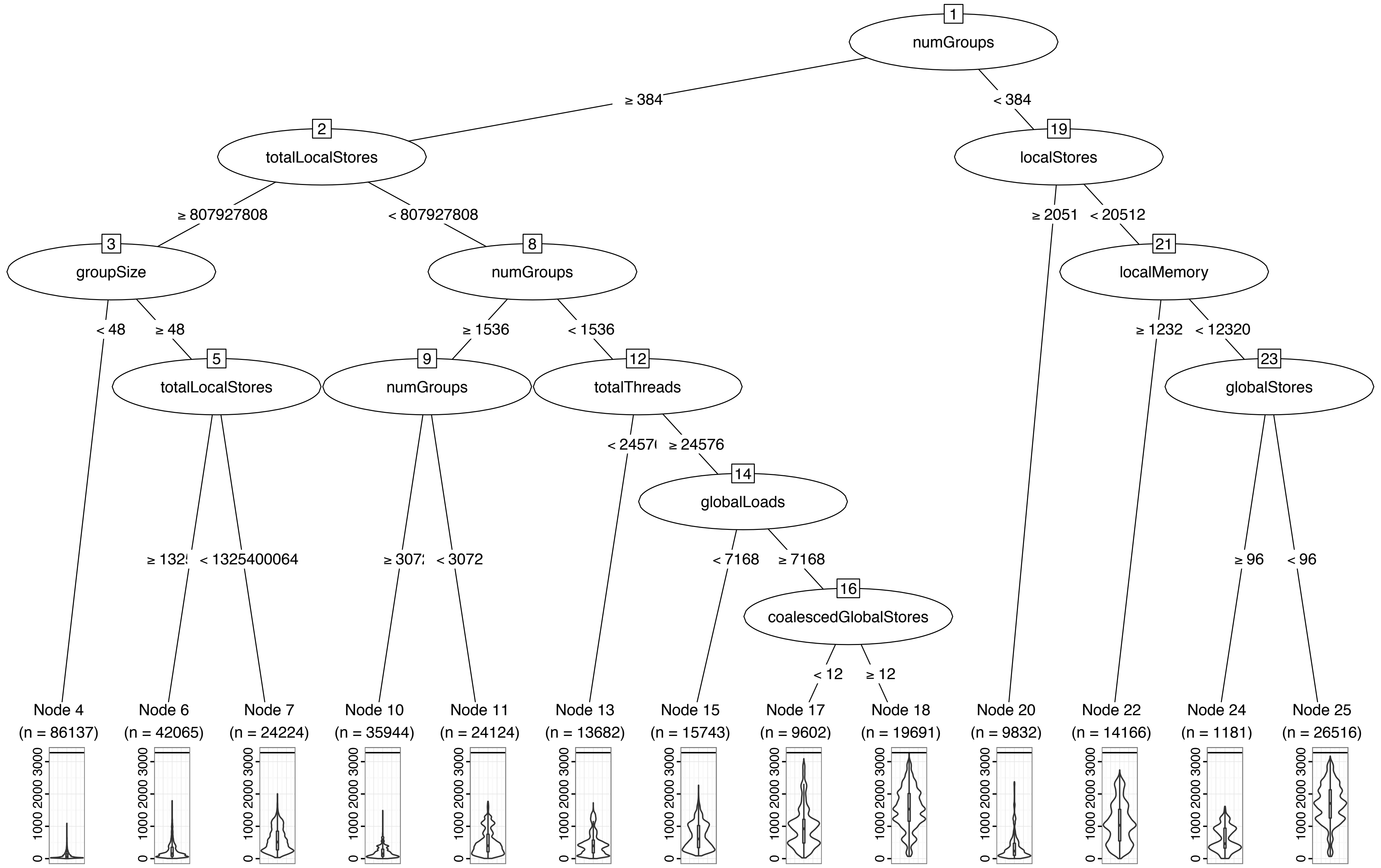
Toomas Remmelg
PhD Student
University of Edinburgh




Extract Features




Performance Model



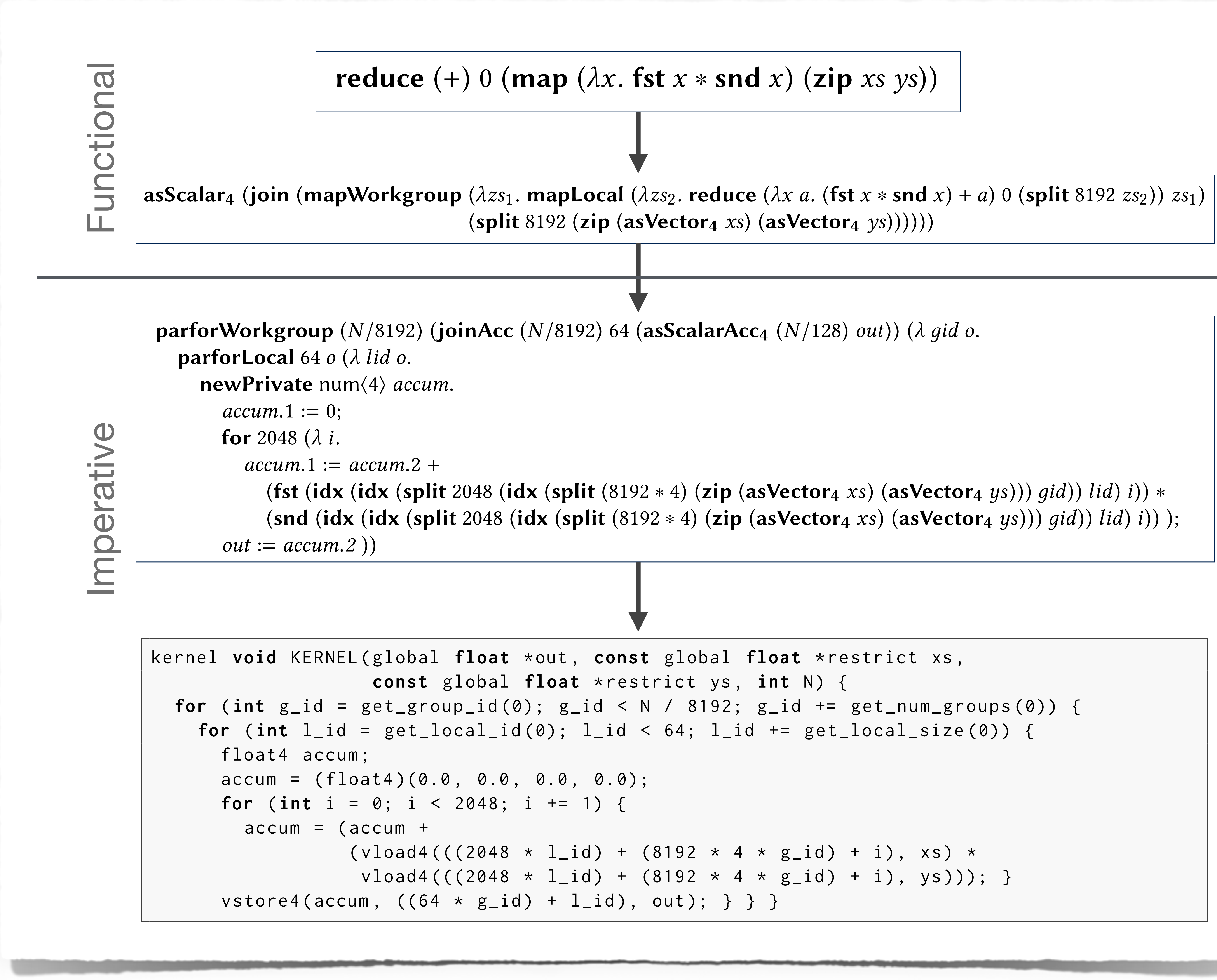

Predictions used to drive the rewrite process

# Data Parallel Idealised Algol as new foundation for LIFT

Collaboration with Bob Atkey (Strathclyde) and Sam Lindley (Edinburgh)

- So far LIFT rewrites only functional expressions
- Data Parallel Idealised Algol (DPIA) combines functional and imperative constructs
- Allows formal translation of functional programs into efficient parallel imperative programs
- Types separate *expressions, acceptors,* and *commands*:
  - *Expressions* are purely functional computations
  - *Acceptors* describe modifiable locations in memory (≅ l-values in C)
  - *Commands* are imperative actions modifying memory

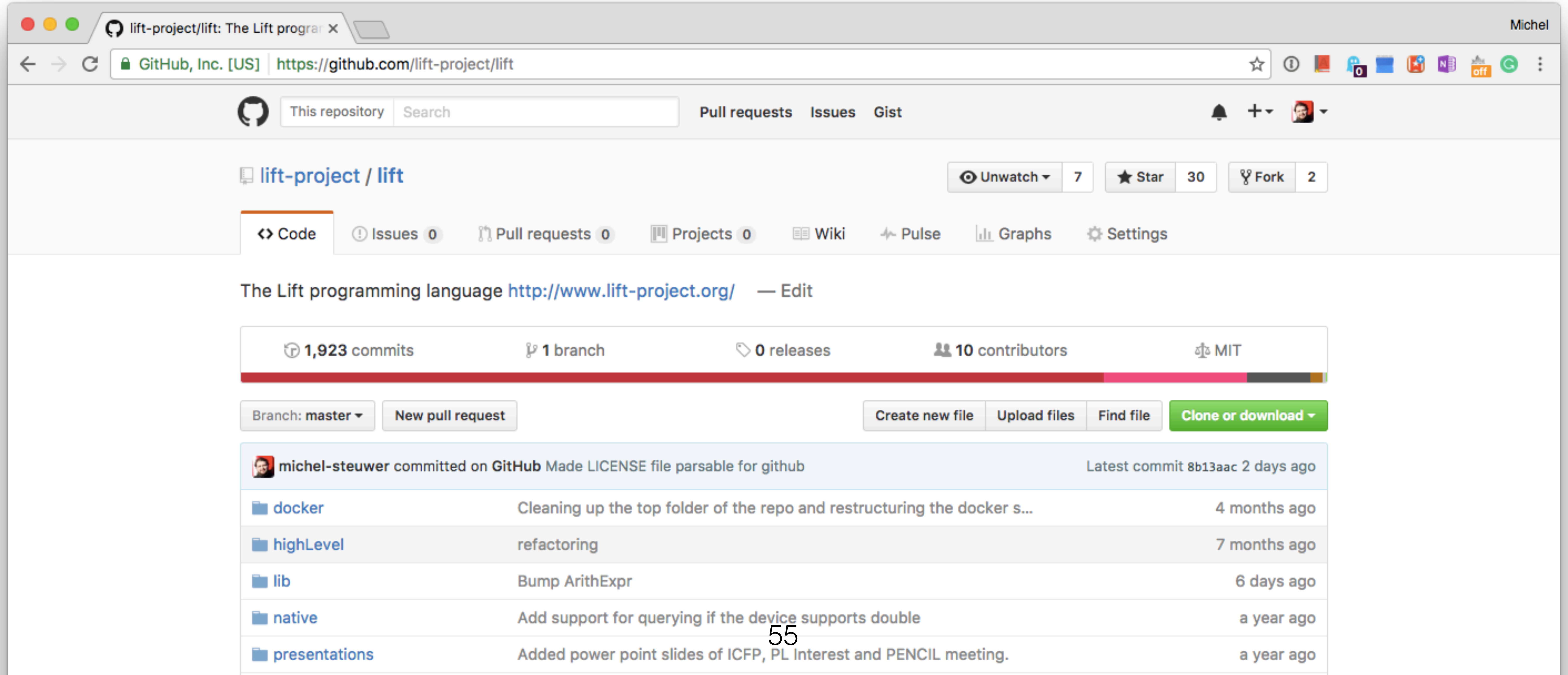
Draft Paper at: https://bentnib.org/dpia.html



# LIFT is Open-Source Software

http://www.lift-project.org/

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



University of Glasgow

# The LIFT Project

Performance Portable Parallel Code Generation via Rewrite Rules

Michel Steuwer — *michel.steuwer@glasgow.ac.uk*

*www.lift-project.org* @LIFTlang

INSPIRING PEOPLE

#UofGWorldChangers

@UofGlasgow