



University  
of Glasgow

# The LIFT Project

Performance Portable Parallel Code Generation via Rewrite Rules

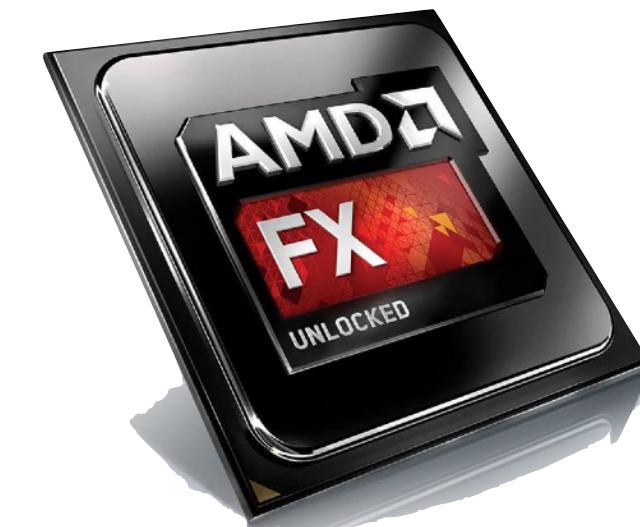
Michel Steuwer — [michel.steuwer@glasgow.ac.uk](mailto:michel.steuwer@glasgow.ac.uk)

**INSPIRING  
PEOPLE**

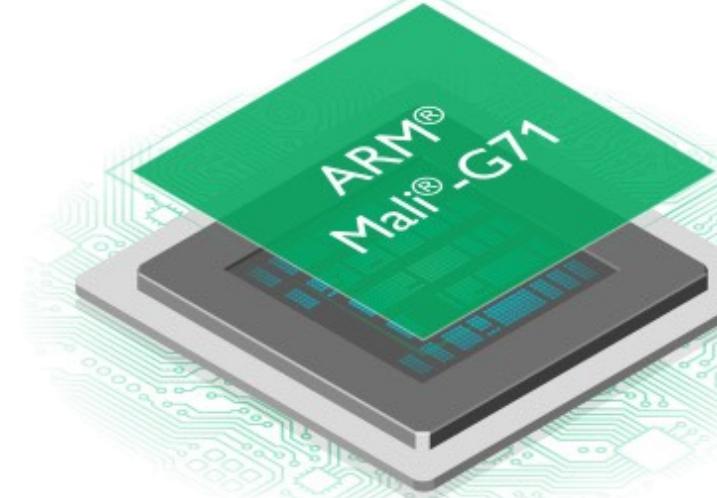


# What are the problems LIFT tries to tackle?

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



CPU



GPU



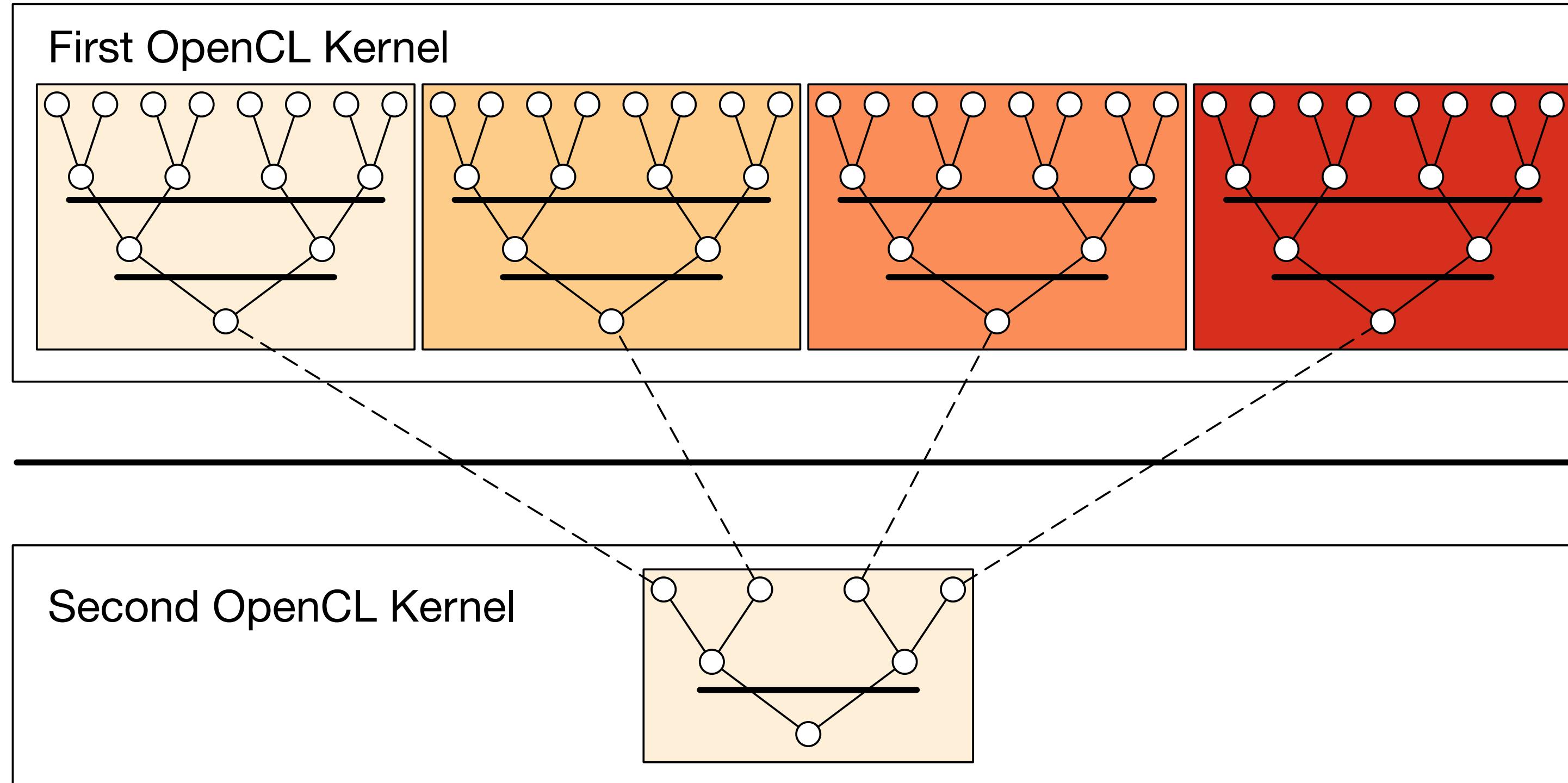
Accelerator



FPGA

# 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

Big, but slow **global** memory

Small, but fast **local** 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

Potential Deadlock!

```
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];
}
```

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];
}
```

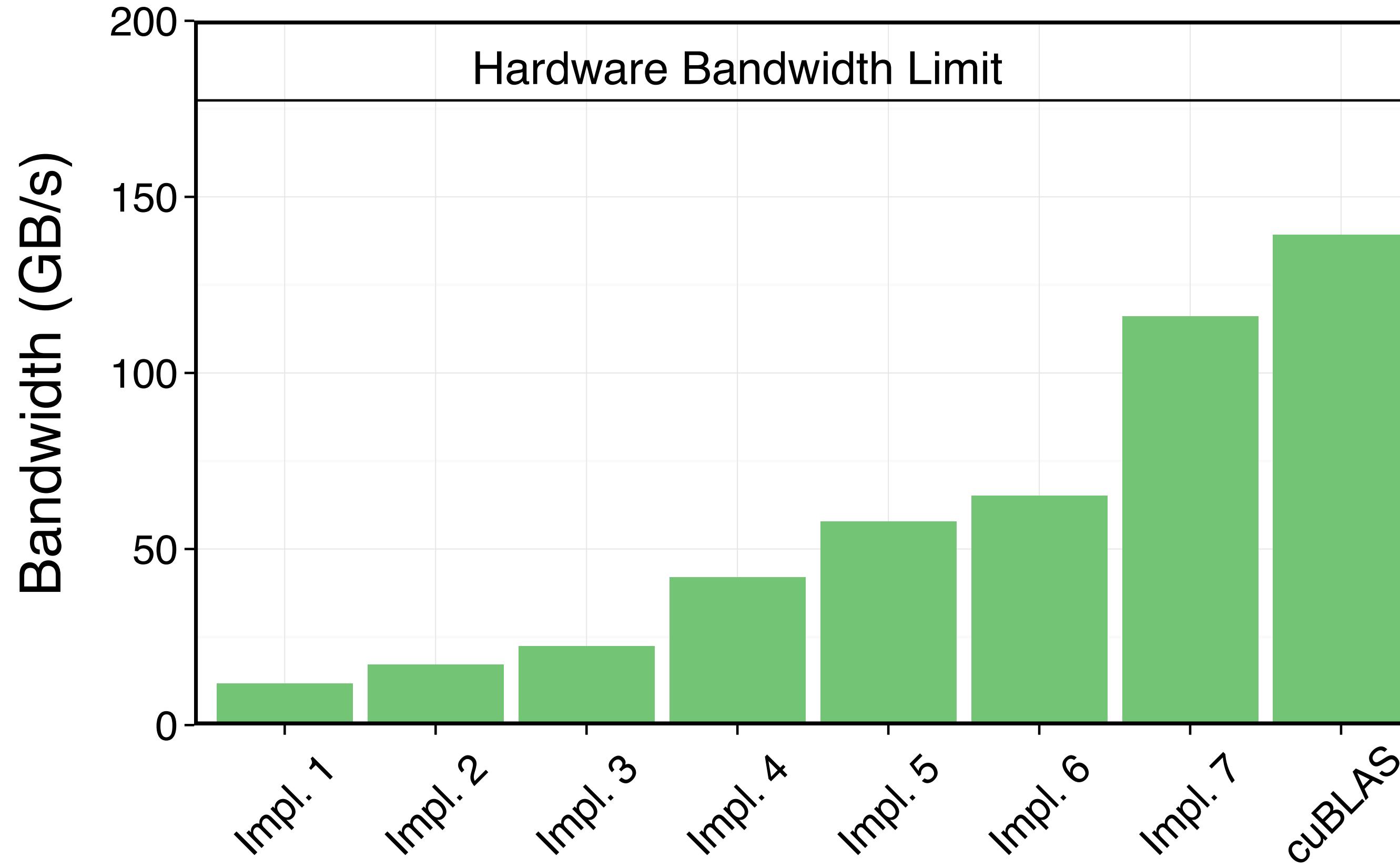
Unoptimized Implementation

```
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

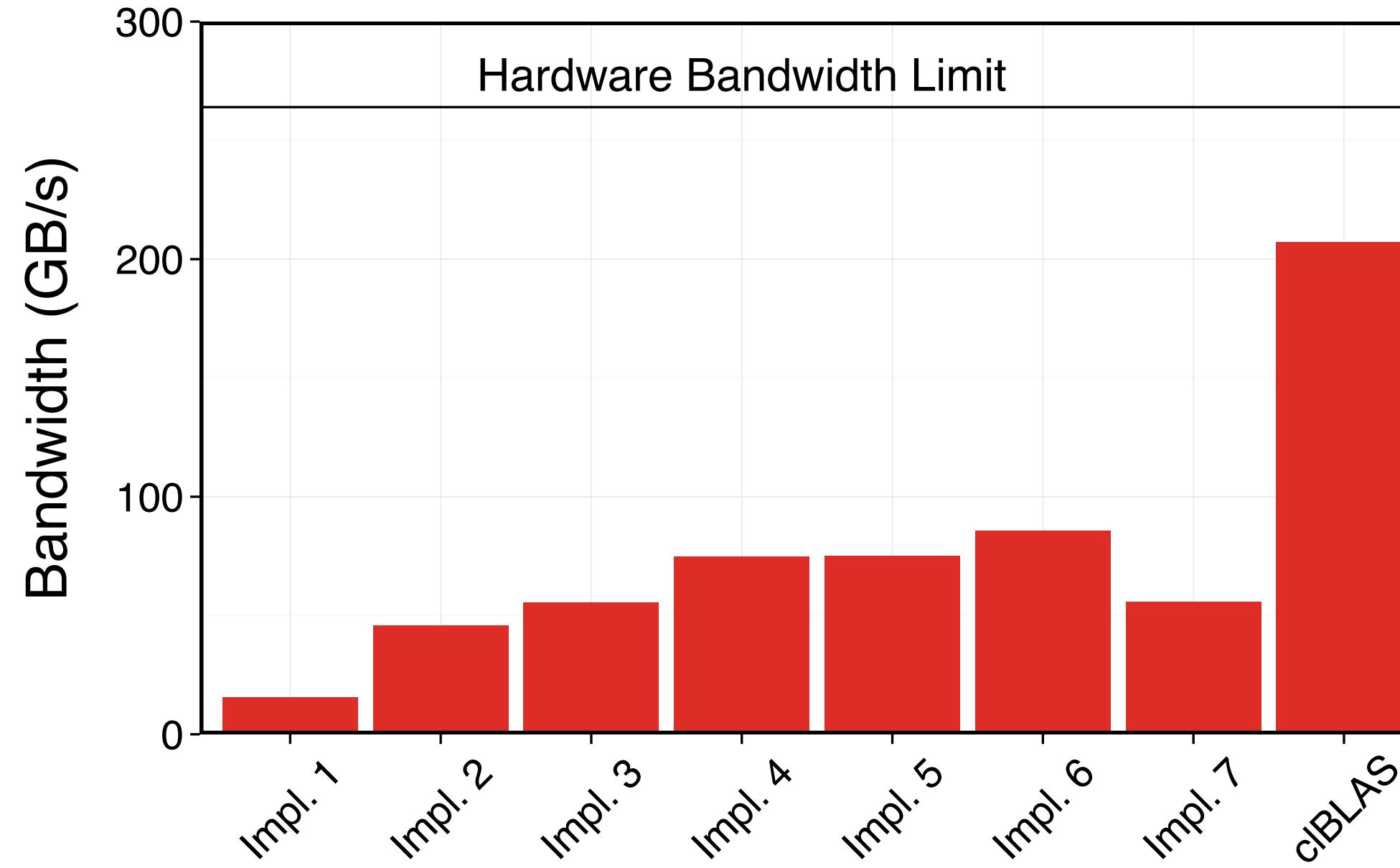
# Performance Results Nvidia



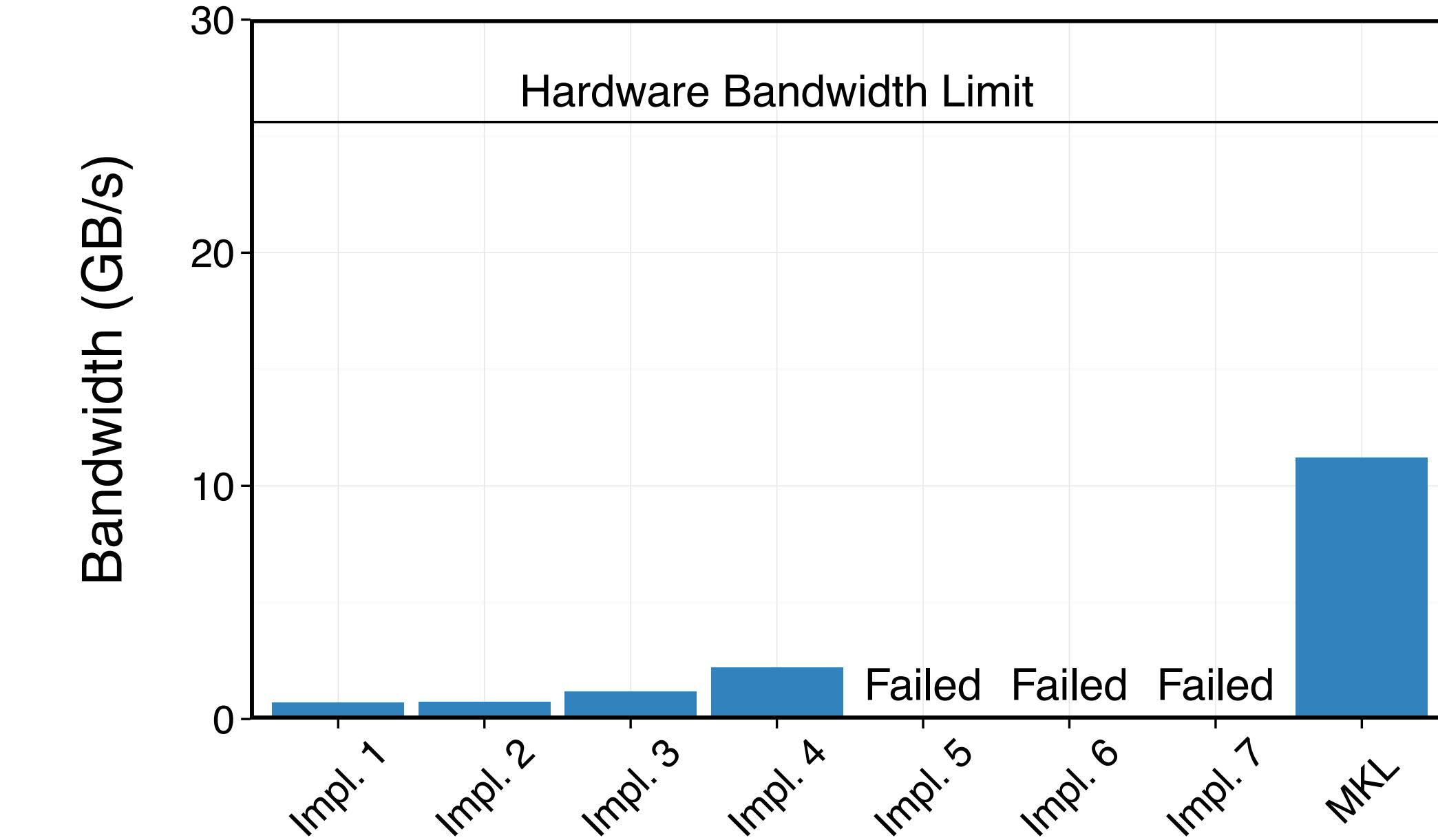
(a) Nvidia's GTX 480 GPU.

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

# Performance Results AMD and Intel



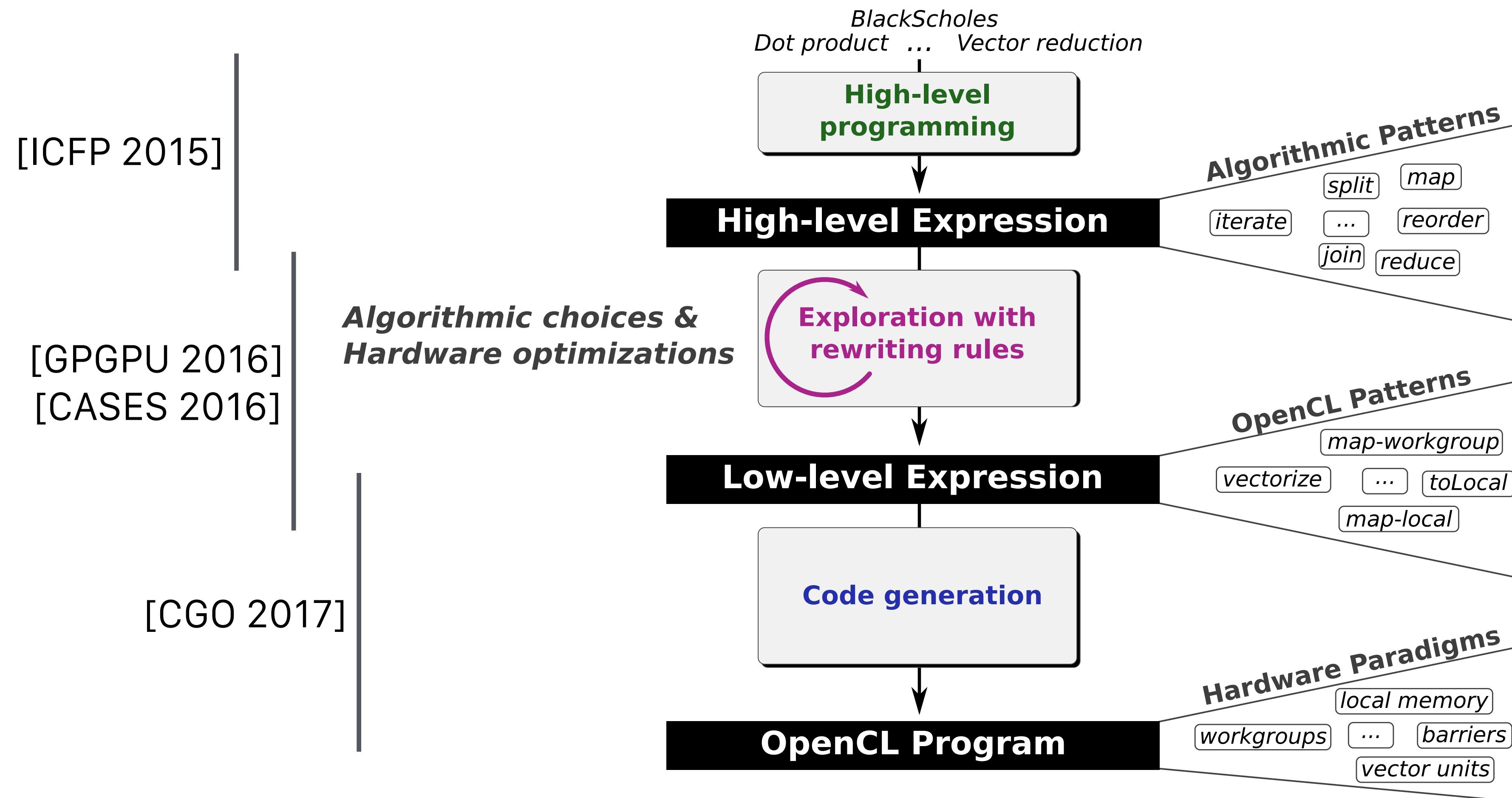
(b) AMD's HD 7970 GPU.



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

|  
rewrite rules      code generation

②

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

rewrite rules

code generation

②

```
vecSum = reduce ∘ join ∘ map-workgroup (
    join ∘ toGlobal (map-local (map-seq id)) ∘ split 1 ∘
    join ∘ map-warp (
        join ∘ map-lane (reduce-seq (+) 0) ∘ split 2 ∘ reorder-stride 1 ∘
        join ∘ map-lane (reduce-seq (+) 0) ∘ split 2 ∘ reorder-stride 2 ∘
        join ∘ map-lane (reduce-seq (+) 0) ∘ split 2 ∘ reorder-stride 4 ∘
        join ∘ map-lane (reduce-seq (+) 0) ∘ split 2 ∘ reorder-stride 8 ∘
        join ∘ map-lane (reduce-seq (+) 0) ∘ split 2 ∘ reorder-stride 16 ∘
        join ∘ map-lane (reduce-seq (+) 0) ∘ split 2 ∘ reorder-stride 32 ∘
    ) ∘ split 64 ∘
    join ∘ map-local (reduce-seq (+) 0) ∘ split 2 ∘ reorder-stride 64 ∘
    join ∘ toLocal (map-local (reduce-seq (+) 0)) ∘
    split (blockSize/128) ∘ reorder-stride 128 ∘
) ∘ split blockSize
```

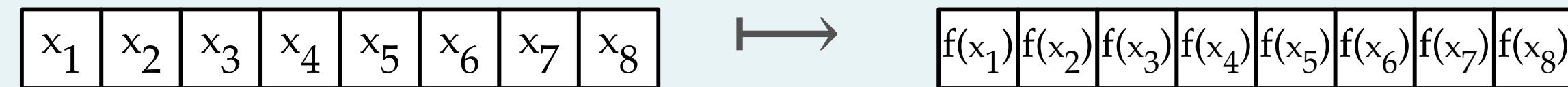
③

```
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);

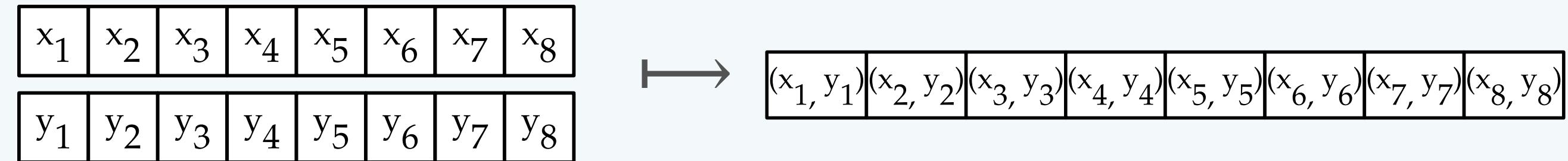
    if (WG_SIZE >= 256) {
        if (tid < 128) {
            l_data[tid] += l_data[tid+128]; }
        barrier(CLK_LOCAL_MEM_FENCE); }
    if (WG_SIZE >= 128) {
        if (tid < 64) {
            l_data[tid] += l_data[tid+ 64]; }
        barrier(CLK_LOCAL_MEM_FENCE); }
    if (tid < 32) {
        if (WG_SIZE >= 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)

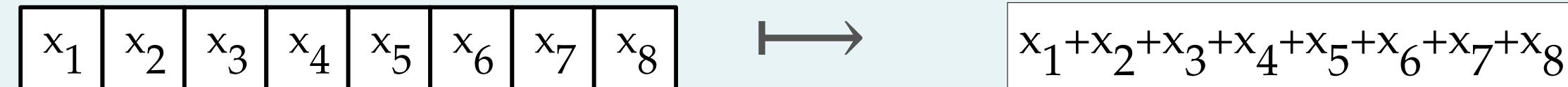
$\text{map}(f, x):$



$\text{zip}(x, y):$



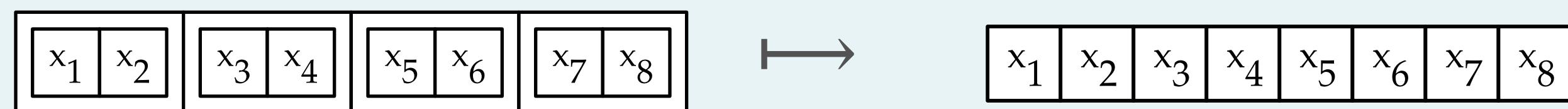
$\text{reduce}(+, 0, x):$



$\text{split}(n, x):$



$\text{join}(x):$



$\text{iterate}(f, n, x):$



$\text{reorder}(\sigma, x):$



# ① High-Level Programs

```
scal(a, vec) = map( $\lambda x \mapsto x * a$ , vec)
```

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

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

```
gemv(mat, x, y,  $\alpha$ ,  $\beta$ ) =  
  map(+, zip(  
    map( $\lambda \text{row} \mapsto \text{scal}(\alpha, \text{dotProduct}(\text{row}, x))$ , mat),  
    scal( $\beta$ , y) ))
```

# Walkthrough

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

|  
rewrite rules

code generation

②

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

|  
rewrite rules

②

```
vecSum = reduce ∘ join ∘ map-workgroup (
    join ∘ toGlobal (map-local (map-seq id)) ∘ split 1 ∘
    join ∘ map-warp (
        join ∘ map-lane (reduce-seq (+) 0) ∘ split 2 ∘ reorder-stride 1 ∘
        join ∘ map-lane (reduce-seq (+) 0) ∘ split 2 ∘ reorder-stride 2 ∘
        join ∘ map-lane (reduce-seq (+) 0) ∘ split 2 ∘ reorder-stride 4 ∘
        join ∘ map-lane (reduce-seq (+) 0) ∘ split 2 ∘ reorder-stride 8 ∘
        join ∘ map-lane (reduce-seq (+) 0) ∘ split 2 ∘ reorder-stride 16 ∘
        join ∘ map-lane (reduce-seq (+) 0) ∘ split 2 ∘ reorder-stride 32 ∘
    ) ∘ split 64 ∘
    join ∘ map-local (reduce-seq (+) 0) ∘ split 2 ∘ reorder-stride 64 ∘
    join ∘ toLocal (map-local (reduce-seq (+) 0)) ∘
    split (blockSize/128) ∘ reorder-stride 128 ∘
) ∘ 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:

$$\text{map } f \rightarrow \text{join} \circ \text{map} (\text{map } f) \circ \text{split } n$$

Map fusion rule:

$$\text{map } f \circ \text{map } g \rightarrow \text{map} (f \circ g)$$

Reduce rules:

$$\text{reduce } f z \rightarrow \text{reduce } f z \circ \text{reducePart } f z$$

$$\text{reducePart } f z \rightarrow \text{reducePart } f z \circ \text{reorder}$$

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

$$\text{reducePart } f z \rightarrow \text{iterate } n (\text{reducePart } f z)$$

## ② OpenCL Primitives

### Primitive

*mapGlobal*

*mapWorkgroup*

*mapLocal*

*mapSeq*

*reduceSeq*

*toLocal* , *toGlobal*

*mapVec*,  
*splitVec*, *joinVec*

### OpenCL concept

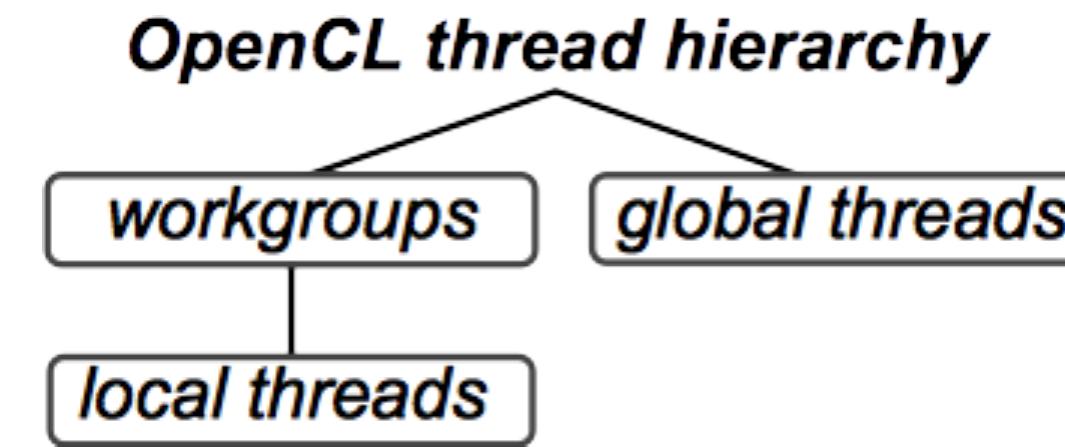
Work-items

Work-groups

Sequential implementations

Memory areas

Vectorisation



## ② OpenCL Rewrite Rules

- Express low-level implementation and optimisation choices

Map rules:

$$\text{map } f \rightarrow \text{mapWorkgroup } f \mid \text{mapLocal } f \mid \text{mapGlobal } f \mid \text{mapSeq } f$$

Local/ global memory rules:

$$\text{mapLocal } f \rightarrow \text{toLocal}(\text{mapLocal } f) \quad \text{mapLocal } f \rightarrow \text{toGlobal}(\text{mapLocal } f)$$

Vectorisation rule:

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

Fusion rule:

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

# Walkthrough

①

$\text{vecSum} = \text{reduce } (+) \ 0$

|  
rewrite rules

②

```
vecSum = reduce ∘ join ∘ map-workgroup (
    join ∘ toGlobal (map-local (map-seq id)) ∘ split 1 ∘
    join ∘ map-warp (
        join ∘ map-lane (reduce-seq (+) 0) ∘ split 2 ∘ reorder-stride 1 ∘
        join ∘ map-lane (reduce-seq (+) 0) ∘ split 2 ∘ reorder-stride 2 ∘
        join ∘ map-lane (reduce-seq (+) 0) ∘ split 2 ∘ reorder-stride 4 ∘
        join ∘ map-lane (reduce-seq (+) 0) ∘ split 2 ∘ reorder-stride 8 ∘
        join ∘ map-lane (reduce-seq (+) 0) ∘ split 2 ∘ reorder-stride 16 ∘
        join ∘ map-lane (reduce-seq (+) 0) ∘ split 2 ∘ reorder-stride 32 ∘
    ) ∘ split 64 ∘
    join ∘ map-local (reduce-seq (+) 0) ∘ split 2 ∘ reorder-stride 64 ∘
    join ∘ toLocal (map-local (reduce-seq (+) 0)) ∘
    split (blockSize/128) ∘ reorder-stride 128 ∘
) ∘ 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

$\text{vecSum} = \text{reduce } (+) \ 0$

rewrite rules

②

```
vecSum = reduce ∘ join ∘ map-workgroup (
    join ∘ toGlobal (map-local (map-seq id)) ∘ split 1 ∘
    join ∘ map-warp (
        join ∘ map-lane (reduce-seq (+) 0) ∘ split 2 ∘ reorder-stride 1 ∘
        join ∘ map-lane (reduce-seq (+) 0) ∘ split 2 ∘ reorder-stride 2 ∘
        join ∘ map-lane (reduce-seq (+) 0) ∘ split 2 ∘ reorder-stride 4 ∘
        join ∘ map-lane (reduce-seq (+) 0) ∘ split 2 ∘ reorder-stride 8 ∘
        join ∘ map-lane (reduce-seq (+) 0) ∘ split 2 ∘ reorder-stride 16 ∘
        join ∘ map-lane (reduce-seq (+) 0) ∘ split 2 ∘ reorder-stride 32 ∘
    ) ∘ split 64 ∘
    join ∘ map-local (reduce-seq (+) 0) ∘ split 2 ∘ reorder-stride 64 ∘
    join ∘ toLocal (map-local (reduce-seq (+) 0)) ∘
    split (blockSize/128) ∘ reorder-stride 128 ∘
) ∘ 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]; }
```

code generation

### 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

$\text{vecSum} = \text{reduce } (+) \ 0$

|  
rewrite rules

②

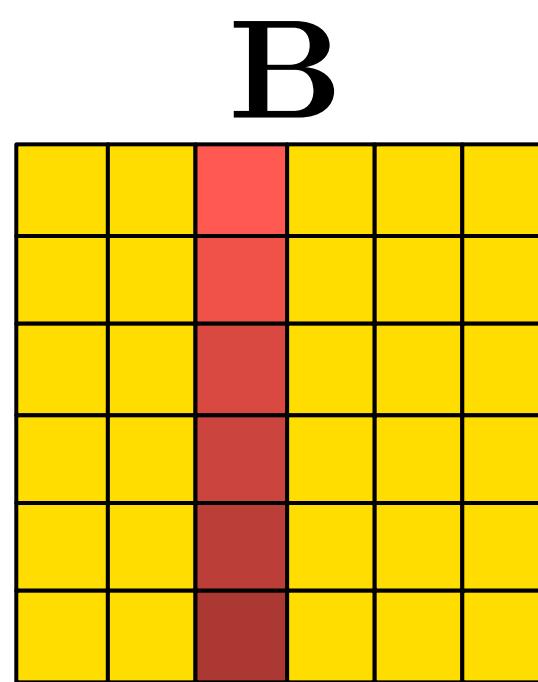
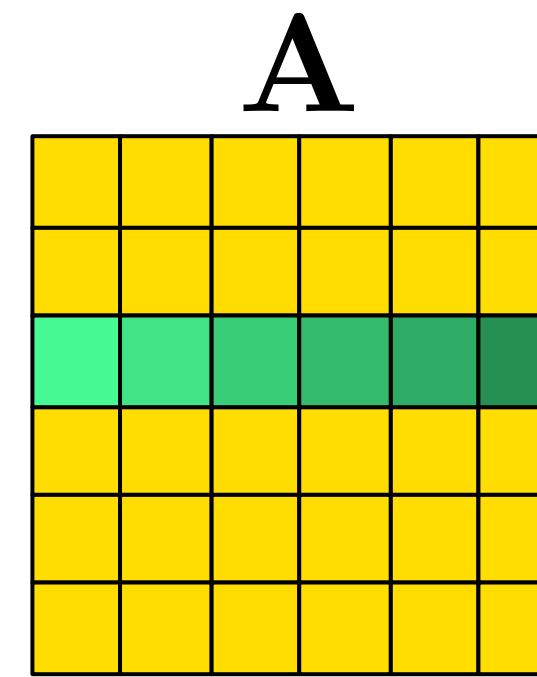
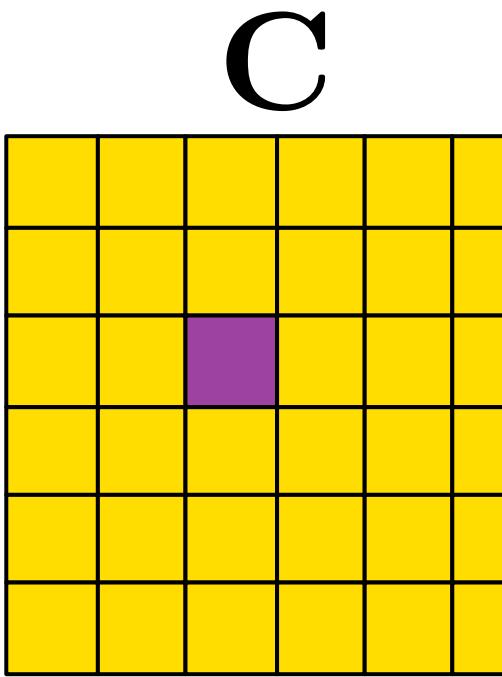
```
vecSum = reduce ∘ join ∘ map-workgroup (
    join ∘ toGlobal (map-local (map-seq id)) ∘ split 1 ∘
    join ∘ map-warp (
        join ∘ map-lane (reduce-seq (+) 0) ∘ split 2 ∘ reorder-stride 1 ∘
        join ∘ map-lane (reduce-seq (+) 0) ∘ split 2 ∘ reorder-stride 2 ∘
        join ∘ map-lane (reduce-seq (+) 0) ∘ split 2 ∘ reorder-stride 4 ∘
        join ∘ map-lane (reduce-seq (+) 0) ∘ split 2 ∘ reorder-stride 8 ∘
        join ∘ map-lane (reduce-seq (+) 0) ∘ split 2 ∘ reorder-stride 16 ∘
        join ∘ map-lane (reduce-seq (+) 0) ∘ split 2 ∘ reorder-stride 32 ∘
    ) ∘ split 64 ∘
    join ∘ map-local (reduce-seq (+) 0) ∘ split 2 ∘ reorder-stride 64 ∘
    join ∘ toLocal (map-local (reduce-seq (+) 0)) ∘
    split (blockSize/128) ∘ reorder-stride 128 ∘
) ∘ 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];
}
```

# Case Study: Matrix Multiplication



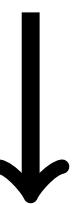
$A \times B =$   
 $\text{map}(\lambda \text{ rowA} \mapsto$   
 $\quad \text{map}(\lambda \text{ colB} \mapsto$   
 $\quad \quad \text{dotProduct}(\text{rowA}, \text{colB})$   
 $\quad \quad , \text{ transpose}(B))$   
 $\quad , A)$

# Tiling as a Rewrite Rules

Naïve matrix multiplication

```

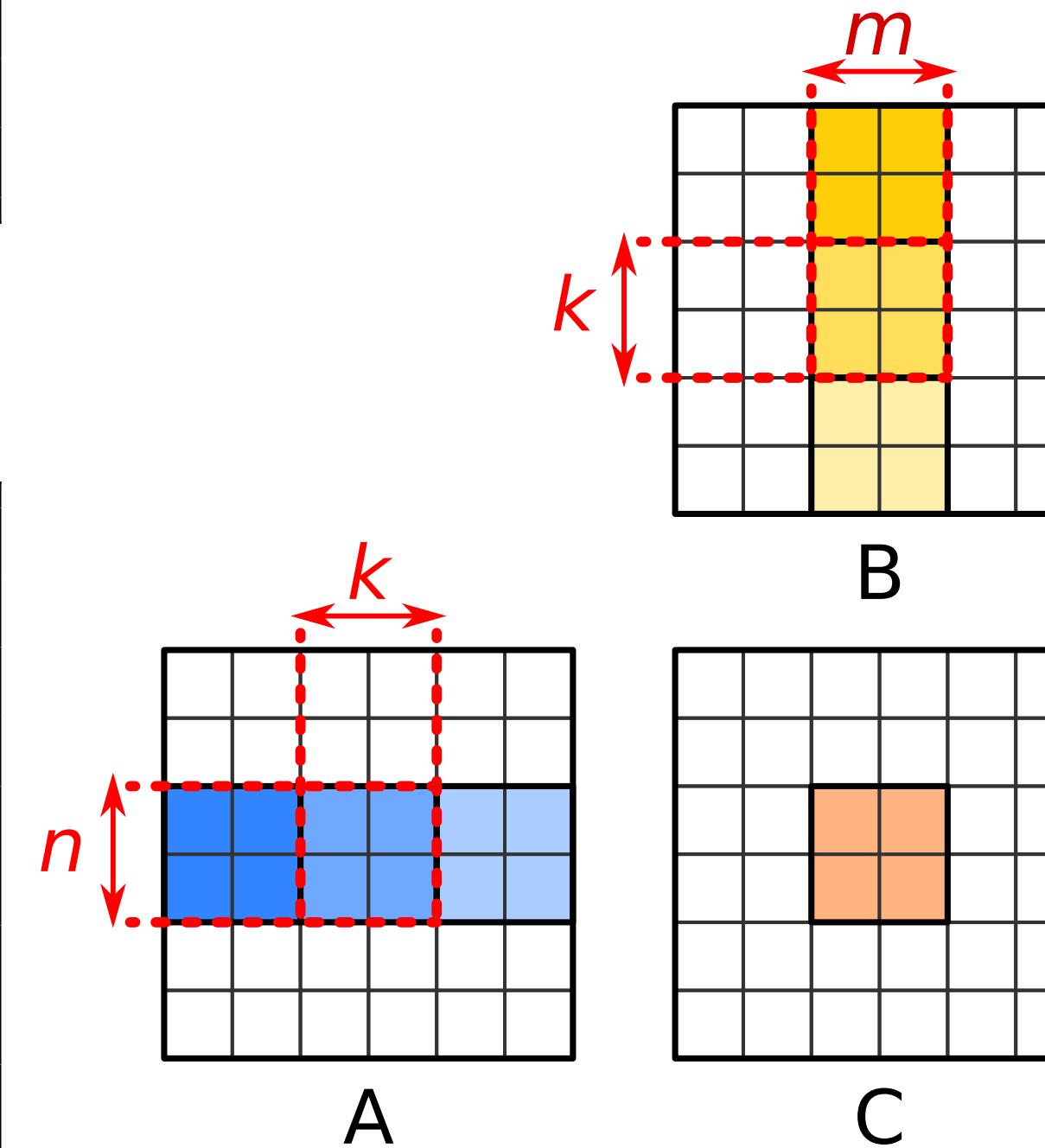
1 map(λ arow .
2   map(λ bcol .
3     reduce(+, 0) ∘ map(×) ∘ zip(arow, bcol)
4     , transpose(B))
5   , A)
```



Apply tiling rules

```

1 untile ∘ map(λ rowOfTilesA .
2   map(λ colOfTilesB .
3     toGlobal(copy2D) ∘
4     reduce(λ (tileAcc, (tileA, tileB)) .
5       map(map(+)) ∘ zip(tileAcc) ∘
6       map(λ as .
7         map(λ bs .
8           reduce(+, 0) ∘ map(×) ∘ zip(as, bs)
9           , toLocal(copy2D(tileB)))
10          , toLocal(copy2D(tileA)))
11          , 0, zip(rowOfTilesA, colOfTilesB))
12        ) ∘ tile(m, k, transpose(B))
13      ) ∘ tile(n, k, A)
```



# Register Blocking as a Rewrite Rules

```

1  until e  $\circ$  map( $\lambda$  rowOfTilesA .
2    map( $\lambda$  colOfTilesB .
3      toGlobal(copy2D)  $\circ$ 
4        reduce( $\lambda$  (tileAcc, (tileA, tileB)) .
5          map(map(+))  $\circ$  zip(tileAcc)  $\circ$ 
6            map( $\lambda$  as .
7              map( $\lambda$  bs .
8                reduce(+, 0)  $\circ$  map( $\times$ )  $\circ$  zip(as, bs)
9                  , toLocal(copy2D(tileB)))
10                 , toLocal(copy2D(tileA)))
11                 ,0, zip(rowOfTilesA, colOfTilesB))
12               )  $\circ$  tile(m, k, transpose(B))
13             )  $\circ$  tile(n, k, A)

```

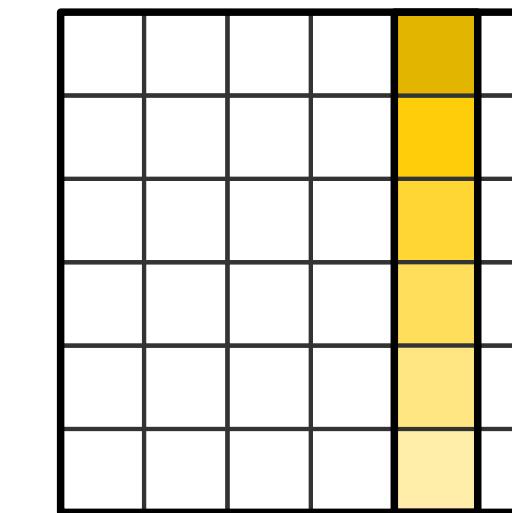


Apply blocking rules

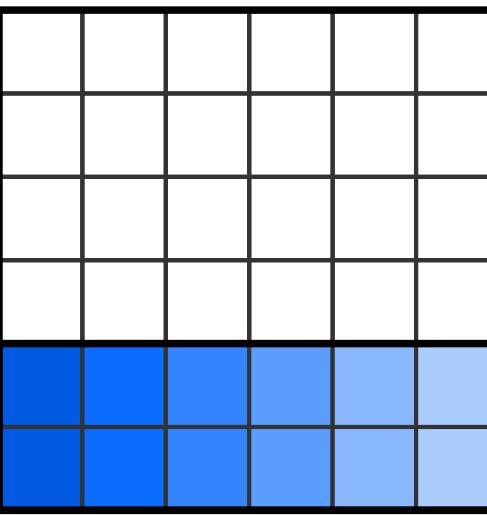
```

1  until e  $\circ$  map( $\lambda$  rowOfTilesA .
2    map( $\lambda$  colOfTilesB .
3      toGlobal(copy2D)  $\circ$ 
4        reduce( $\lambda$  (tileAcc, (tileA, tileB)) .
5          map(map(+))  $\circ$  zip(tileAcc)  $\circ$ 
6            map( $\lambda$  aBlocks .
7              map( $\lambda$  bs .
8                reduce(+, 0)  $\circ$ 
9                  map( $\lambda$  (aBlock, b) .
10                   map( $\lambda$  (a,bp) . a  $\times$  bp
11                     , zip(aBlock, toPrivate(id(b))))
12                     )  $\circ$  zip(transpose(aBlocks), bs)
13                     , toLocal(copy2D(tileB)))
14                     , split(l, toLocal(copy2D(tileA))))
15                     ,0, zip(rowOfTilesA, colOfTilesB))
16                   )  $\circ$  tile(m, k, transpose(B))
17                 )  $\circ$  tile(n, k, A)

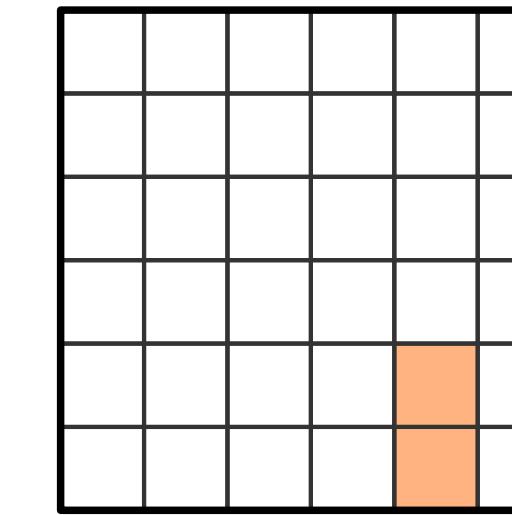
```



B



A



C

# Register Blocking as a Rewrite Rules

*registerBlocking =*

$\text{Map}(f) \Rightarrow \text{Join}() \circ \text{Map}(\text{Map}(f)) \circ \text{Split}(k)$

$\text{Map}(a \mapsto \text{Map}(b \mapsto f(a, b))) \Rightarrow \text{Transpose}() \circ \text{Map}(b \mapsto \text{Map}(a \mapsto f(a, b)))$

$\text{Map}(f \circ g) \Rightarrow \text{Map}(f) \circ \text{Map}(g)$

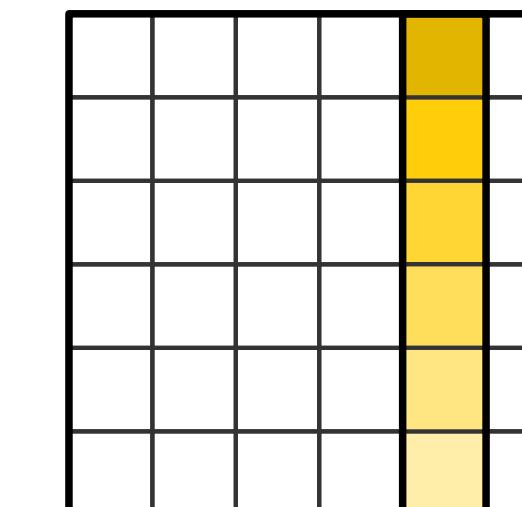
$\text{Map}(\text{Reduce}(f)) \Rightarrow \text{Transpose}() \circ \text{Reduce}((\text{acc}, x) \mapsto \text{Map}(f) \circ \text{Zip}(\text{acc}, x))$

$\text{Map}(\text{Map}(f)) \Rightarrow \text{Transpose}() \circ \text{Map}(\text{Map}(f)) \circ \text{Transpose}()$

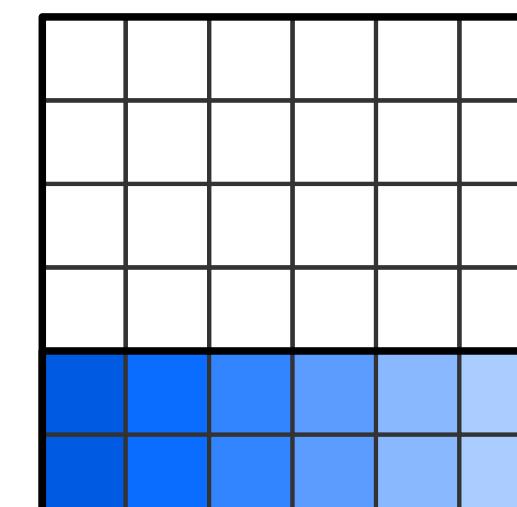
$\text{Transpose}() \circ \text{Transpose}() \Rightarrow id$

$\text{Reduce}(f) \circ \text{Map}(g) \Rightarrow \text{Reduce}((\text{acc}, x) \mapsto f(\text{acc}, g(x)))$

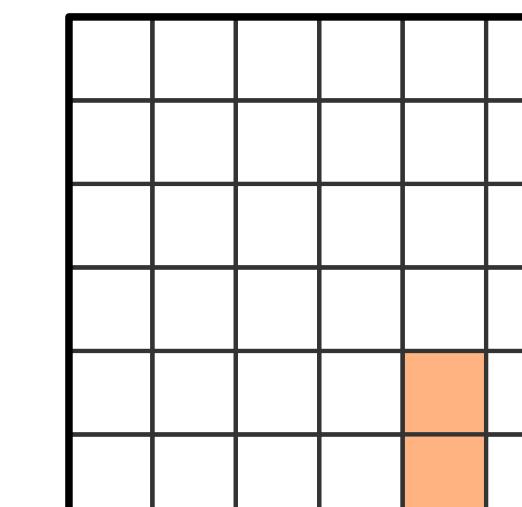
$\text{Map}(f) \circ \text{Map}(g) \Rightarrow \text{Map}(f \circ g)$



B

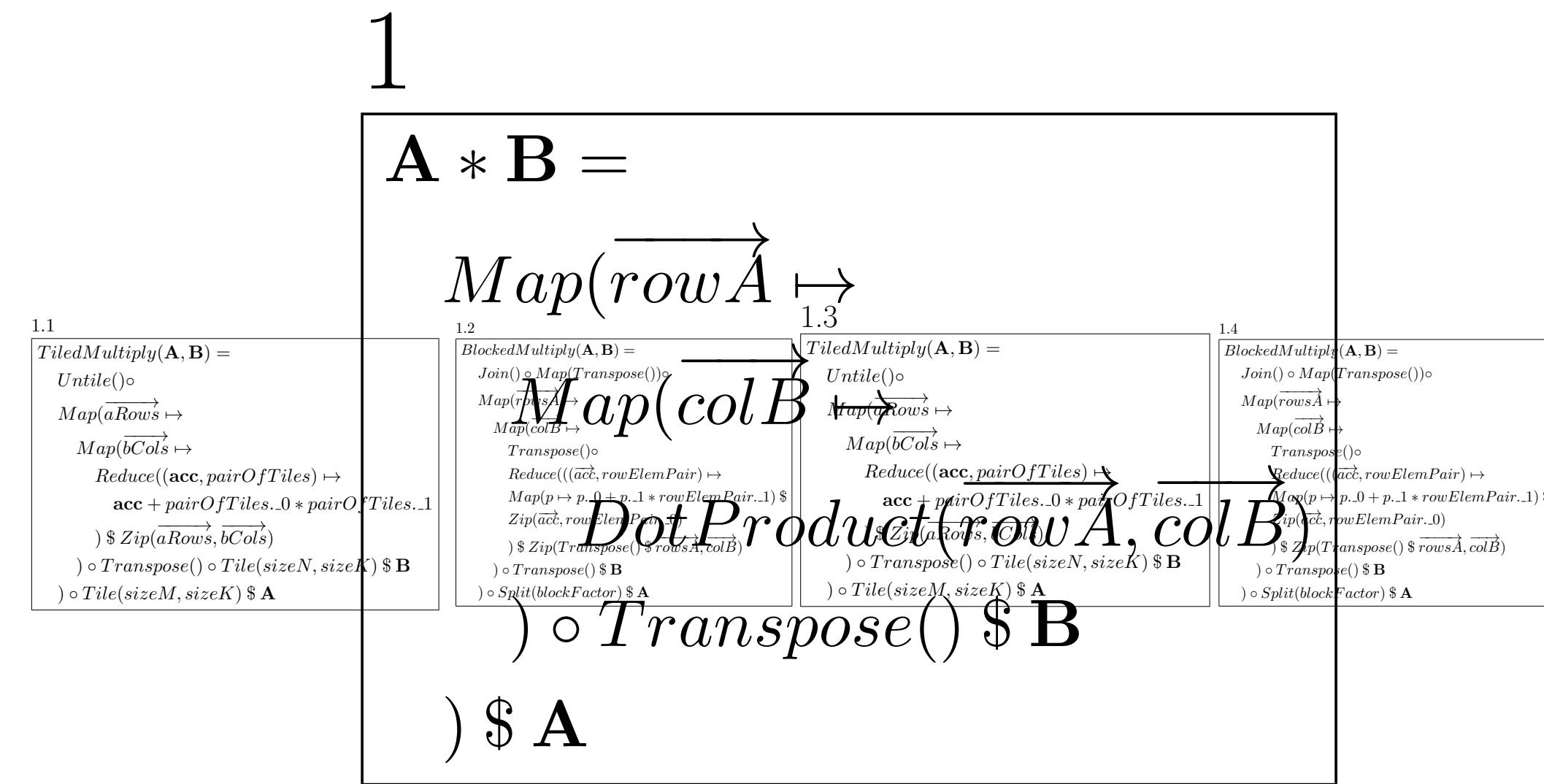
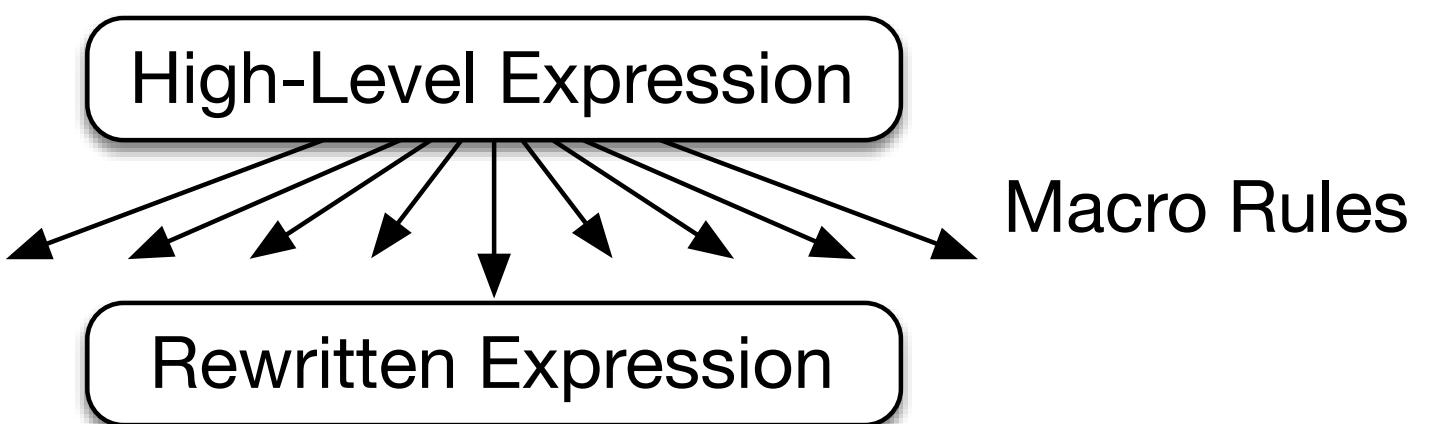


A

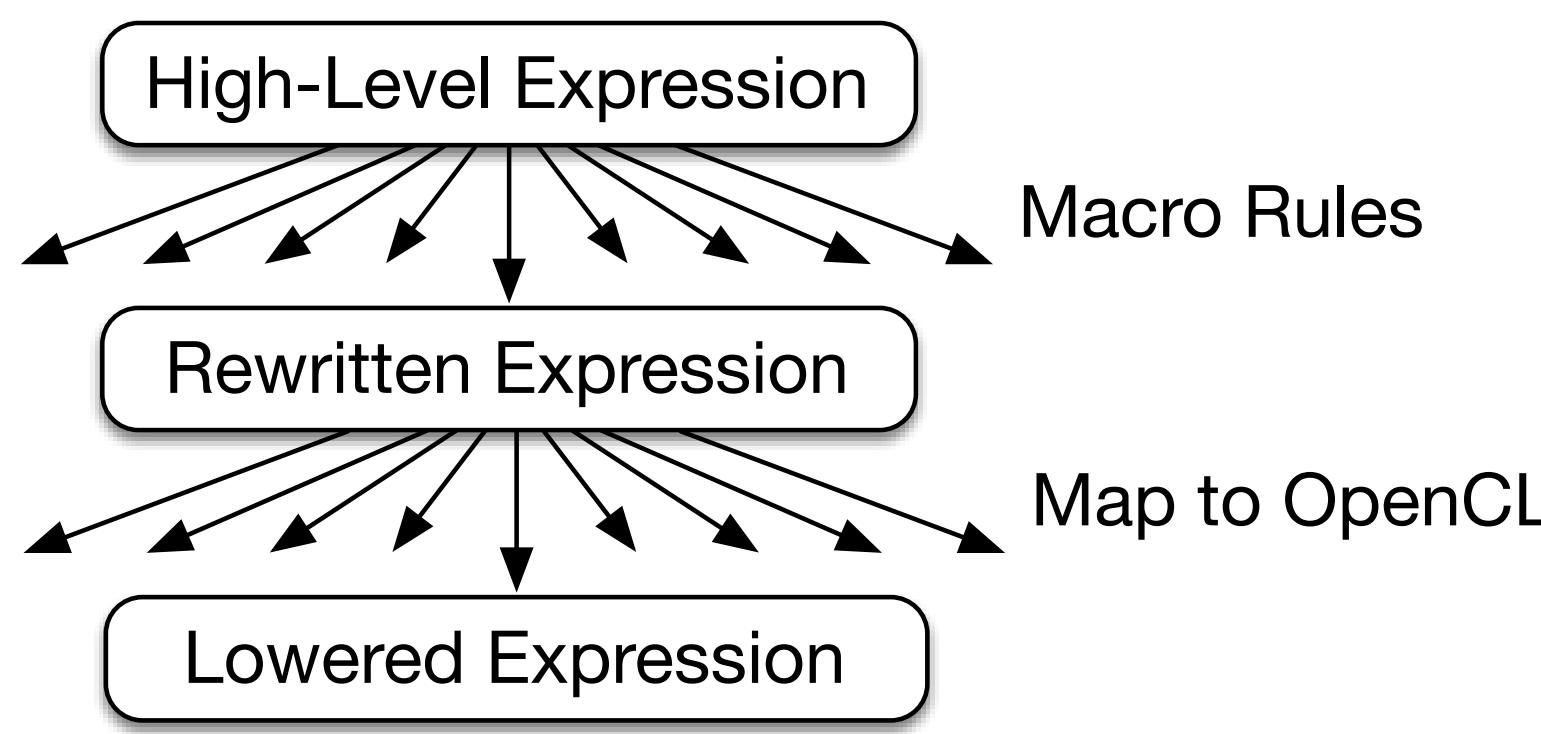


C

# Exploration Strategy



# Exploration Strategy



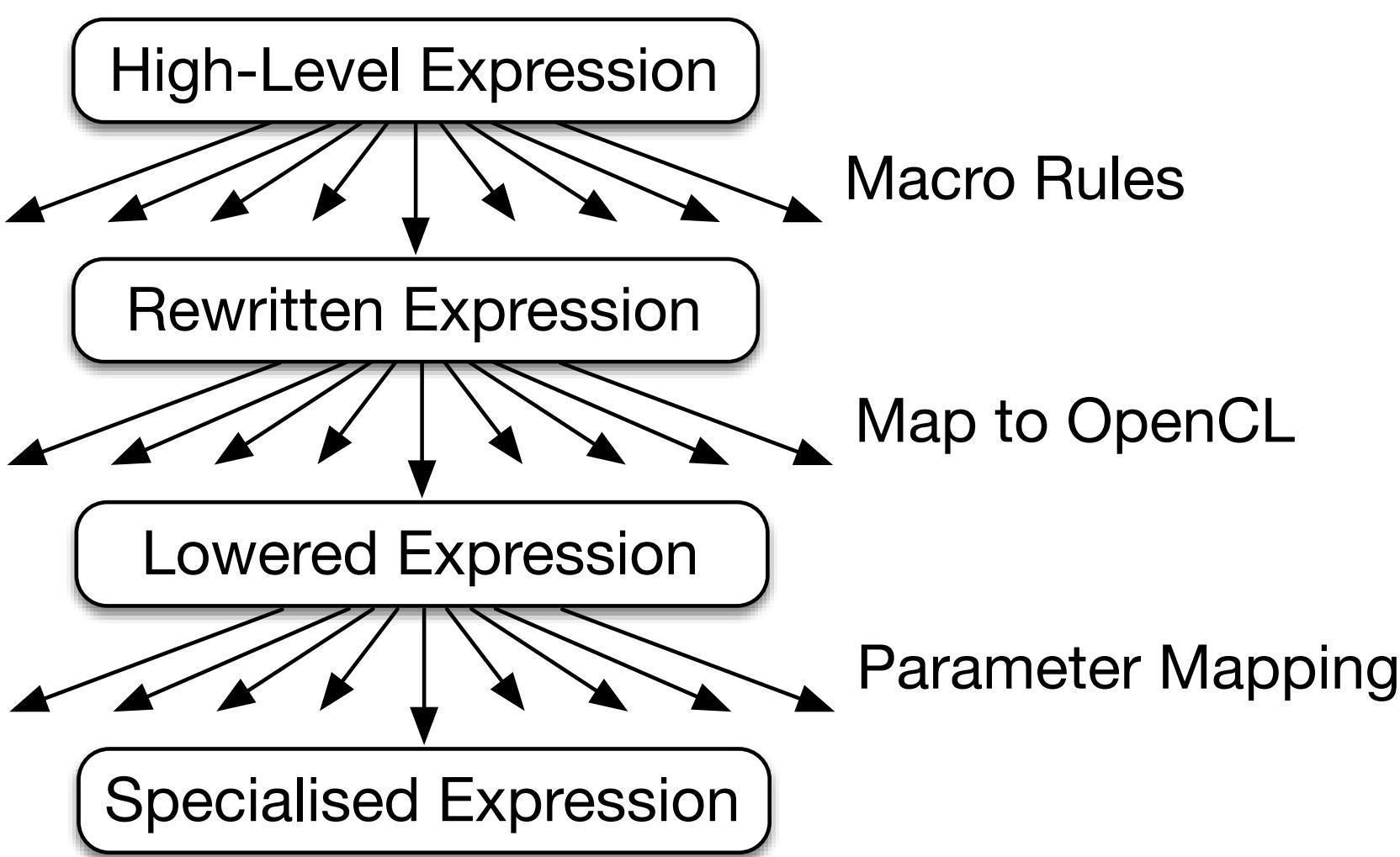
1.3

*TiledMultiply(A, B) =*

$$\begin{aligned}
 & \text{Untile}() \circ \\
 & \xrightarrow{\substack{1.3.1 \\ TiledMultiply(A, B) = \\ Untile() \circ \\ MapWrg(1)(aRows \mapsto \\ MapWrg(0)(bCols \mapsto \\ MapWrg(0)(bCols \mapsto \\ ReduceSeq(acc, pairOfTiles \mapsto \\ acc + toLocal(pairOfTiles..0) \\ * toLocal(pairOfTiles..1) \\ ) \$ Zip(aRows, bCols) \\ ) \$ Transpose() \circ Tile(sizeN, sizeK) \$ B \\ ) \circ Tile(sizeM, sizeK) \$ A}} \\
 & \xrightarrow{\substack{1.3.2 \\ TiledMultiply(A, B) = \\ Untile() \circ \\ MapWrg(1)(aRows \mapsto \\ MapWrg(0)(bCols \mapsto \\ MapWrg(0)(bCols \mapsto \\ ReduceSeq(acc, pairOfTiles \mapsto \\ acc + toLocal(pairOfTiles..0) \\ * toLocal(pairOfTiles..1) \\ ) \$ Zip(aRows, bCols) \\ ) \$ Transpose() \circ Tile(sizeN, sizeK) \$ B \\ ) \circ Tile(sizeM, sizeK) \$ A}} \\
 & \xrightarrow{\substack{1.3.3 \\ TiledMultiply(A, B) = \\ Untile() \circ \\ MapWrg(1)(aRows \mapsto \\ MapWrg(0)(bCols \mapsto \\ MapWrg(0)(bCols \mapsto \\ ReduceSeq(acc, pairOfTiles \mapsto \\ acc + toLocal(pairOfTiles..0) \\ * toLocal(pairOfTiles..1) \\ ) \$ Zip(aRows, bCols) \\ ) \$ Transpose() \circ Tile(sizeN, sizeK) \$ B \\ ) \circ Tile(sizeM, sizeK) \$ A}}
 \end{aligned}$$

The diagram shows three parallel derivation paths (1.3.1, 1.3.2, and 1.3.3) for the *TiledMultiply(A, B)* expression. Each path involves an *Untile()* step followed by a series of *Map*, *ReduceSeq*, *Zip*, *Transpose*, and *Tile* operations. The paths differ in the order and specific details of these operations, with some steps being shared or derived from others. Red circles highlight specific parts of the expressions, such as the *Map* and *ReduceSeq* steps, indicating points of interest or focus in the exploration process.

# Exploration Strategy



## 1.3.2

$TiledMultiply(\mathbf{A}, \mathbf{B}) =$   
 $Untile() \circ$

1.3.2.1  $MapWrg(1)(\overrightarrow{aRows} \mapsto$

$TiledMultiply(\mathbf{A}, \mathbf{B}) =$   
 $Untile() \circ$   
 $MapWrg(1)(\overrightarrow{aRows} \mapsto$   
 $MapWrg(0)(\overrightarrow{bCols} \mapsto$   
 $ReduceSeq((acc, pairOfTiles) \mapsto$   
 $acc + toLocal(pairOfTiles..0)$   
 $* toLocal(pairOfTiles..1)$   
 $) \$ Zip(\overrightarrow{aRows}, \overrightarrow{bCols})$   
 $) \circ Transpose() \circ Tile(128, 16) \$ \mathbf{B}$   
 $) \circ Tile(128, 16) \$ \mathbf{A}$

1.3.2.4  $TiledMultiply(\mathbf{A}, \mathbf{B}) =$

$Untile() \circ$   
 $MapWrg(1)(\overrightarrow{aRows} \mapsto$   
 $MapWrg(0)(\overrightarrow{bCols} \mapsto$   
 $ReduceSeq((acc, pairOfTiles) \mapsto$   
 $acc + toLocal(pairOfTiles..0)$   
 $* toLocal(pairOfTiles..1)$   
 $) \$ Zip(\overrightarrow{aRows}, \overrightarrow{bCols})$   
 $) \circ Transpose() \circ Tile(128, 16) \$ \mathbf{B}$   
 $) \circ Tile(128, 16) \$ \mathbf{A}$

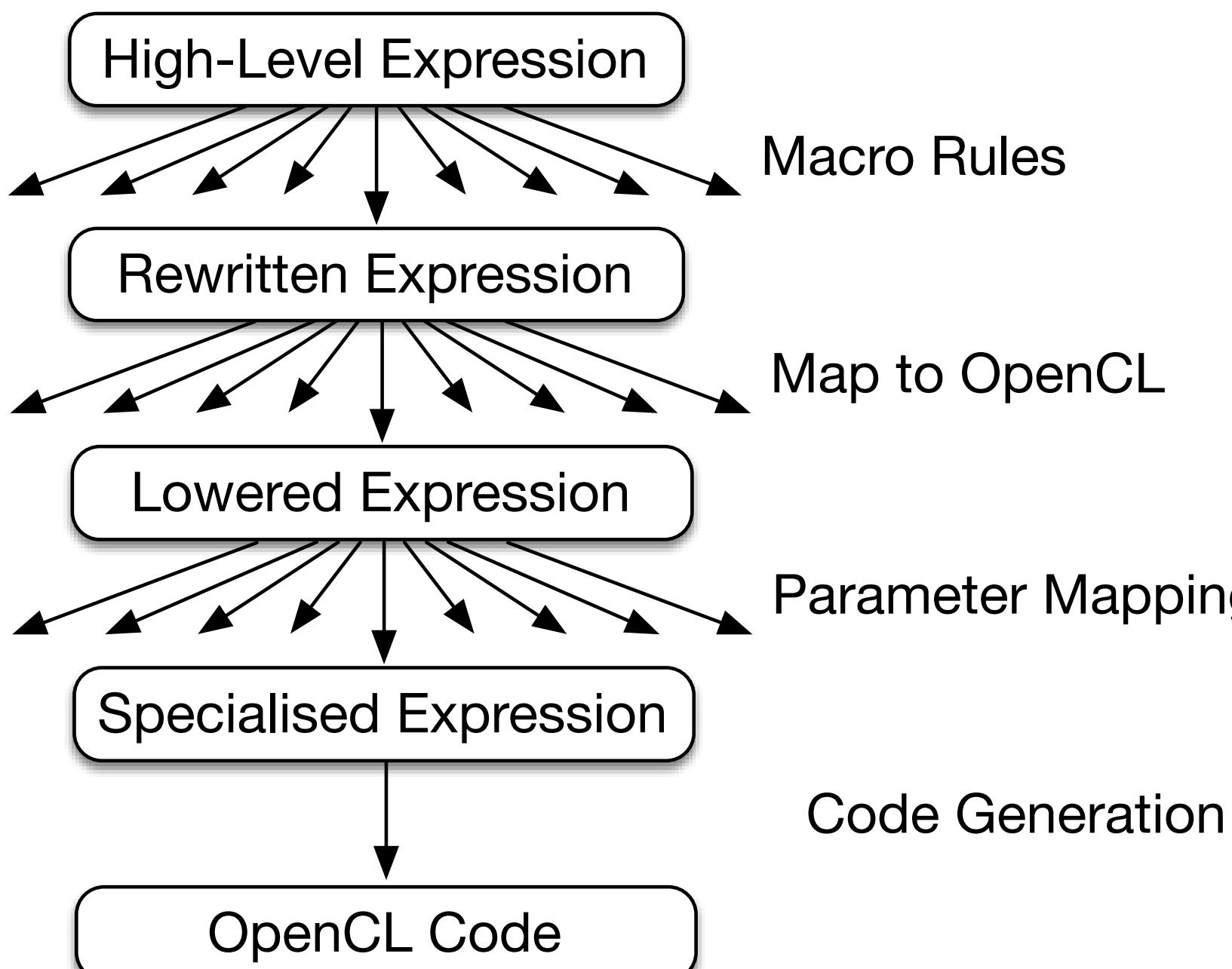
1.3.2.5  $TiledMultiply(\mathbf{A}, \mathbf{B}) =$

$Untile() \circ$   
 $MapWrg(1)(\overrightarrow{aRows} \mapsto$   
 $MapWrg(0)(\overrightarrow{bCols} \mapsto$   
 $ReduceSeq((acc, pairOfTiles) \mapsto$   
 $acc + toLocal(pairOfTiles..0)$   
 $* toLocal(pairOfTiles..1)$   
 $) \$ Zip(\overrightarrow{aRows}, \overrightarrow{bCols})$   
 $) \circ Transpose() \circ Tile(128, 16) \$ \mathbf{B}$   
 $) \circ Tile(128, 16) \$ \mathbf{A}$

1.3.2.6  $TiledMultiply(\mathbf{A}, \mathbf{B}) =$

$Untile() \circ$   
 $MapWrg(1)(\overrightarrow{aRows} \mapsto$   
 $MapWrg(0)(\overrightarrow{bCols} \mapsto$   
 $ReduceSeq((acc, pairOfTiles) \mapsto$   
 $acc + toLocal(pairOfTiles..0)$   
 $* toLocal(pairOfTiles..1)$   
 $) \$ Zip(\overrightarrow{aRows}, \overrightarrow{bCols})$   
 $) \circ Transpose() \circ Tile(128, 16) \$ \mathbf{B}$   
 $) \circ Tile(128, 16) \$ \mathbf{A}$

# Exploration Strategy



## 1.3.2.5

```

1 kernel mm_amd_opt(global float *A, B, C,
2   int K, M, N);
3   local float tileA [512];
4   local float tileB [512];
5
6   private float acc_0; ...; acc_31;
7   private float blockOfB_0; ...; blockOfB_3;
8   private float blockOfA_0; ...; blockOfA_7;
9
10  int lid0 = local_id(0); lid1 = local_id(1);
11  int wid0 = group_id(0); widx = group_id(1);
12  for (int w1=wid1; w1<M/64; w1+=num_grps(1)) {
13    for (int w0=wid0; w0<N/64; w0+=num_grps(0)) {
14      acc_0 = 0.0f; acc_31 = 0.0f;
15      for (int i=0; i<K/8; i++) {
16        vstore4(vload4(lid1*M/4+2*i*M+16*w1+lid0,A), 16*lid1+lid0, tileA);
17        vstore4(vload4(lid1*N/4+2*i*N+16*w0+lid0,B), 16*lid1+lid0, tileB);
18        barrier (...); MapWrg(1)(aRows) ↗
19        acc_0 += blockOfA_0 * blockOfB_0; ...; acc_28 += blockOfA_7 * blockOfB_0;
20        acc_1 += blockOfA_0 * blockOfB_1; ...; acc_29 += blockOfA_7 * blockOfB_1;
21        acc_2 += blockOfA_0 * blockOfB_2; ...; acc_30 += blockOfA_7 * blockOfB_2;
22        acc_3 += blockOfA_0 * blockOfB_3; ...; acc_31 += blockOfA_7 * blockOfB_3;
23      }
24      barrier (...); MapWrg(0)(bCols) ↗
25      acc_0 += blockOfA_0 * blockOfB_0; ...; acc_28 += blockOfA_7 * blockOfB_0;
26      acc_1 += blockOfA_0 * blockOfB_1; ...; acc_29 += blockOfA_7 * blockOfB_1;
27      acc_2 += blockOfA_0 * blockOfB_2; ...; acc_30 += blockOfA_7 * blockOfB_2;
28      acc_3 += blockOfA_0 * blockOfB_3; ...; acc_31 += blockOfA_7 * blockOfB_3;
29    }
30    barrier (...); ReduceSeq((acc, pairOfTiles) ↗
31    acc + toLocal(pairOfTiles..0)
32  }
33  barrier (...); acc + toLocal(pairOfTiles..1) ↗
34  ) $ Zip(aRows, bCols)
35
36  C[ 0+8*lid1*N+64*w0+64*w1*N+0*N+lid0]=acc_0; ...; C[ 0+8*lid1*N+64*w0+64*w1*N+7*N+lid0]=acc_28;
37  C[16+8*lid1*N+64*w0+64*w1*N+0*N+lid0]=acc_29; ...; C[16+8*lid1*N+64*w0+64*w1*N+7*N+lid0]=acc_31;
38  C[32+8*lid1*N+64*w0+64*w1*N+0*N+lid0]=acc_32; ...; C[32+8*lid1*N+64*w0+64*w1*N+7*N+lid0]=acc_33;
39  C[48+8*lid1*N+64*w0+64*w1*N+0*N+lid0]=acc_34; ...; C[48+8*lid1*N+64*w0+64*w1*N+7*N+lid0]=acc_35;
40 } } } ) $ Transpose() $ Tile(128,16) $ A
41
42 ) $ Tile(128,16) $ B
43
44 ) $ Transpose() $ Tile(128,16) $ C
45
46 ) $ Tile(128,16) $ D
  
```

This code block shows the generated OpenCL kernel for a tiled matrix multiplication. The kernel performs a reduction operation across tiles of size 128x16. It uses local memory for tiles and accumulates results in global memory. The code is annotated with arrows indicating the mapping from the high-level expression stages to specific parts of the generated code. The annotations include **MapWrg(1)(aRows)**, **MapWrg(0)(bCols)**, **ReduceSeq((acc, pairOfTiles)**, **acc + toLocal(pairOfTiles..0)**, **acc + toLocal(pairOfTiles..1)**, **Zip(aRows, bCols)**, **Transpose()**, **Tile(128, 16)**, and **A**, **B**, **C**, **D**.

# 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

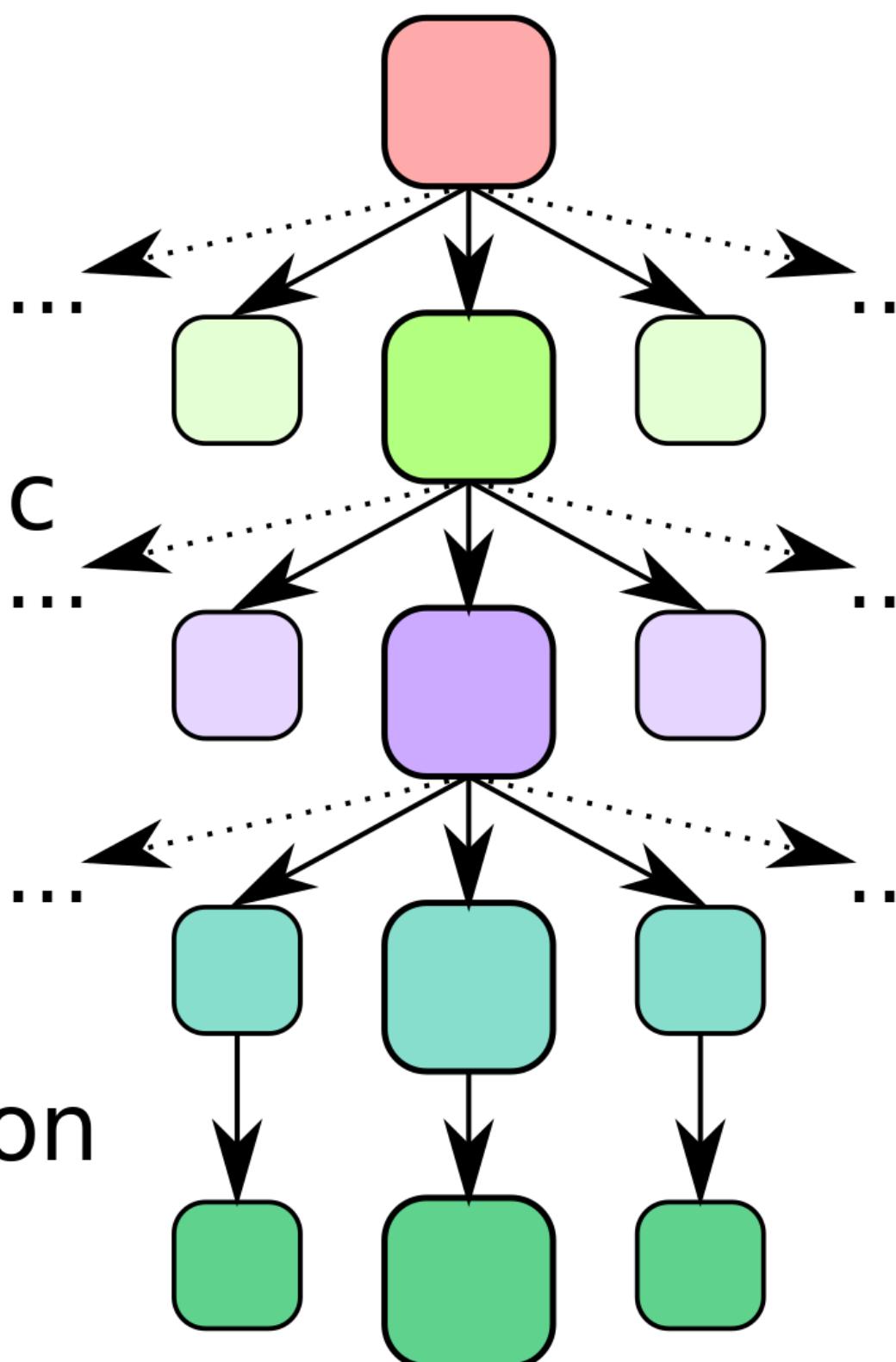
Phases:

Algorithmic  
Exploration

OpenCL specific  
Exploration

Parameter  
Exploration

Code Generation



Program Variants:

High-Level Program 1

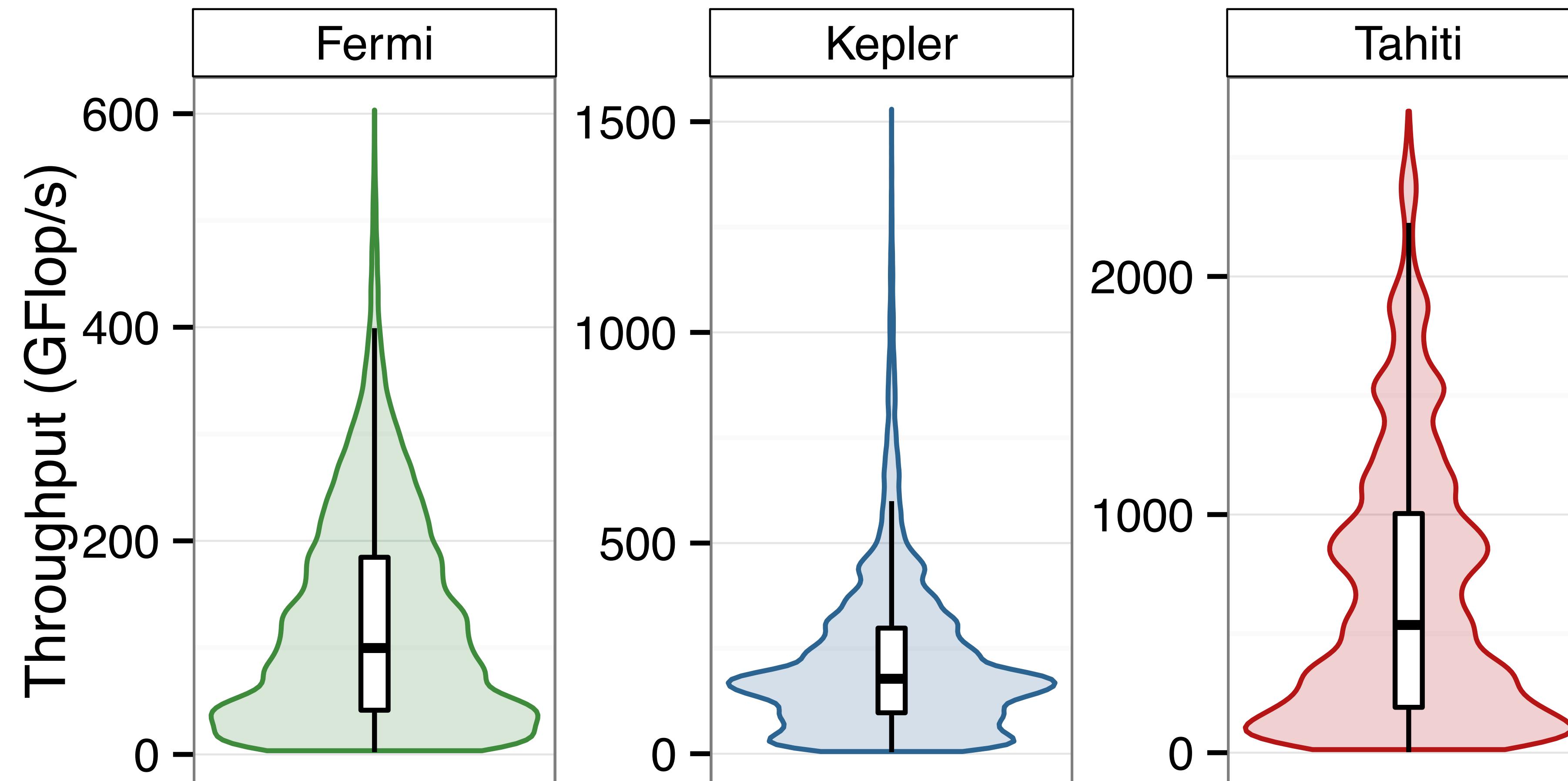
Algorithmic Rewritten Program 8

OpenCL Specific Program 760

Fully Specialized Program 46,000

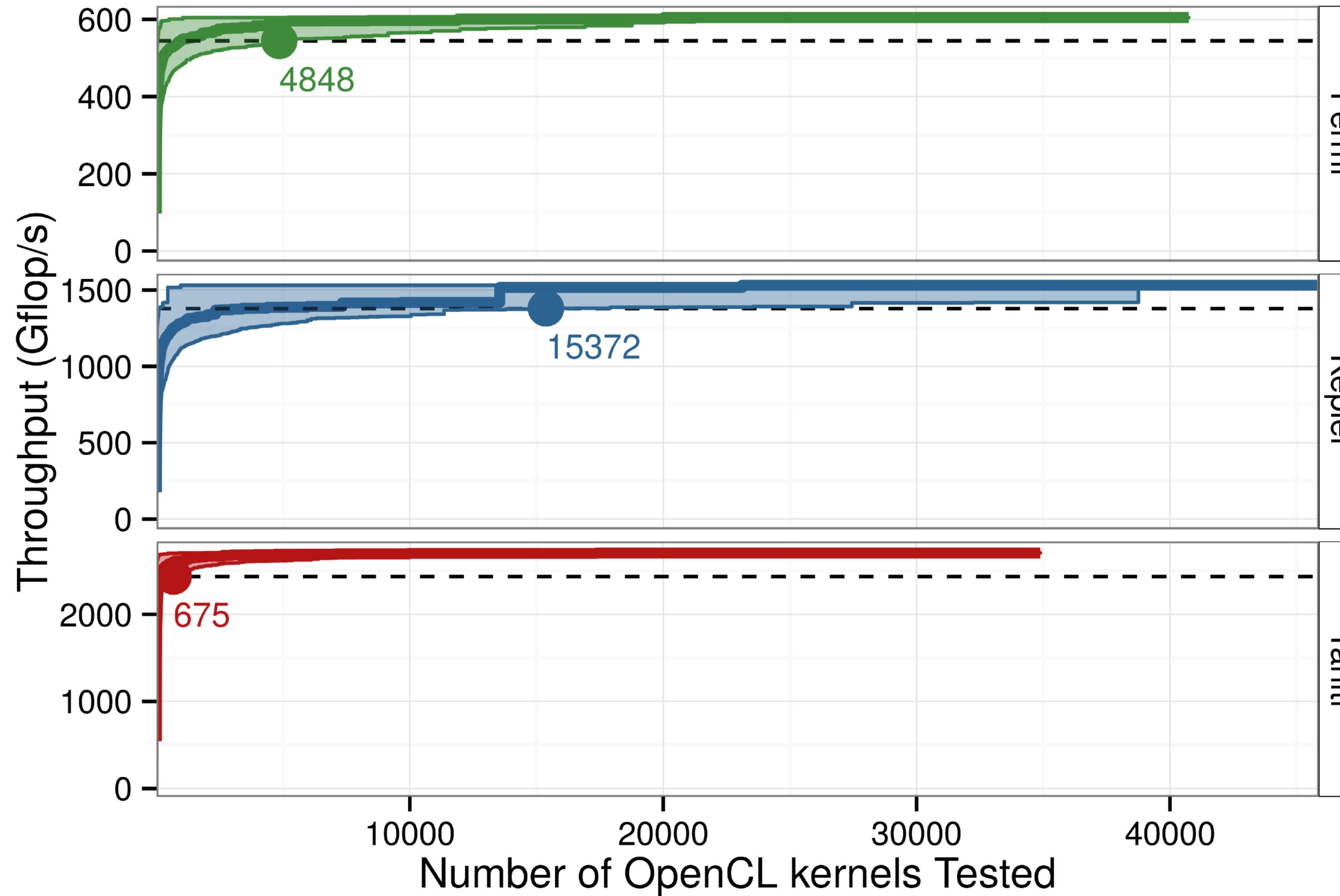
OpenCL Code 46,000

# Exploration Space for Matrix Multiplication



Only few OpenCL kernel with very good performance

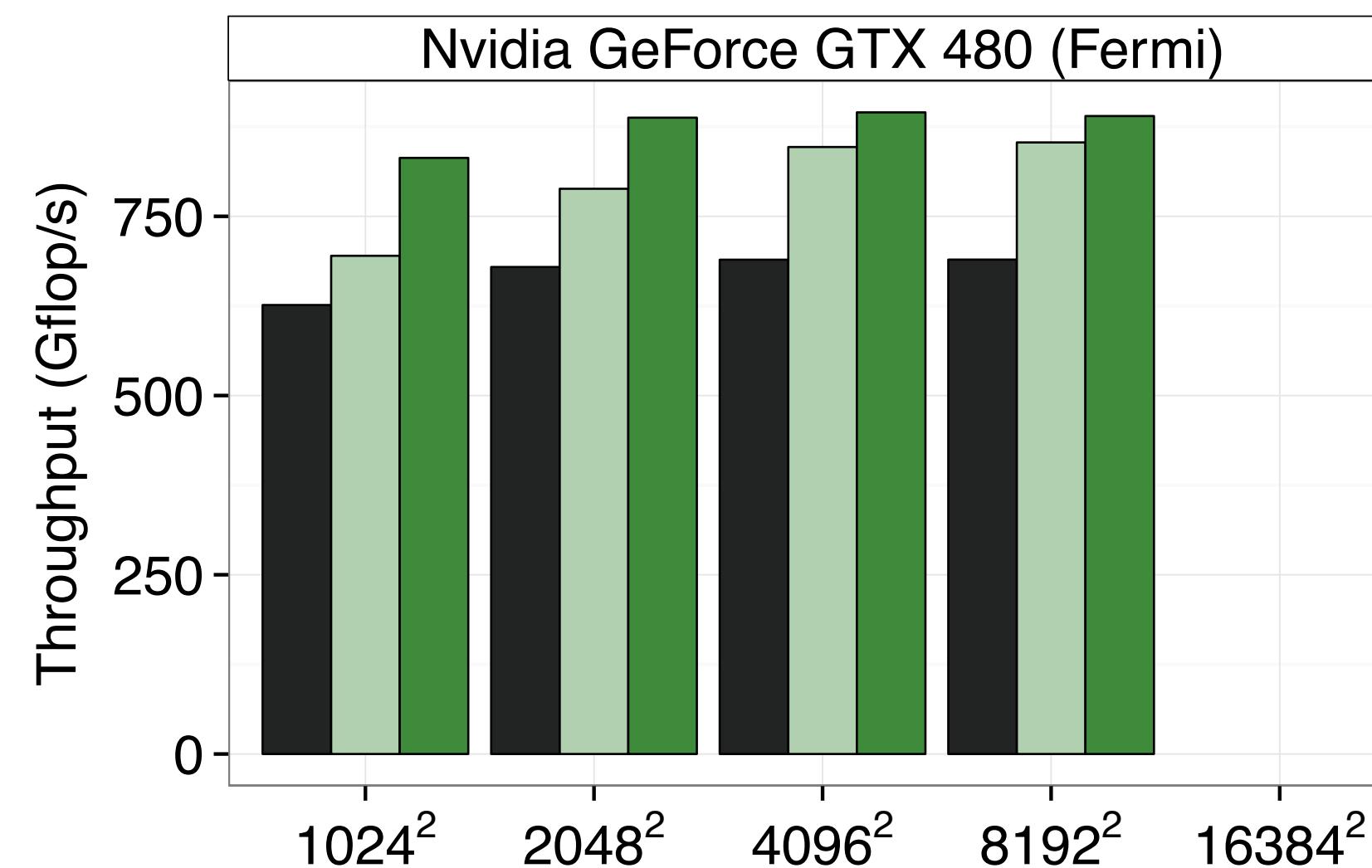
# Performance Evolution for Randomised Search



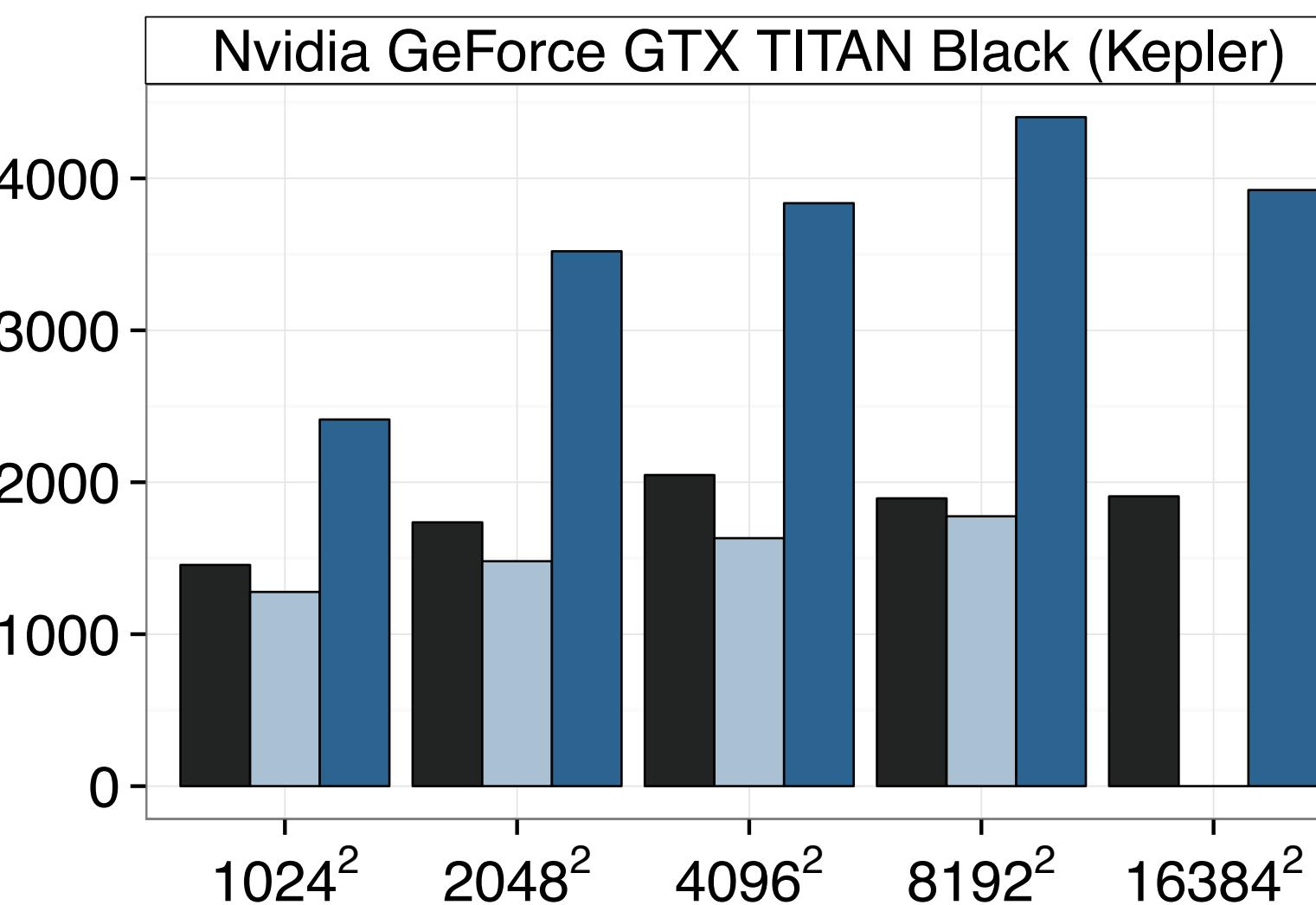
Even with a simple random search strategy one can expect to find  
a good performing kernel quickly

# Performance Results Matrix Multiplication

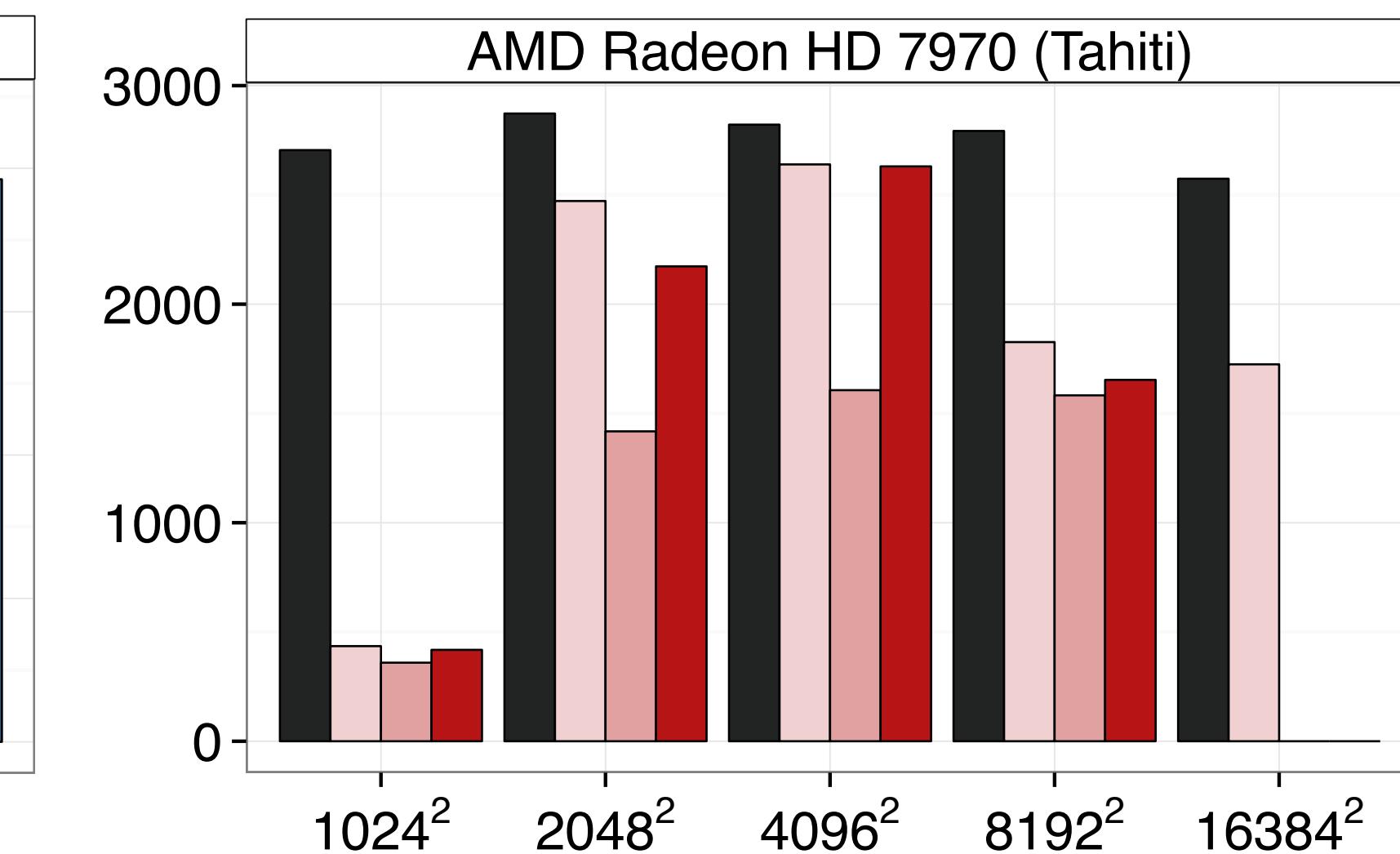
Generated MAGMA cuBLAS



Generated MAGMA cuBLAS

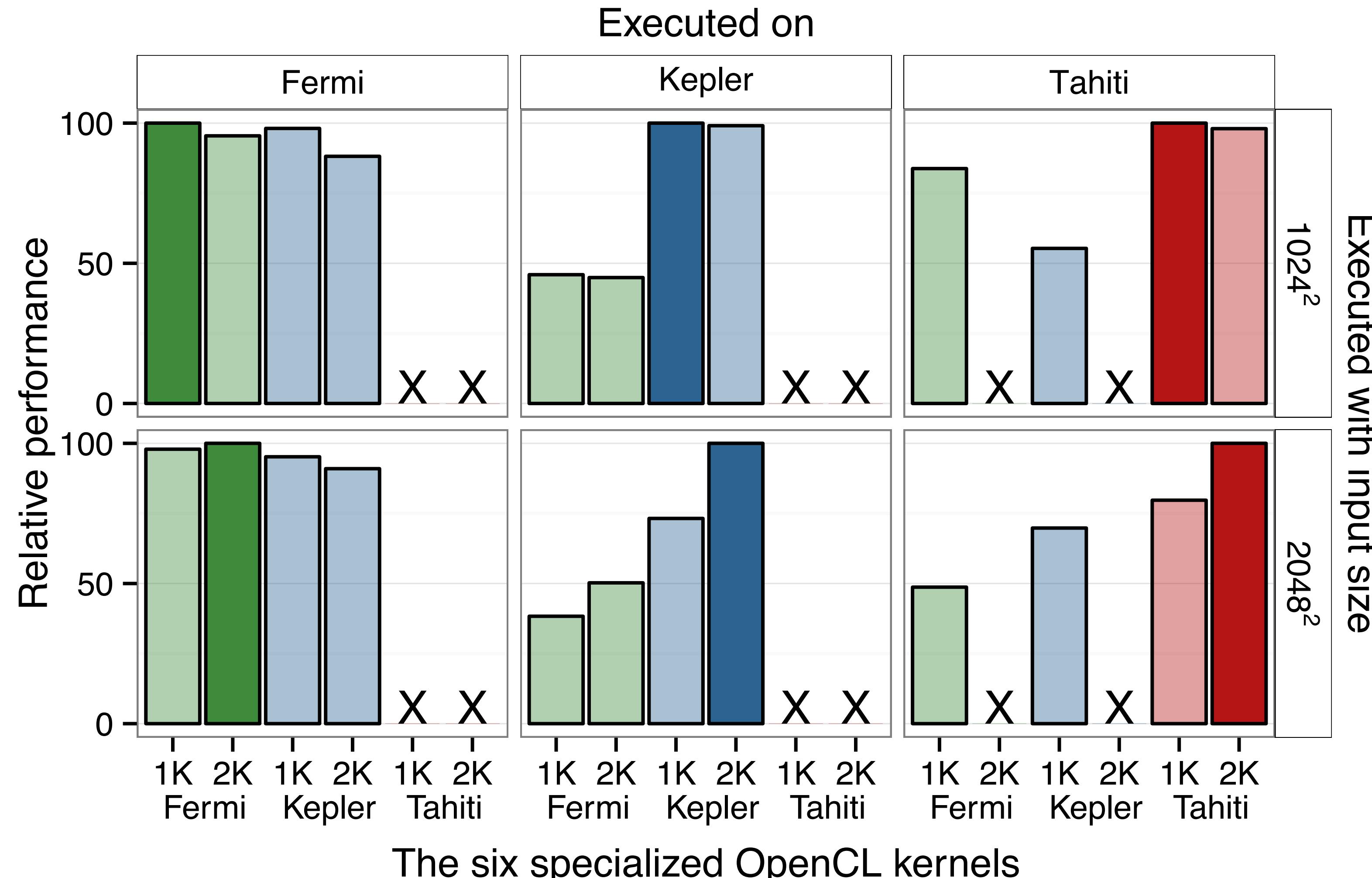


Generated clMAGMA clBLAS clBLAS Tuned



Performance close or better than hand-tuned MAGMA library

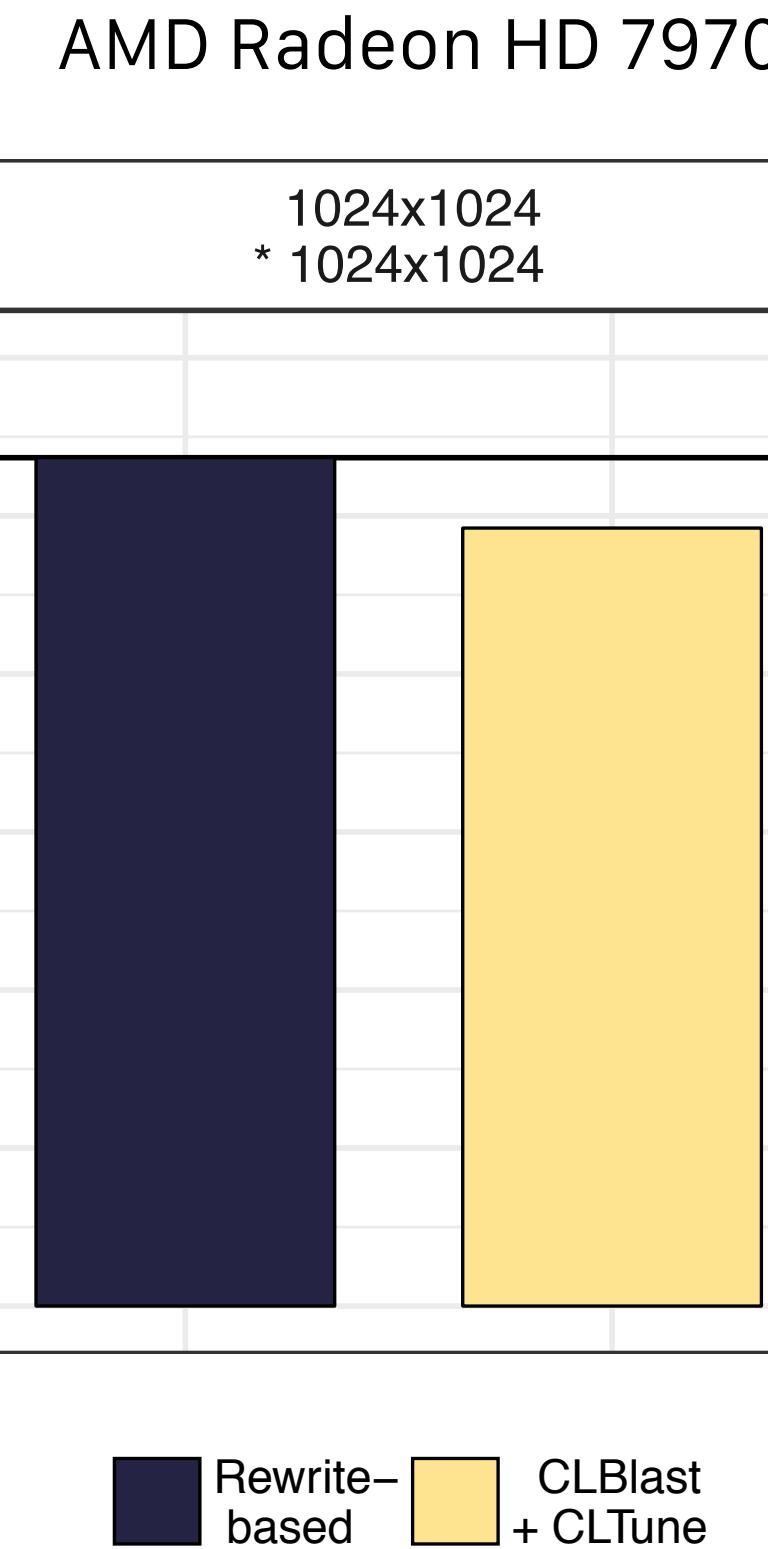
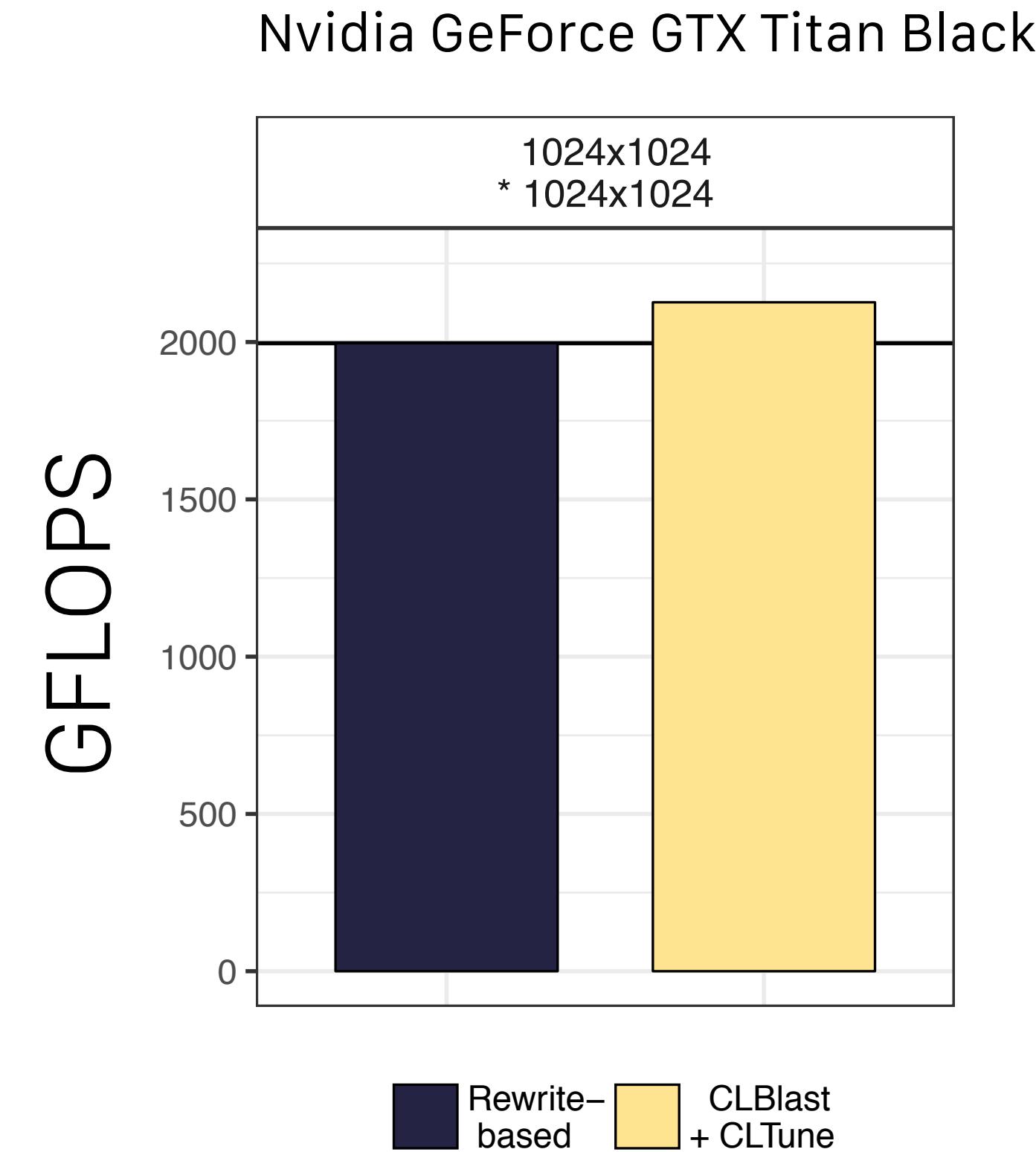
# Performance Portability Matrix Multiplication



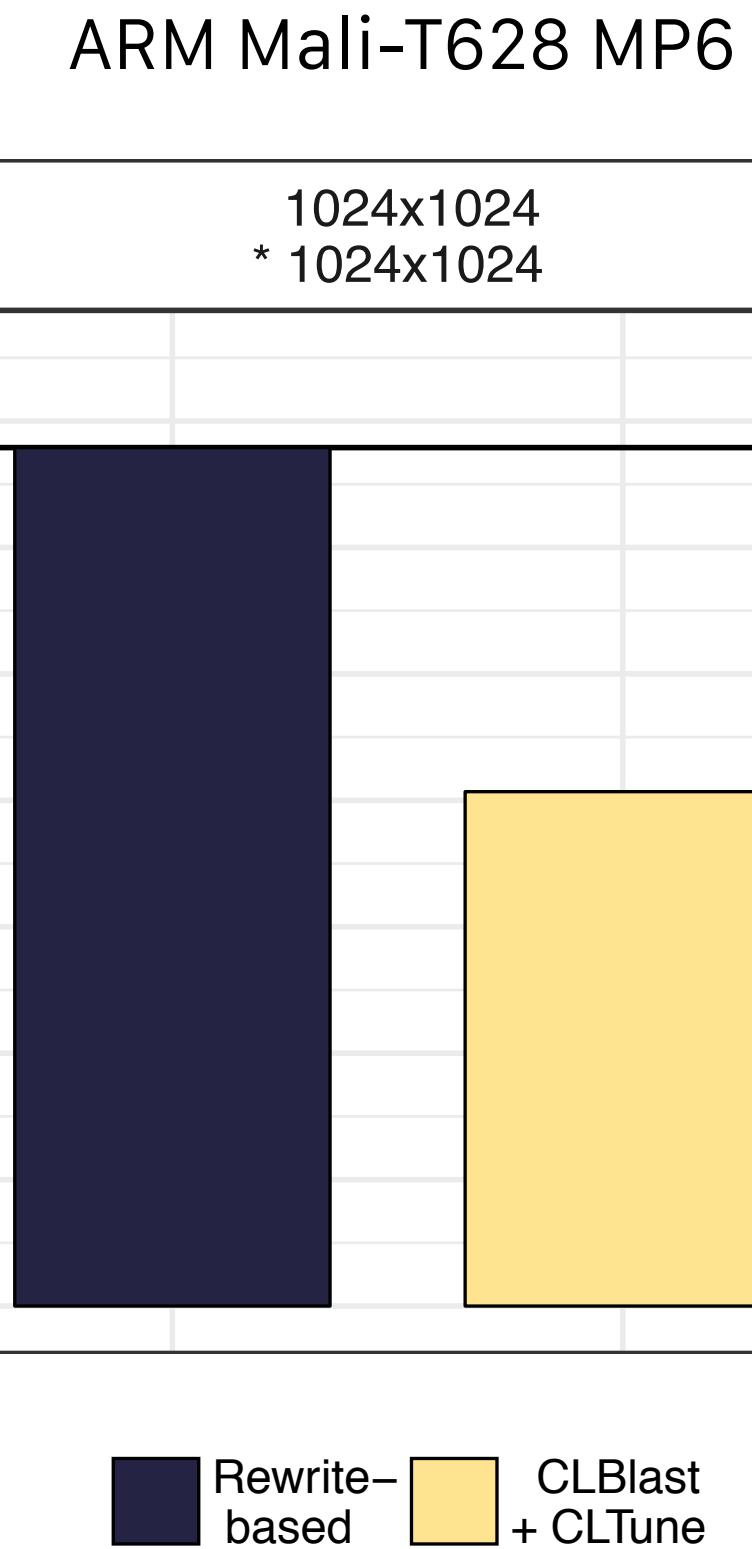
Generated kernels are specialised for device and input size

# Desktop GPUs vs. Mobile GPU

Desktop GPUs

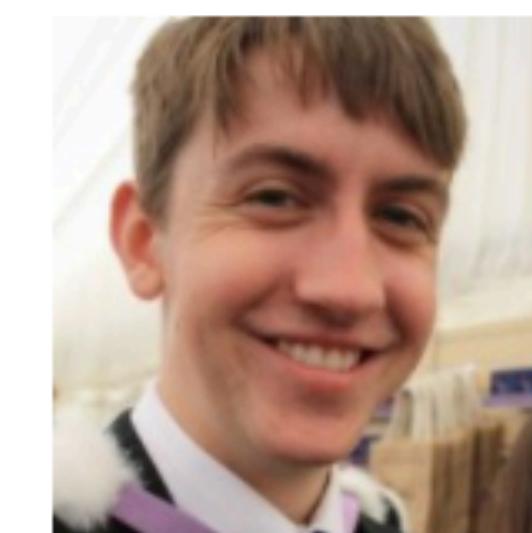
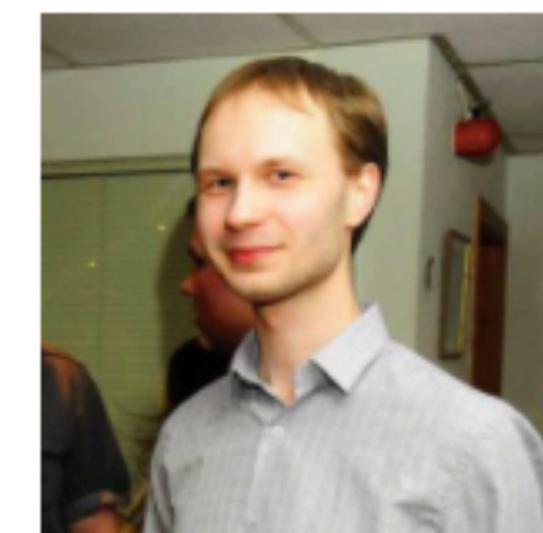


Mobile GPU



Performance portable even for mobile GPU device!

# The LIFT Team



University  
of Glasgow



THE UNIVERSITY  
of EDINBURGH

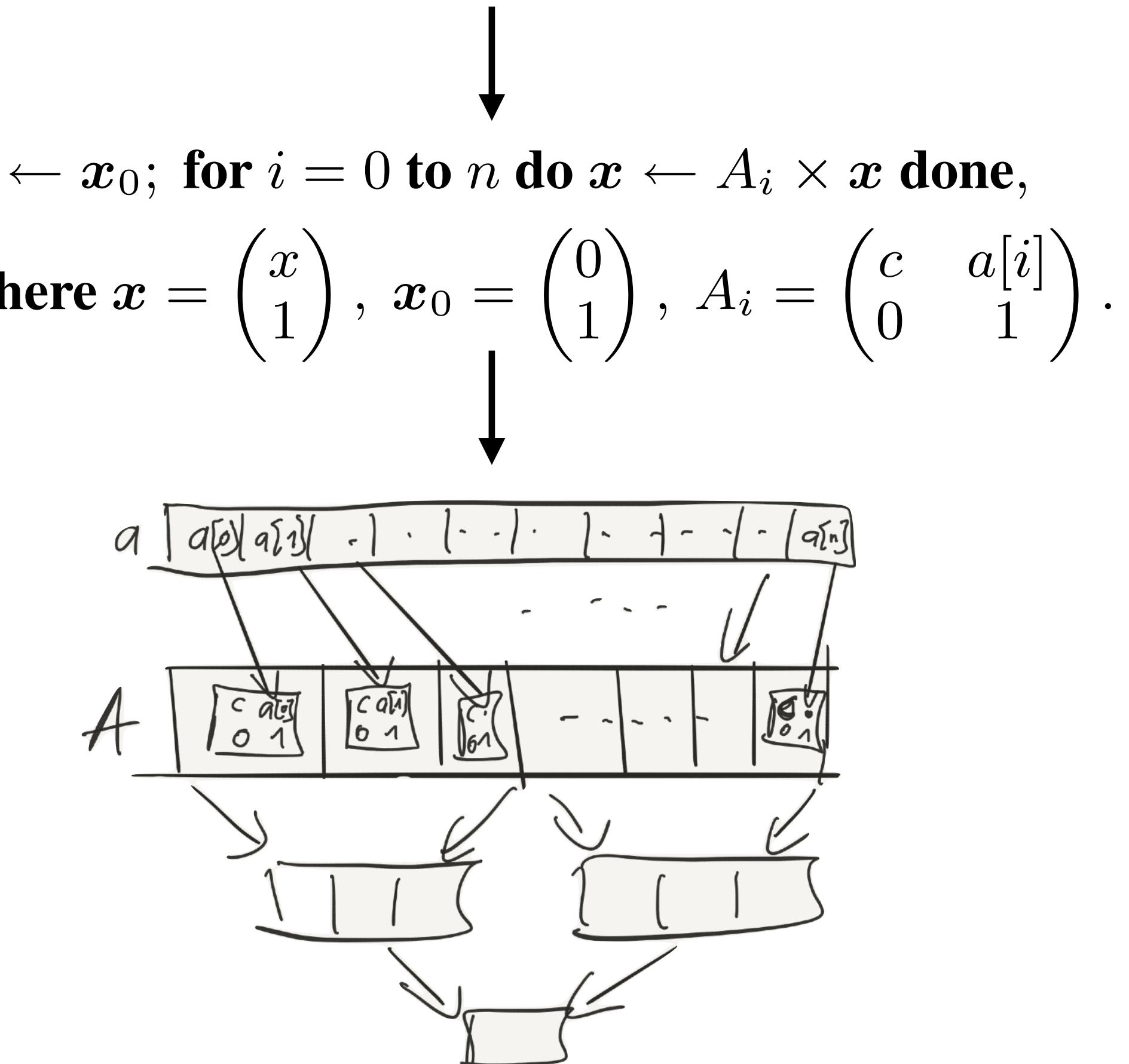
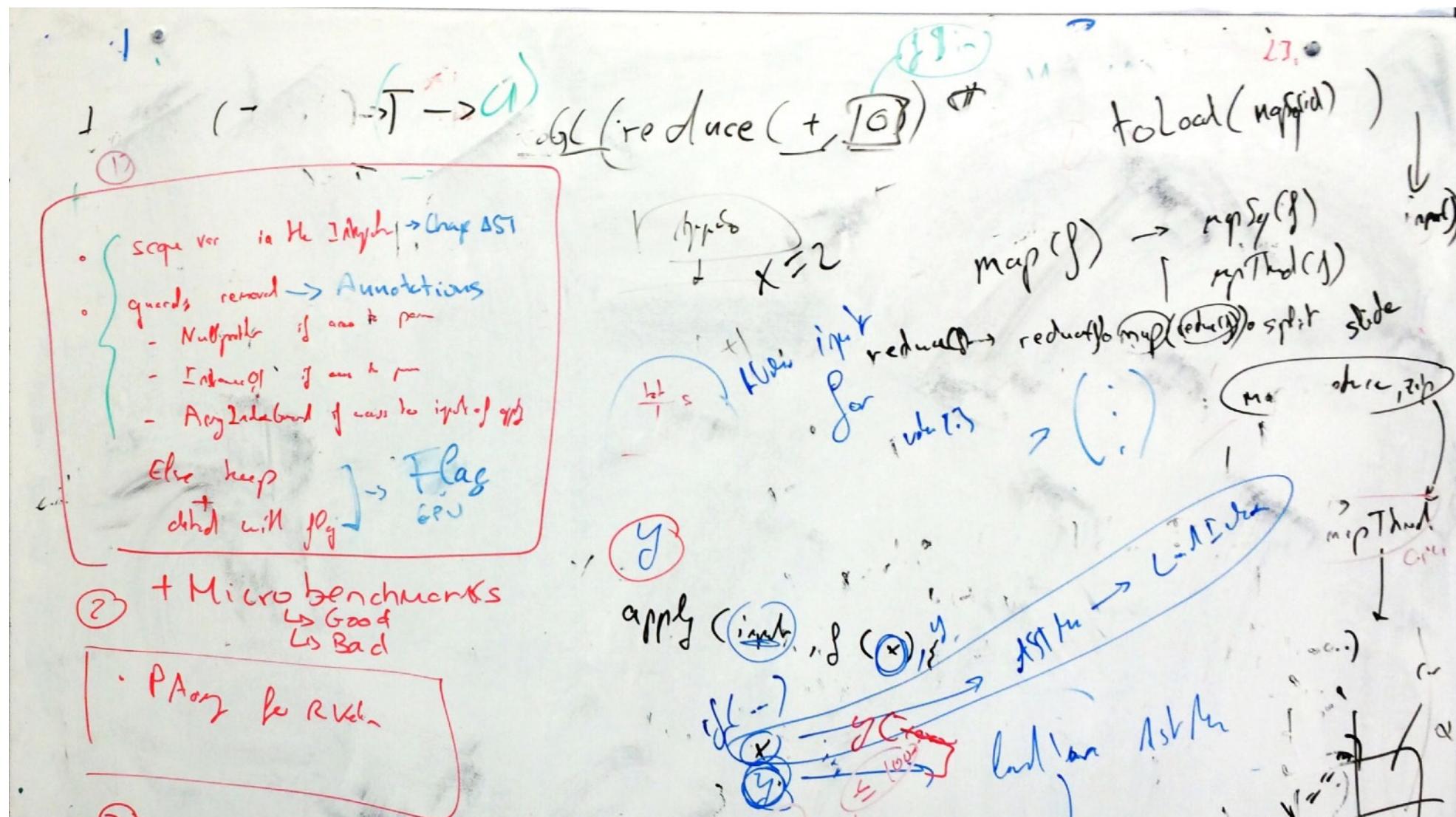


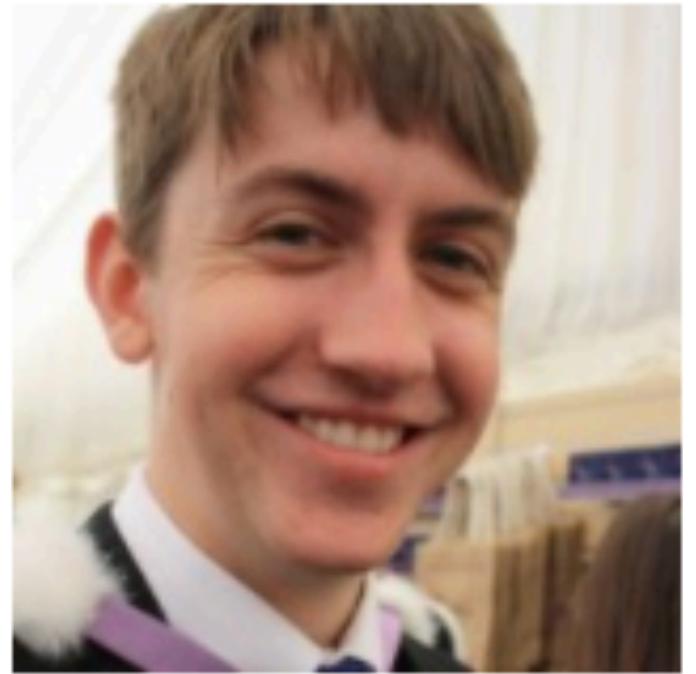


# Identify hidden parallelism in LIFT programs

Federico Pizzuti  
PhD Student  
University of Edinburgh

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





Adam Harries  
PhD Student  
University of Edinburgh

```
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))))
```

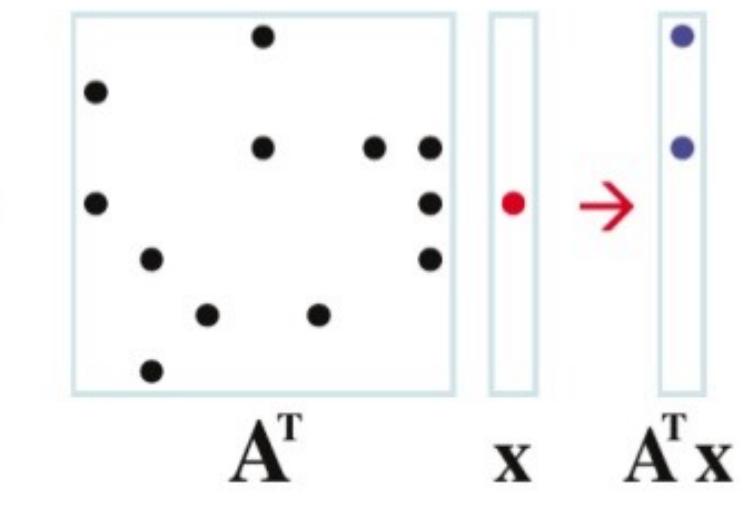
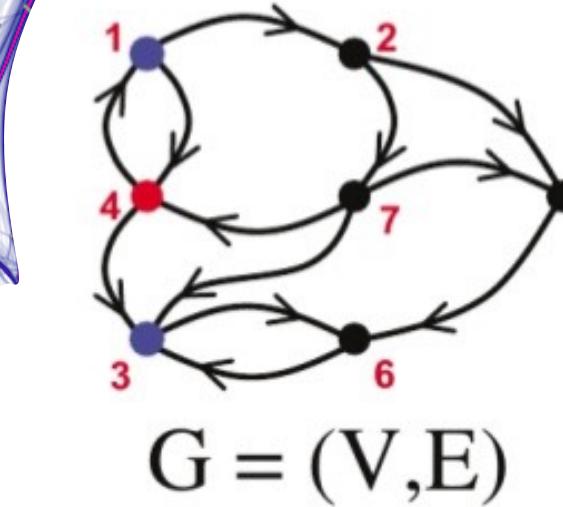
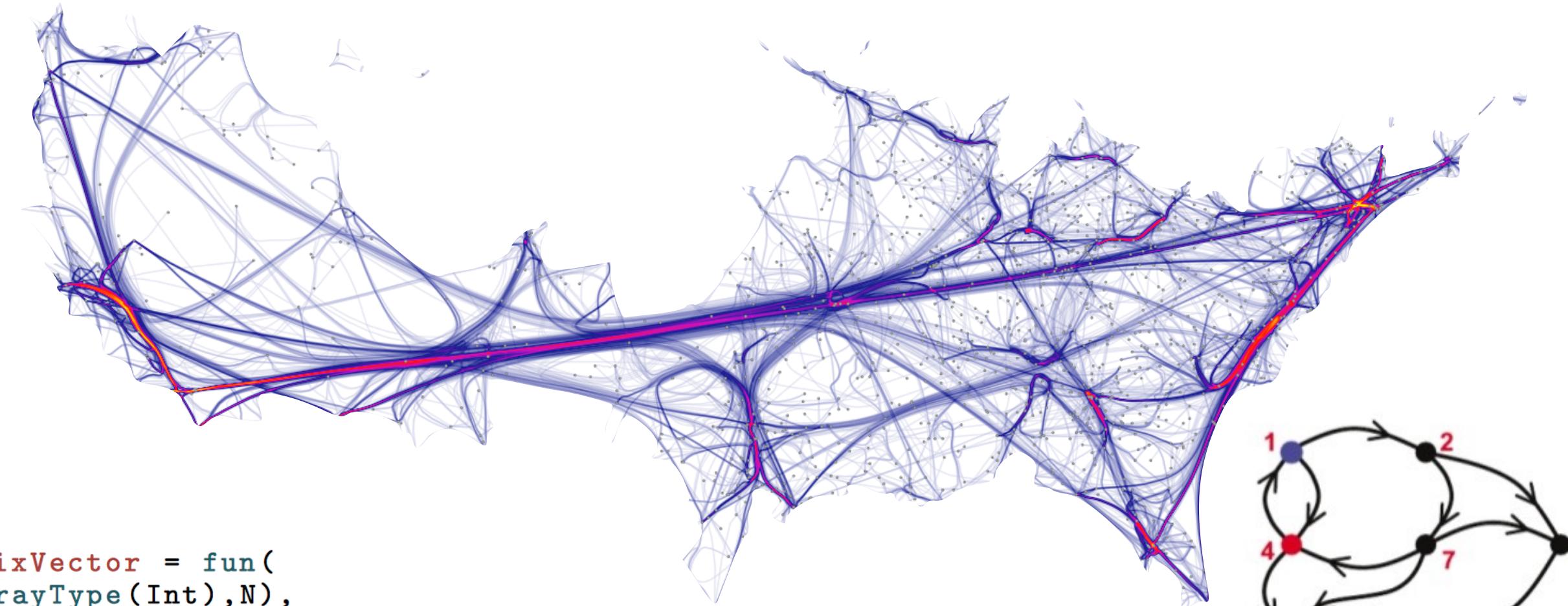
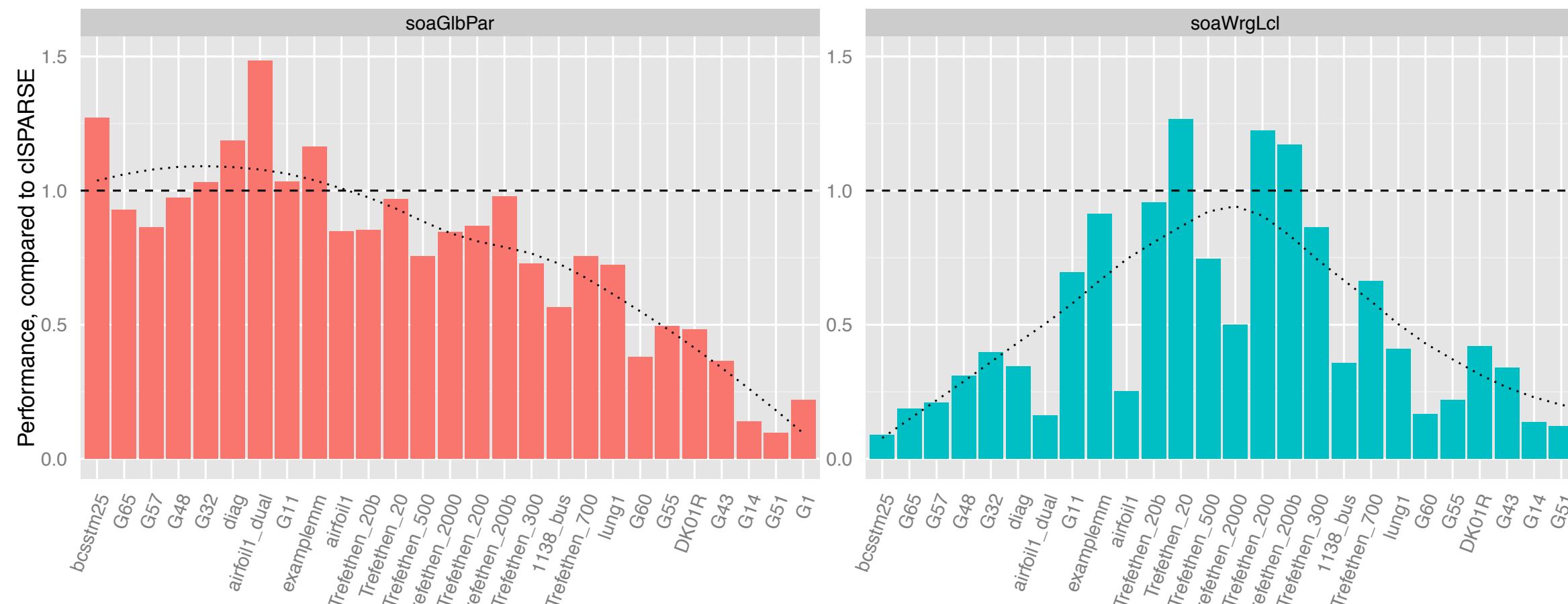
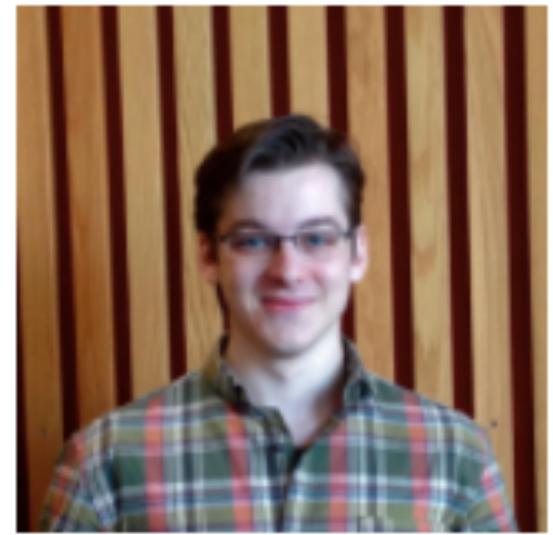


Image credit: [Kepner2011]

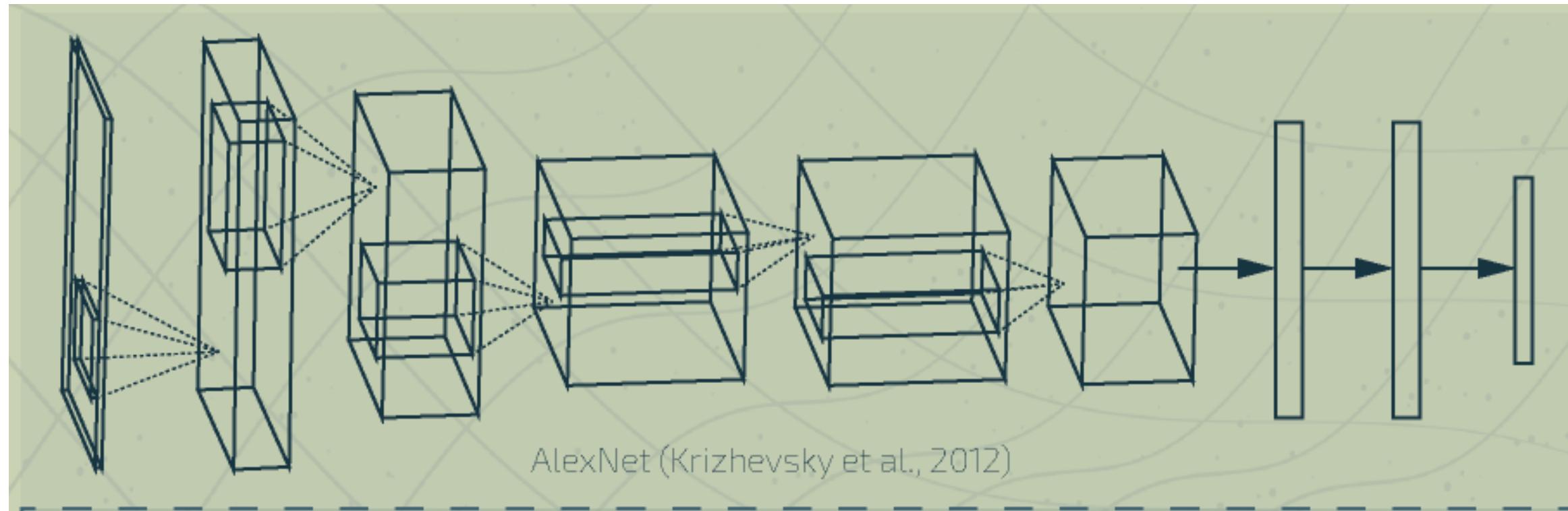


Differently  
optimised kernels  
for different inputs



Naums Mogers  
PhD Student  
University of Edinburgh

# Optimising Deep Learning with LIFT

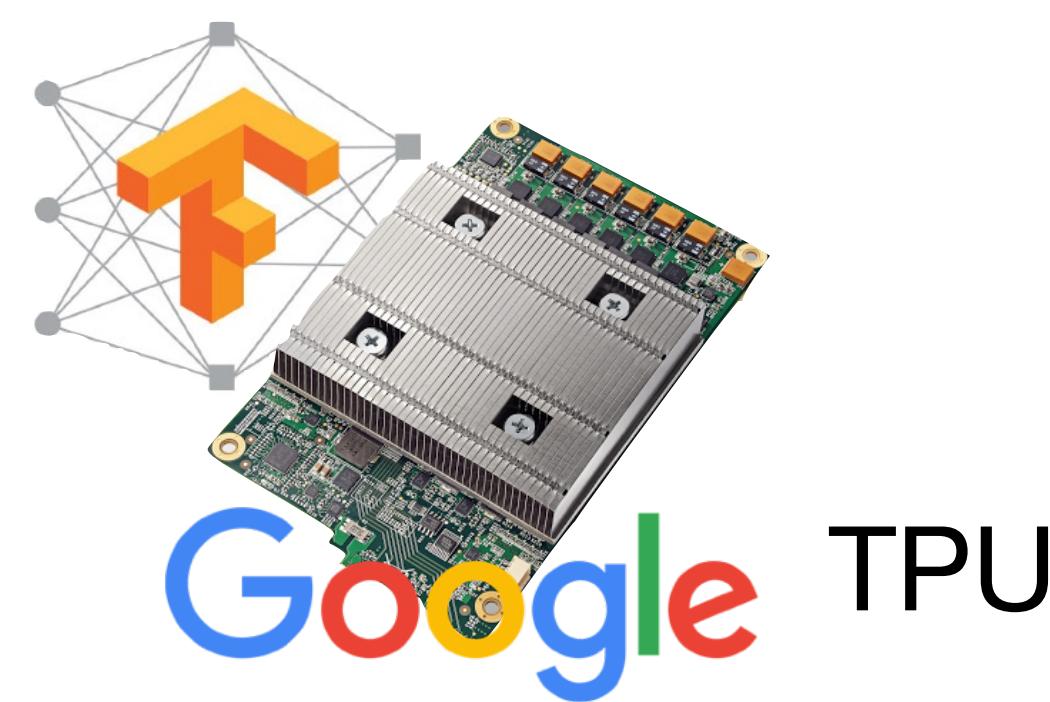


Express layers with LIFT primitives

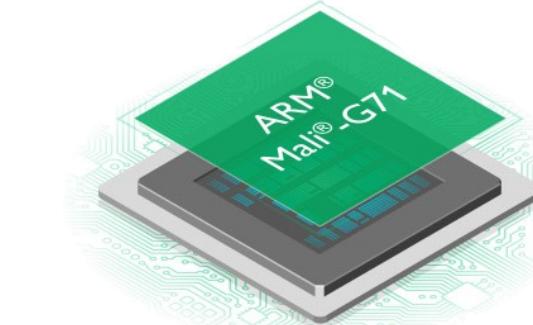
```
fully_connected(f, weights, bias, inputs) :=  
    Map((neuron_weights, neuron_bias) → f() o Reduce(add, neuron_bias) o  
    Map(mult) $ Zip(inputs, neuron_weights)) $ Zip(weights, bias)
```

Optimise individual layers and across layers via rewrites

FPGAs  
Microsoft®  
Research



Low Power Devices





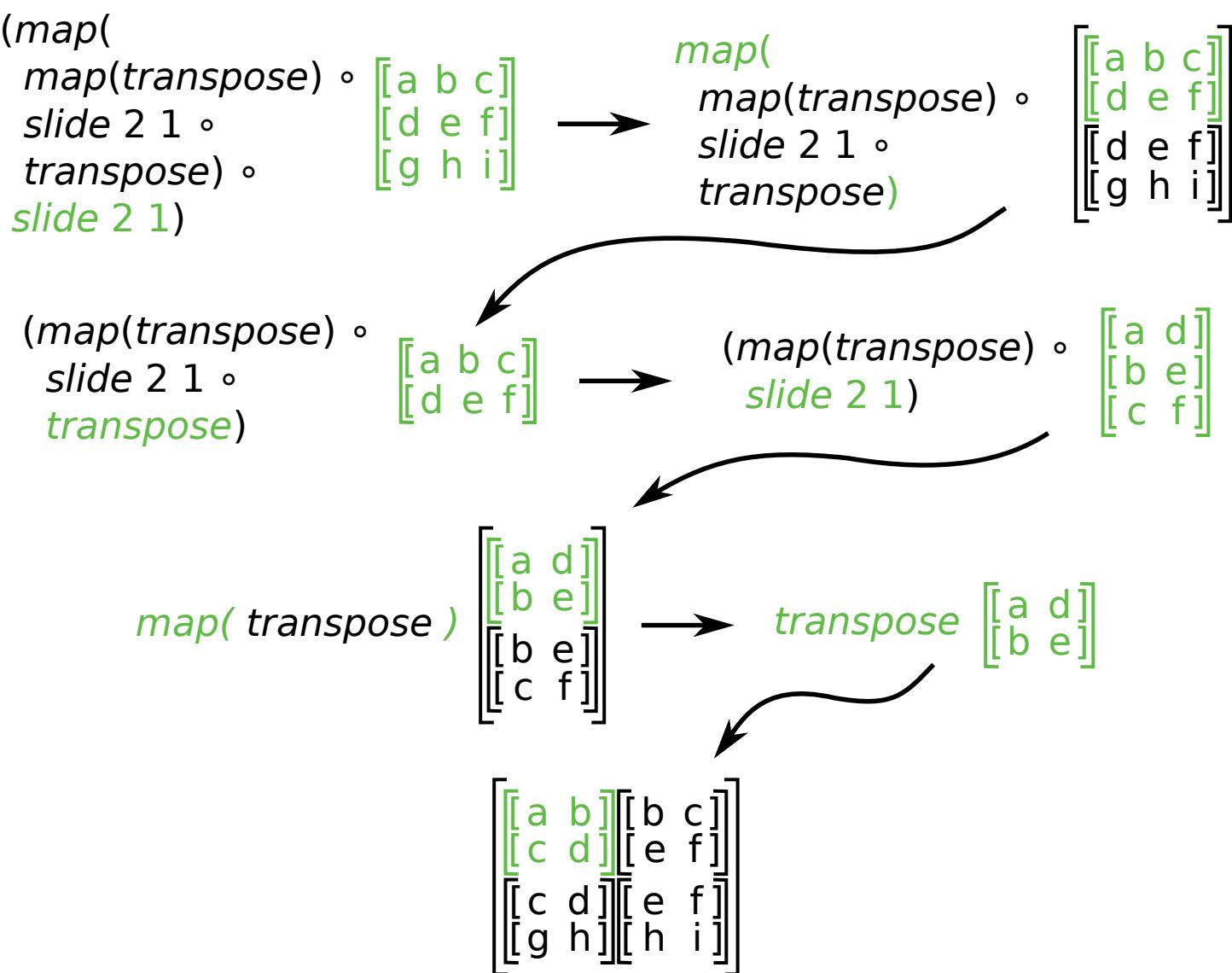
Bastian Hagedorn  
PhD Student  
University of Münster



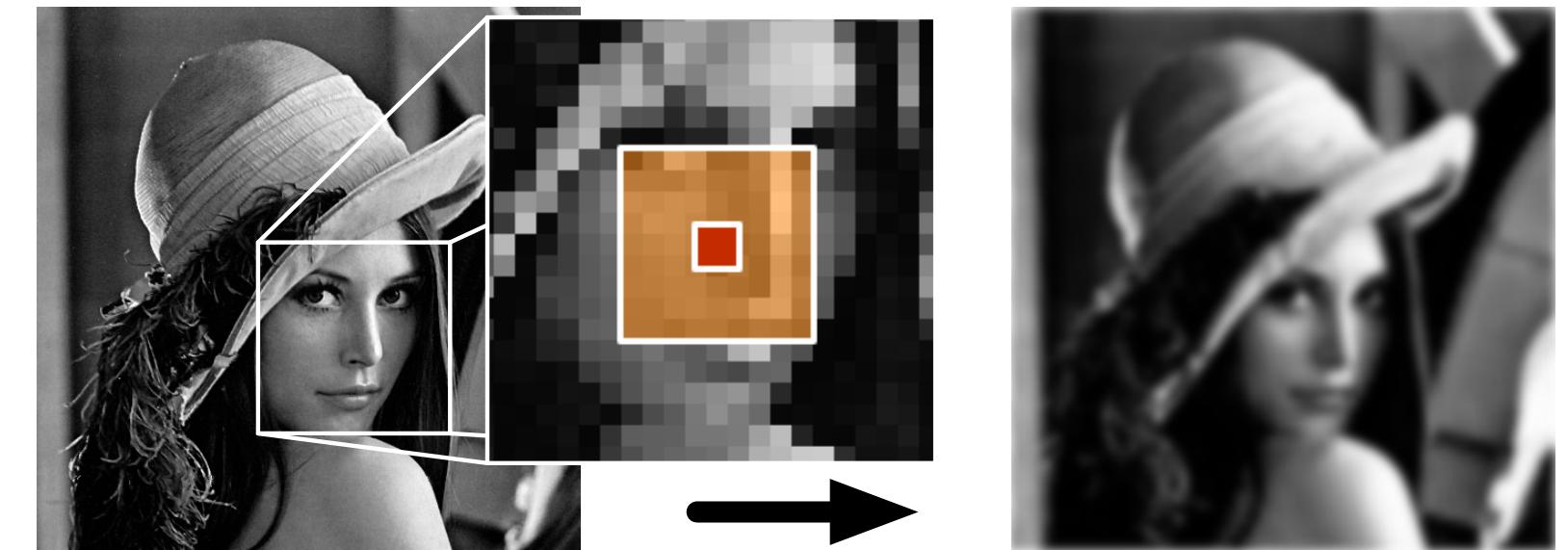
Larisa Stoltzfus  
PhD Student  
University of Edinburgh

# Stencil Computations in LIFT

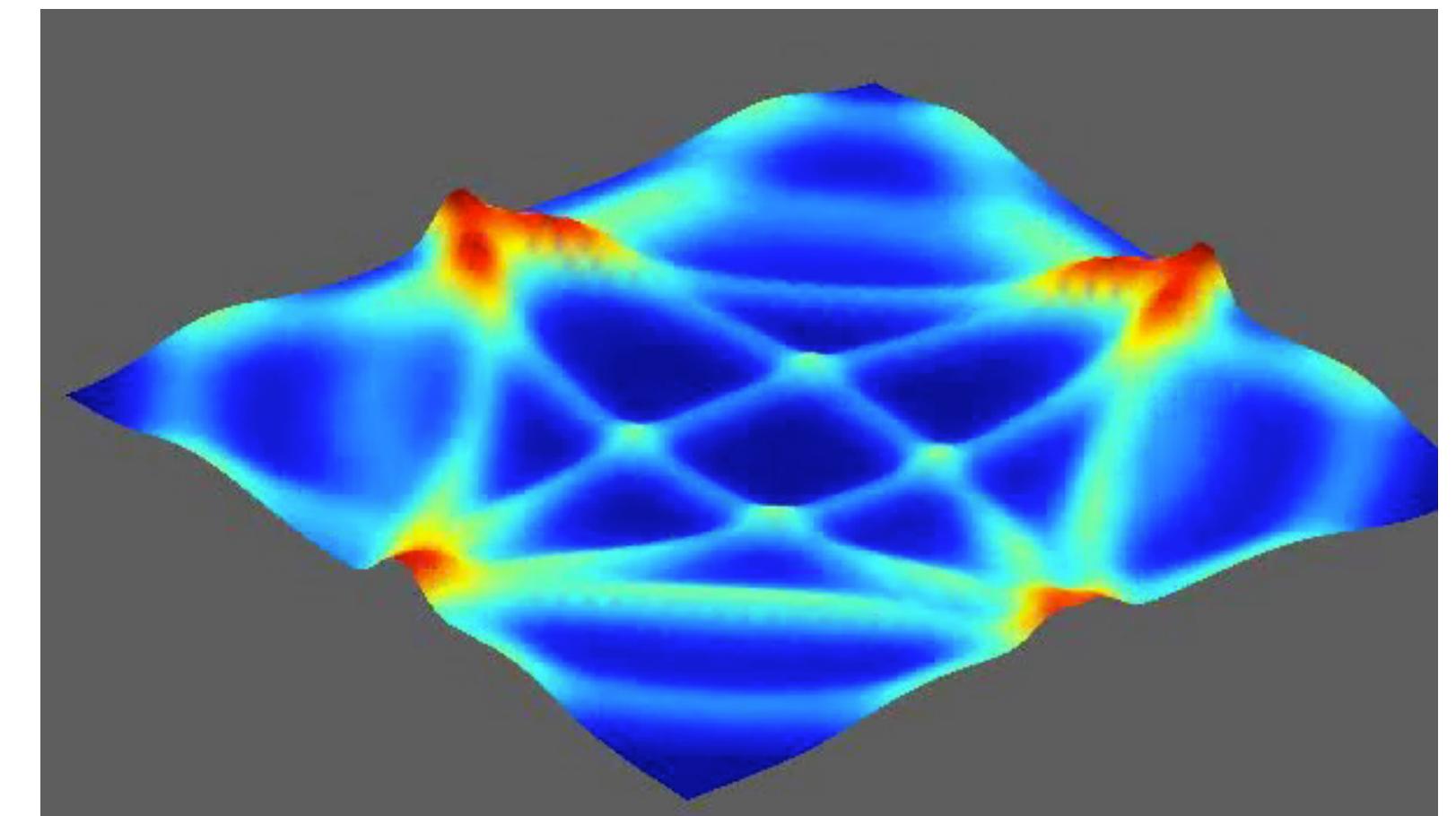
## Express Stencil with Skeletons



## Image Processing

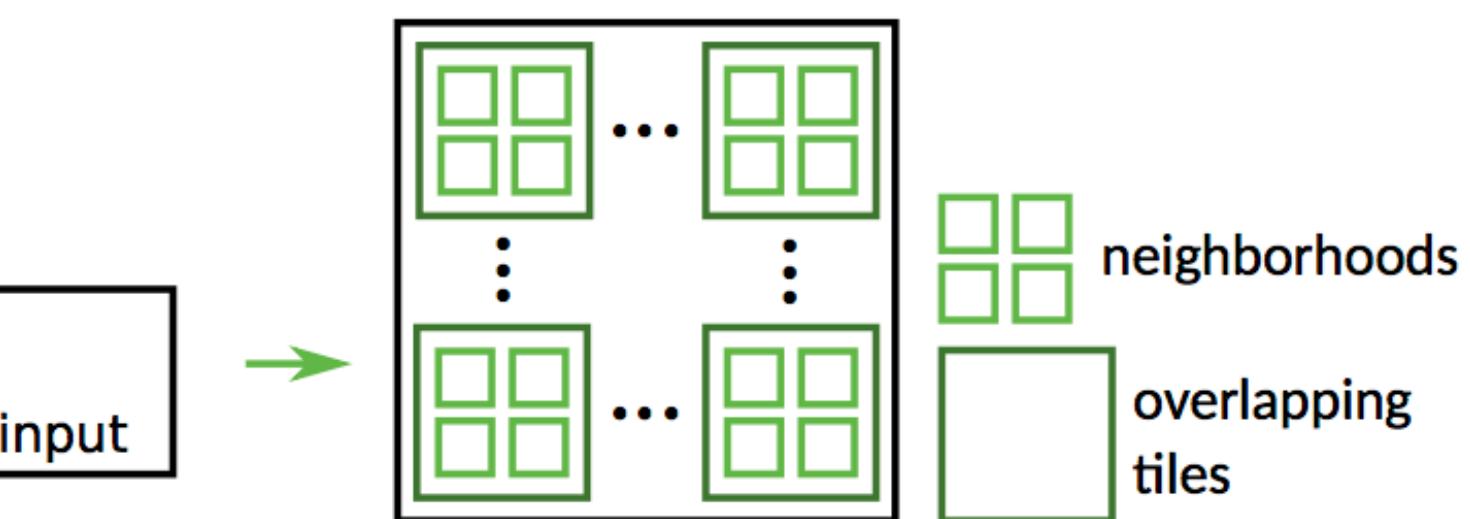


## Acoustics Simulation

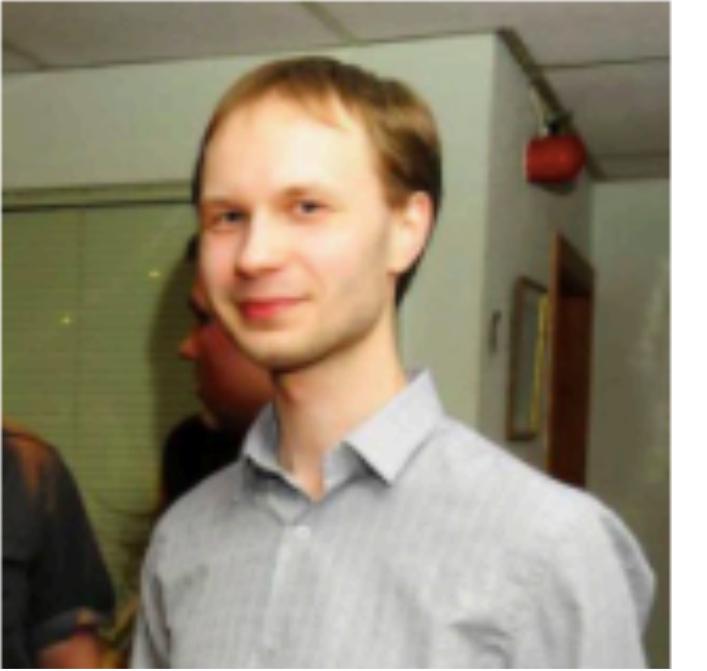


[Video](#)

## Explore optimisations as rewrites



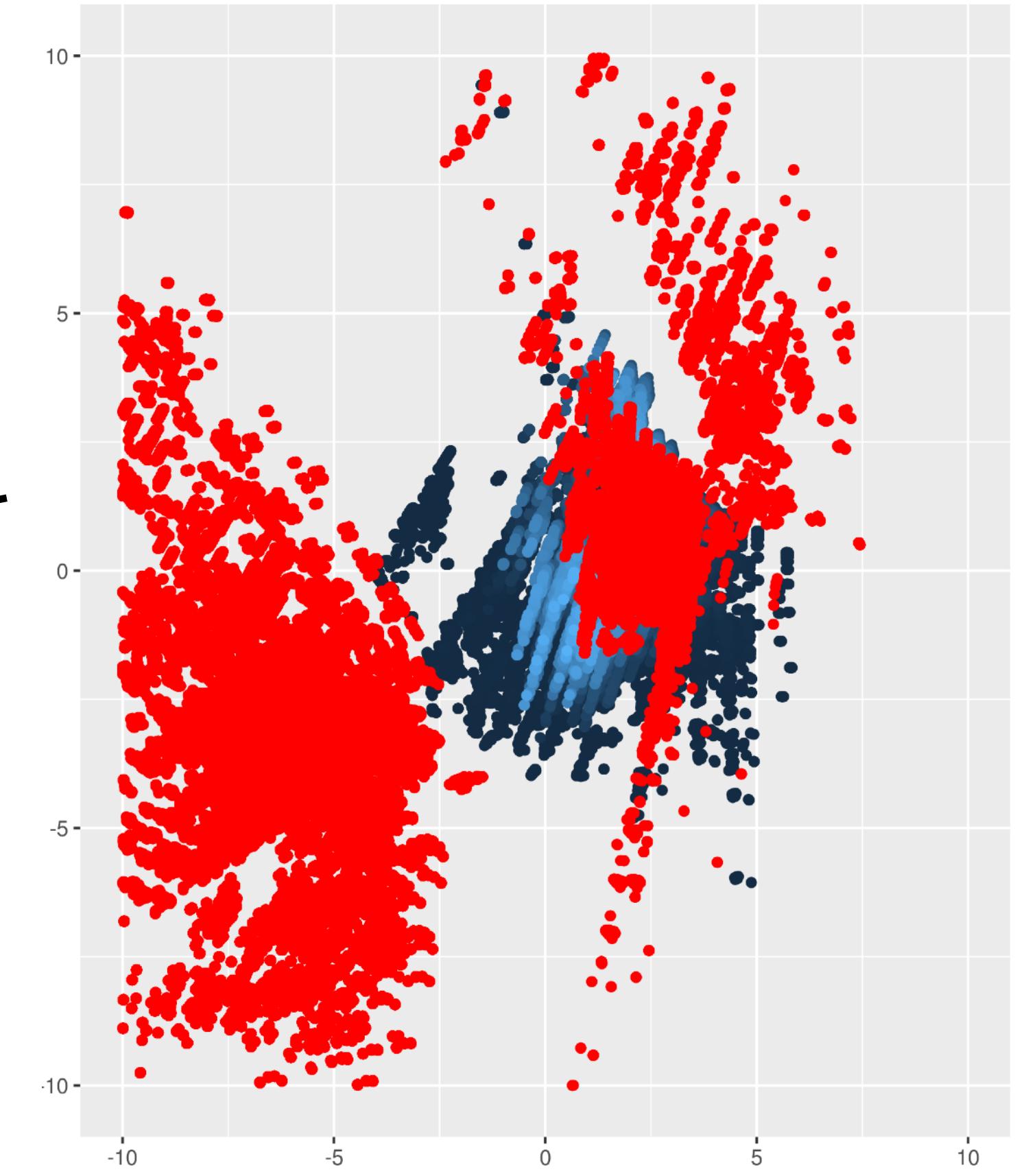
# Performance Modeling of LIFT Programs



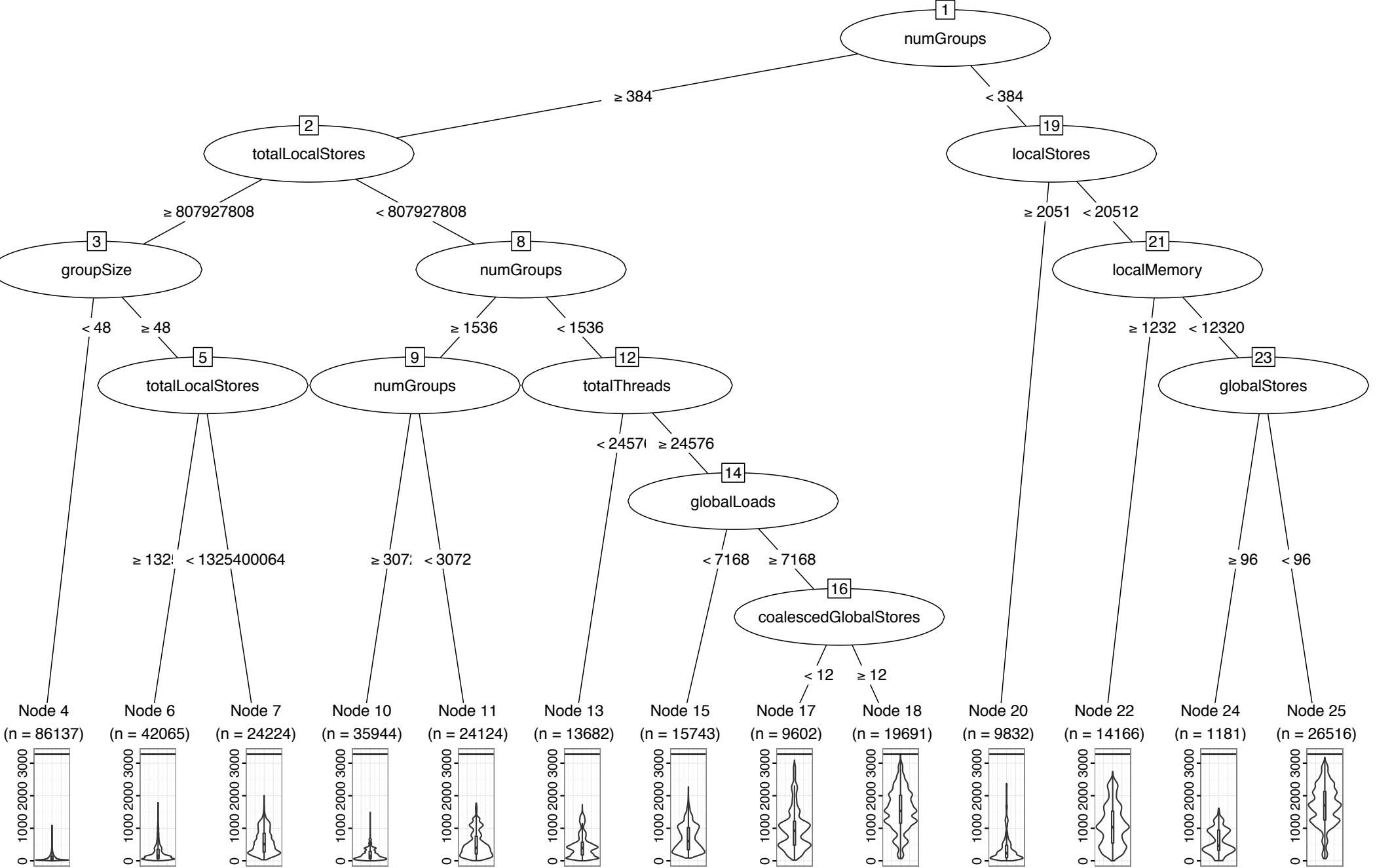
Toomas Remmelt  
PhD Student  
University of Edinburgh

untile  $\circ$  map( $\lambda$  rowOfTilesA .  
map( $\lambda$  colOfTilesB ,  
toGlobal(copy2D))  
reduce( $\lambda$  (tileA, tileB) .  
map(map( $\lambda$  -  $\circ$  zip(tileAcc)  $\circ$   
map( $\lambda$  as .  
map( $\lambda$  bs .  
reduce(+, 0)  $\circ$  map( $\times$ )  $\circ$  zip(as, bs)  
, toLocal(copy2D(tileB)))  
, toLocal(copy2D(tileA)))  
, 0, zip(rowOfTilesA, colOfTilesB))  
)  $\circ$  tile(m, k, transpose))  
)  $\circ$  tile(n, k, A)

Extract Features



Performance Model



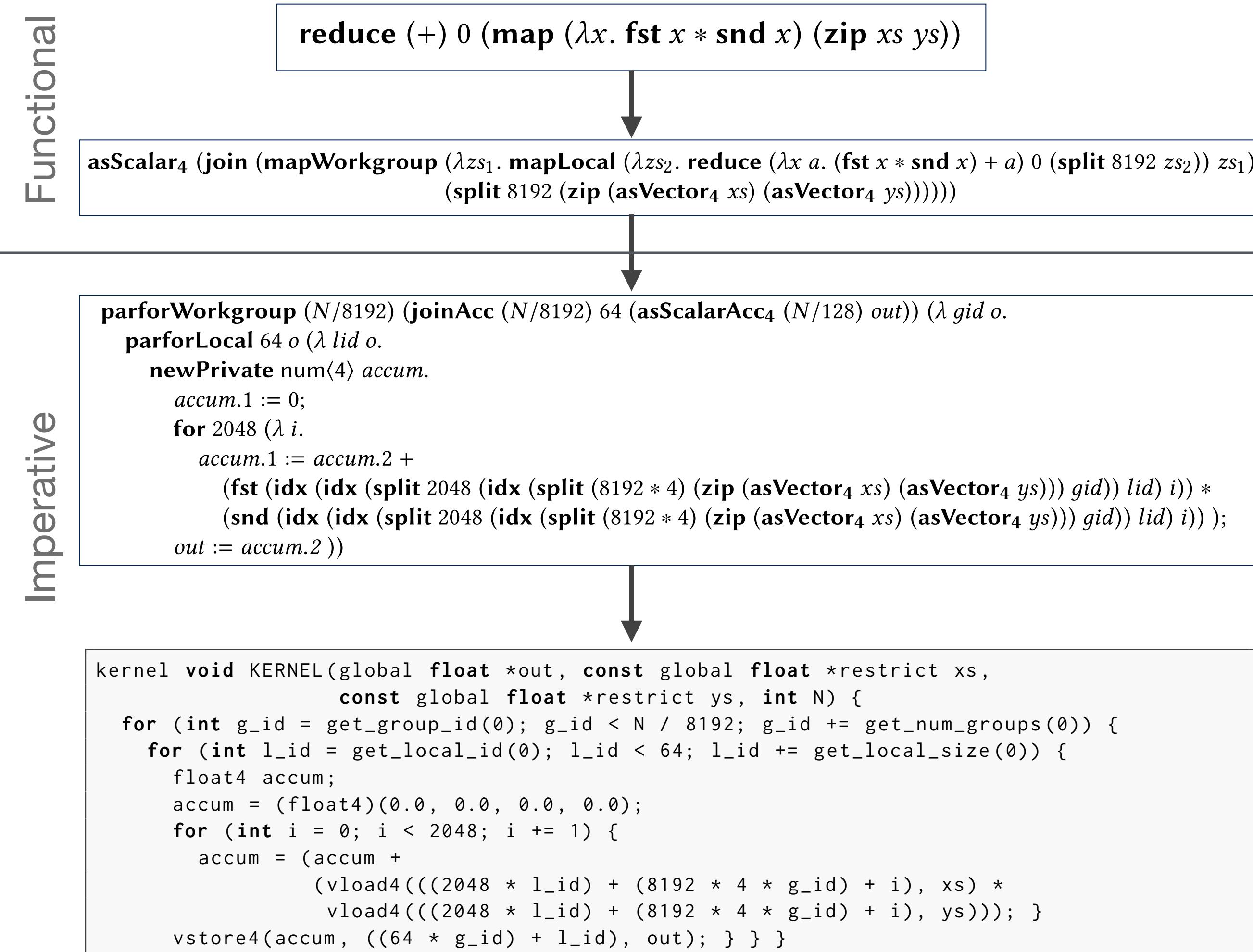
Predictions  
used to drive the  
rewrite process

# 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 ( $\cong$  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>

The screenshot shows the GitHub repository page for 'lift-project/lift'. The repository has 1,923 commits, 1 branch, 0 releases, 10 contributors, and is licensed under MIT. The last commit was made 2 days ago. The repository page includes links for creating new files, uploading files, finding files, and cloning or downloading the code. The GitHub interface features a clean, modern design with a light gray background and white text.

lift-project / lift

1,923 commits 1 branch 0 releases 10 contributors MIT

Branch: master New pull request Create new file Upload files Find file Clone or download

michel-steuwer committed on GitHub Made LICENSE file parsable for github Latest commit 8b13aac 2 days ago

docker Cleaning up the top folder of the repo and restructuring the docker s... 4 months ago

highLevel refactoring 7 months ago

lib Bump ArithExpr 6 days ago

native Add support for querying if the device supports double 1 year ago

presentations Added power point slides of ICFP, PL Interest and PENCIL meeting. 1 year ago

# The LIFT Project

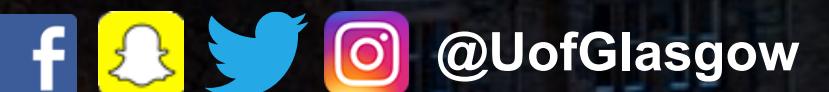
Performance Portable Parallel Code Generation via Rewrite Rules

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

[www.lift-project.org](http://www.lift-project.org)    @LIFTlang

**INSPIRING  
PEOPLE**

#UofGWorldChangers



@UofGlasgow