The *Lift* Project: Performance Portable GPU Code Generation via Rewrite Rules

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

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

THE UNIVERSITY of EDINBURGH
informatics

icsa

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

Prologue

To achieve performance portability
we *need* high-level abstraction!

Traditional imperative programming
approaches *always* lead to non-portable code

Traditional compiler & runtimes have no freedom
to explore alternative implementations

Lessons from the past

1968

A Case against the GO TO Statement.

by Edsger W.Dijkstra
Technological University
Eindhoven, The Netherlands

Since a number of years I am familiar with the obser
quality of programmers is a decreasing function of the de
statements in the programs they produce. Later I discover
the go to statement has such disastrous effects and did I
that the go to statement should be abolished from all "hi
programming languages (i.e. everything except -perhaps- p
At that time I did not attach too much importance to this
submit my considerations for publication because in very
in which the subject turned up, I have been urged to do s

My first remark is that, although the programmer's a
he has constructed a correct program, the process taking

1972

STRUCTURED PROGRAMMING

O.-J. DAHL, E. W. DIJKSTRA
and C. A. R. HOARE

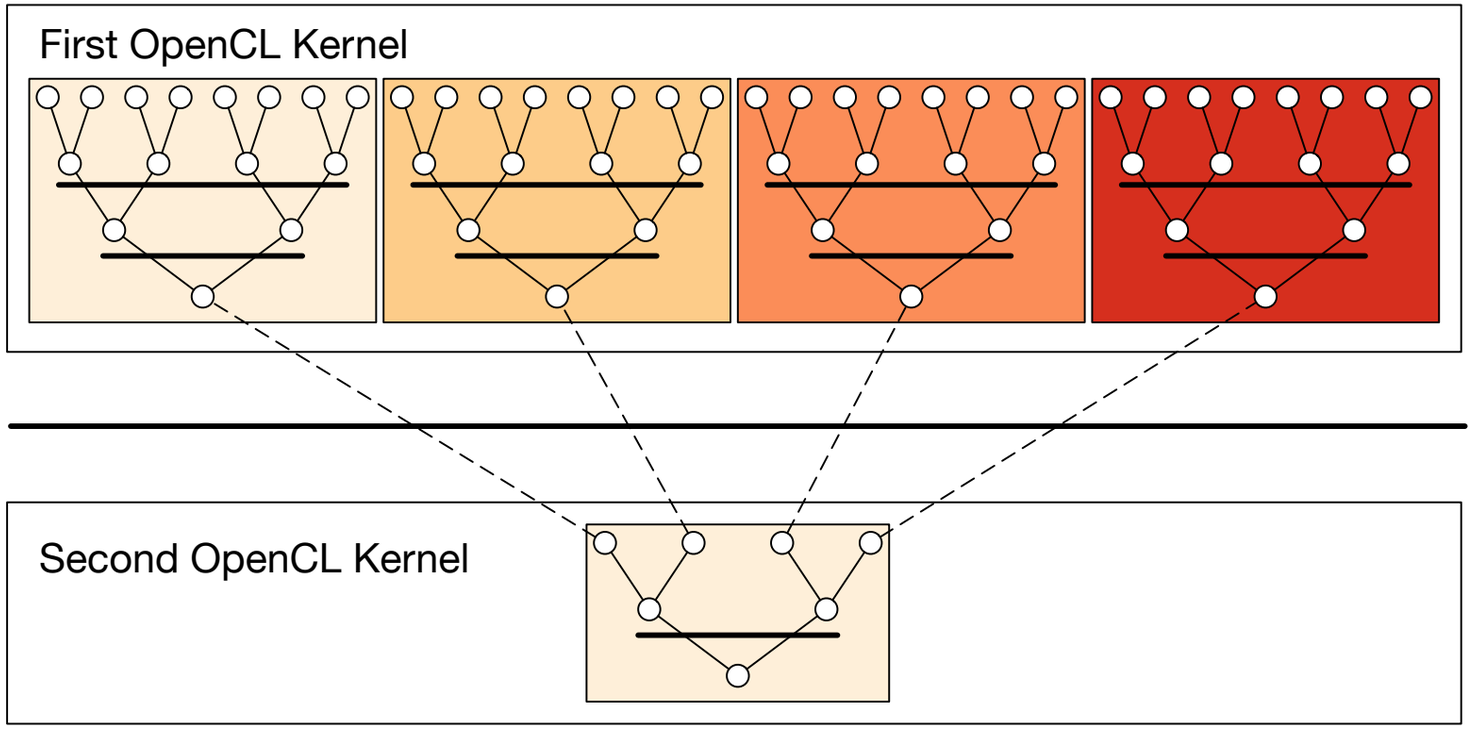
“High-level” abstractions like `if` and `for` have carried us through the sequential age of computing

We need appropriate high-level abstractions for the parallel and concurrent age of computing

End of Prologue

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



Programming with OpenCL

- Case Study: Parallel reduction in 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];
}
```

Programming with OpenCL

- Case Study: Parallel reduction in 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**

Programming with OpenCL

- Case Study: Parallel reduction in 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];
}
```

Programming with OpenCL

- Case Study: Parallel reduction in 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

Programming with OpenCL

- Case Study: Parallel reduction in 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];
}
```

Functionally correct implementations in OpenCL are hard!

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];
}
```

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];
}
```

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];
}
```

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];
}
```

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]; }
```

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]; }
```

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]; }
```

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];
}
```

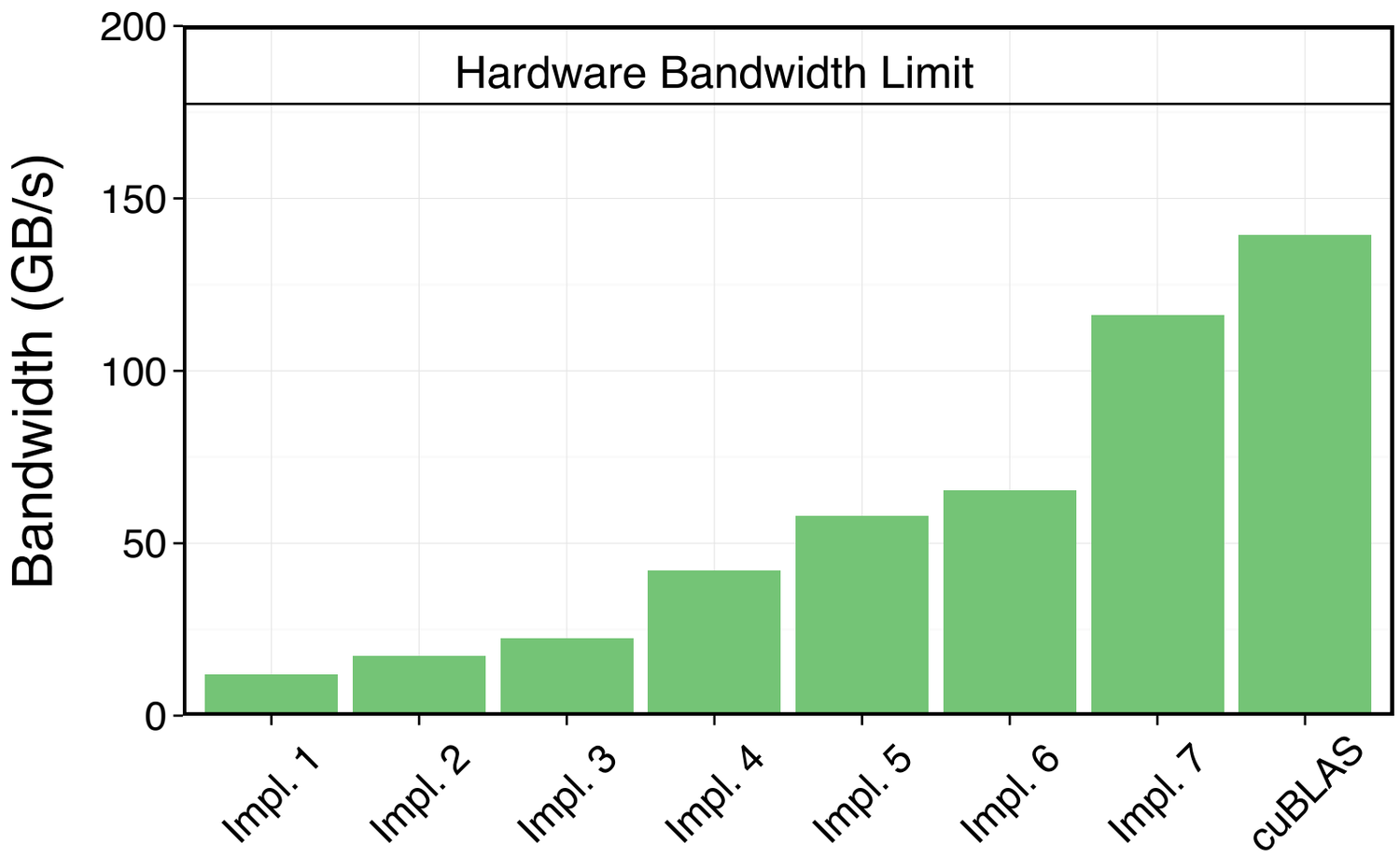
Unoptimized Implementation

```
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

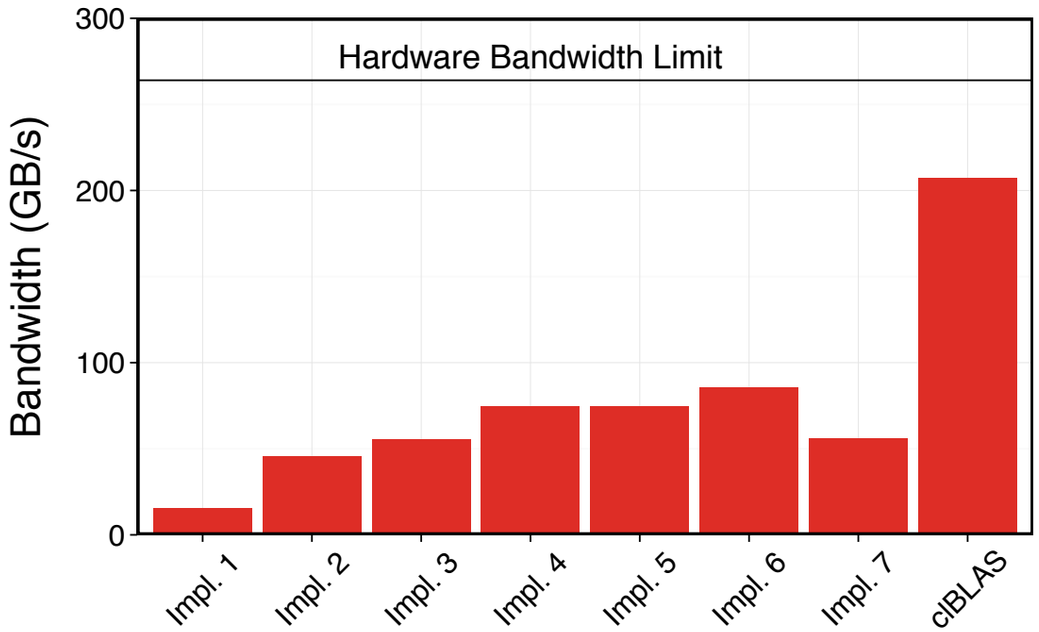
Performance Results Nvidia



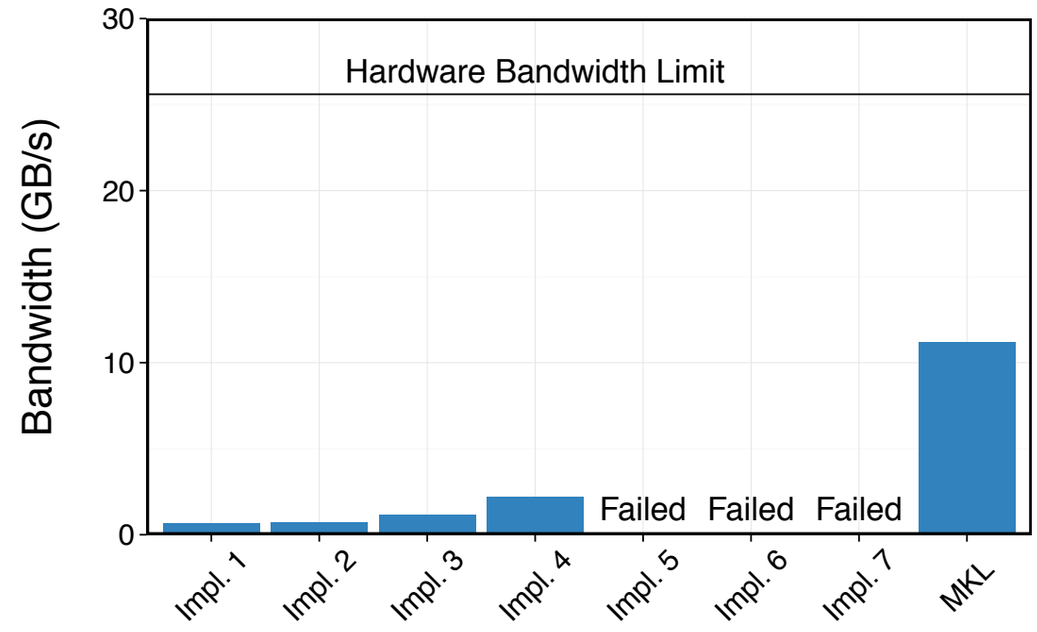
(a) Nvidia's GTX 480 GPU.

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

Performance Results AMD and Intel



(b) AMD's HD 7970 GPU.

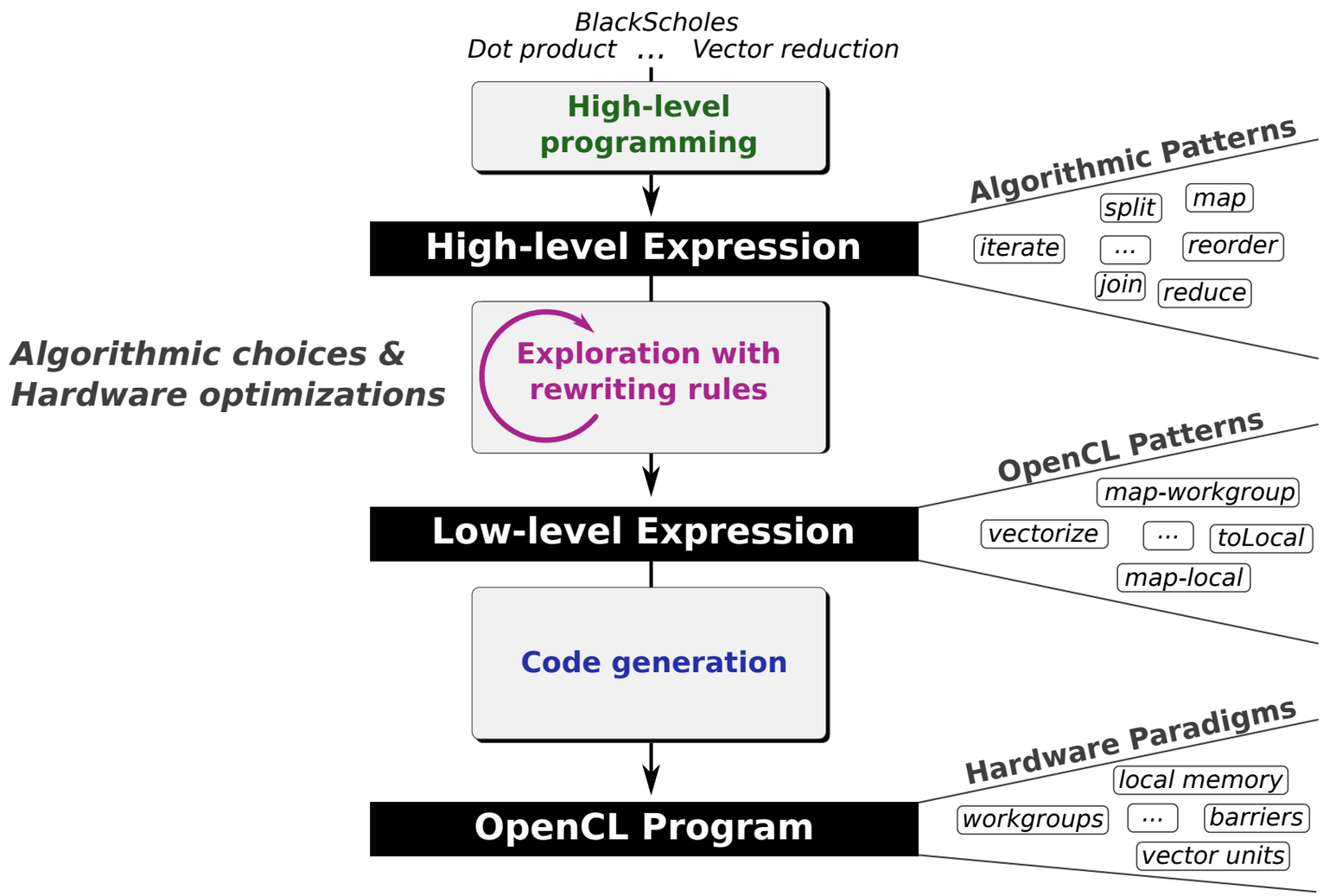


(c) Intel's E5530 dual-socket CPU.

- ... 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

Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach "Generating Performance Portable Code using Rewrite Rules: From High-Level Functional Expressions to High-Performance OpenCL Code" in ICFP 2015.

Walkthrough

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

rewrite rules

code generation

②

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

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];
}
```

Walkthrough

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

rewrite rules

code generation

②

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

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];
}
```

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

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

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

rewrite rules

code generation

②

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

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];
}
```

Walkthrough

① $sum(vec) = reduce(+, 0, vec)$

rewrite rules

code generation

②

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

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];
}
```

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:

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

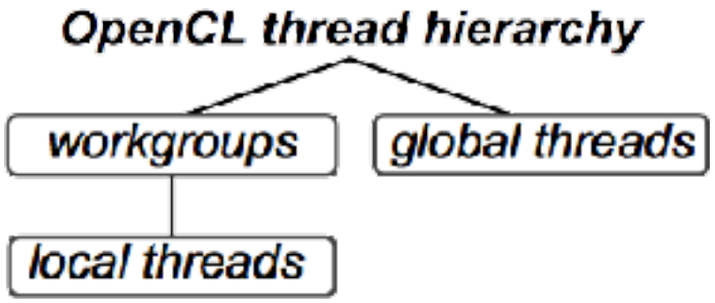
$$reducePart\ f\ z \rightarrow reducePart\ f\ z \circ reorder$$

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

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

② OpenCL Primitives

Primitive	OpenCL concept
$mapGlobal$	Work-items
$mapWorkgroup$ $mapLocal$	Work-groups
$mapSeq$ $reduceSeq$	Sequential implementations
$toLocal$, $toGlobal$	Memory areas
$mapVec$, $splitVec$, $joinVec$	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 $\mathrm{vecSum} = \mathit{reduce}\ (+)\ 0$

rewrite rules

code generation

②

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

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];
}
```

Walkthrough

① $vecSum = reduce\ (+)\ 0$

rewrite rules

code generation

②

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

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];
}
```

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 $\mathrm{vecSum} = \mathit{reduce}\ (+)\ 0$

rewrite rules

code generation

②

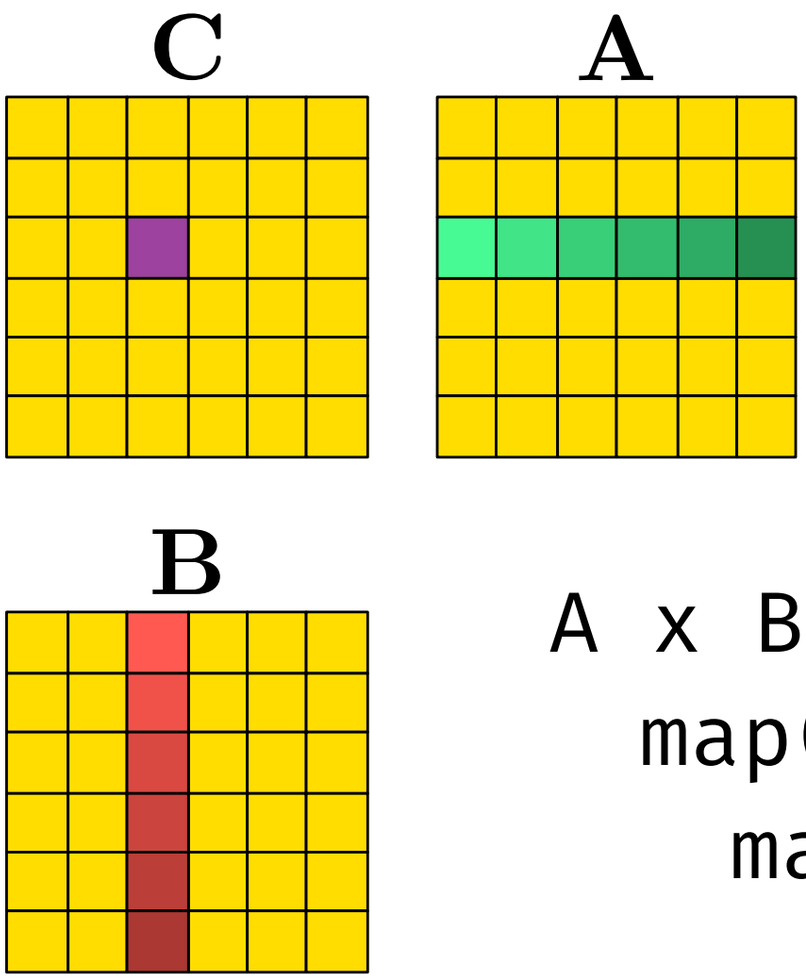
$$
\begin{aligned}
&\mathrm{vecSum} = \mathit{reduce} \circ \mathit{join} \circ \mathit{map\text{-}workgroup}\ (\\
&\quad \mathit{join} \circ \mathit{toGlobal}\ (\mathit{map\text{-}local}\ (\mathit{map\text{-}seq}\ \mathit{id})) \circ \mathit{split}\ 1\ \circ\\
&\quad \mathit{join} \circ \mathit{map\text{-}warp}\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

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)
```

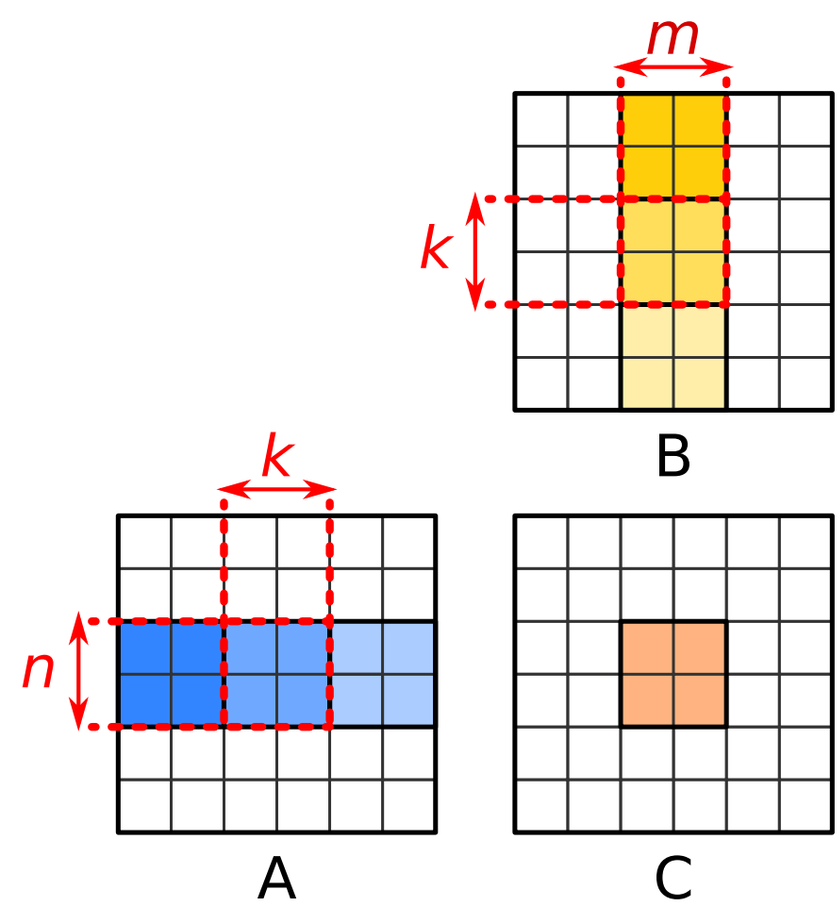
Tiling as a Rewrite Rules

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)
```

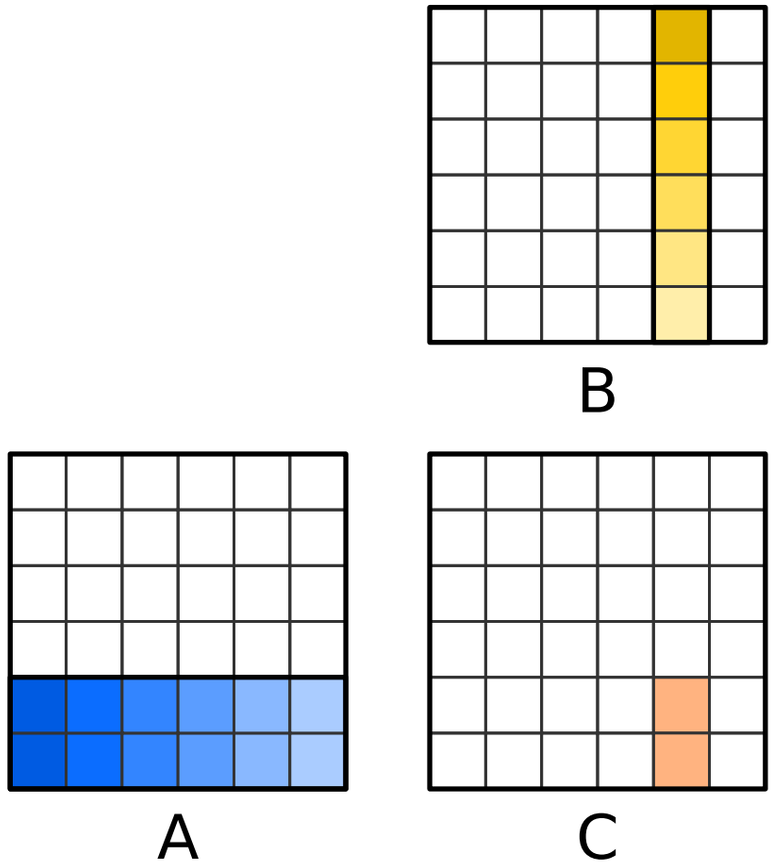


Register Blocking as a Rewrite Rules

```
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)
```

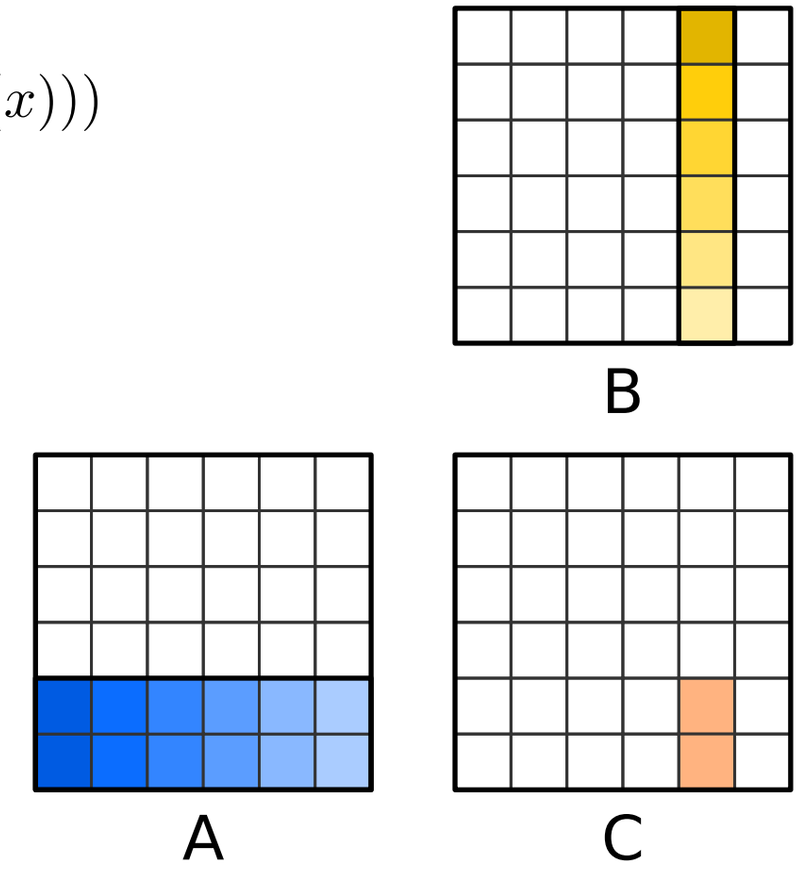
↓ Apply blocking rules

```
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)
```

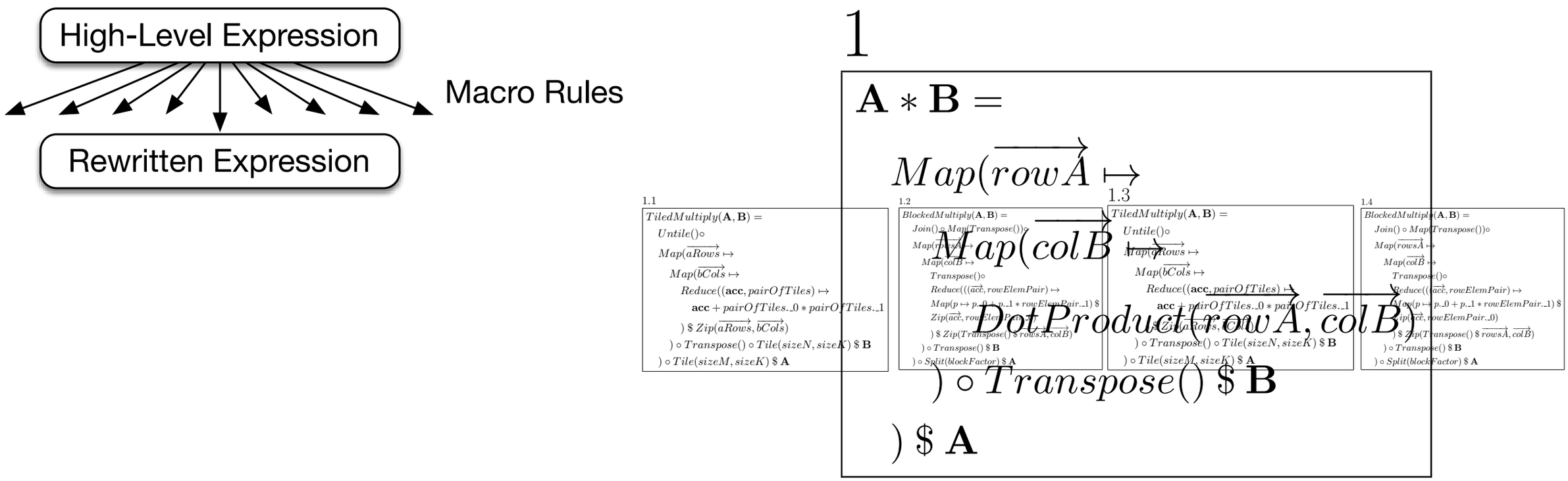


Register Blocking as a Rewrite Rules

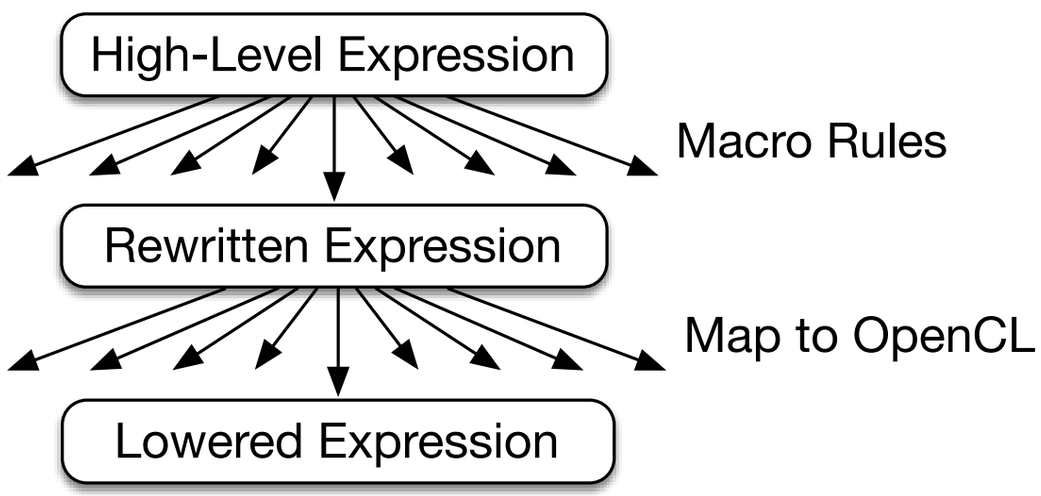
$$
\begin{aligned}
&registerBlocking = \\
&\quad Map(f) \Rightarrow Join() \circ Map(Map(f)) \circ Split(k) \\
&\quad Map(a \mapsto Map(b \mapsto f(a,b))) \Rightarrow Transpose() \circ Map(b \mapsto Map(a \mapsto f(a,b))) \\
&\quad Map(f \circ g) \Rightarrow Map(f) \circ Map(g) \\
&\quad Map(Reduce(f))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



Exploration Strategy



Exploration Strategy



1.3

$$
\begin{aligned}
&TiledMultiply(\mathbf{A},\mathbf{B}) = \\
&\quad Untile() \circ \\
&\quad\quad MapWrg(1)(\overrightarrow{aRows} \mapsto \\
&\quad\quad\quad MapWrg(0)(\overrightarrow{bCols} \mapsto \\
&\quad\quad\quad\quad ReduceSeq((\mathbf{acc}, pairOfTiles) \mapsto \\
&\quad\quad\quad\quad\quad \mathbf{acc} + toLocal(pairOfTiles.\_0) \\
&\quad\quad\quad\quad\quad\quad * toLocal(pairOfTiles.\_1) \\
&\quad\quad\quad\quad) \$ Zip(\overrightarrow{aRows}, \overrightarrow{bCols}) \\
&\quad\quad\quad) \circ Transpose() \circ Tile(sizeN, sizeK) \$ \mathbf{B} \\
&\quad\quad) \circ Tile(sizeM, sizeK) \$ \mathbf{A}
\end{aligned}
$$

1.3.1

$$
\begin{aligned}
&TiledMultiply(\mathbf{A},\mathbf{B}) = \\
&\quad Untile() \circ \\
&\quad\quad MapWrg(1)(\overrightarrow{aRows} \mapsto \\
&\quad\quad\quad MapWrg(0)(\overrightarrow{bCols} \mapsto \\
&\quad\quad\quad\quad ReduceSeq((\mathbf{acc}, pairOfTiles) \mapsto \\
&\quad\quad\quad\quad\quad \mathbf{acc} + toLocal(pairOfTiles.\_0) \\
&\quad\quad\quad\quad\quad\quad * toLocal(pairOfTiles.\_1) \\
&\quad\quad\quad\quad) \$ Zip(\overrightarrow{aRows}, \overrightarrow{bCols}) \\
&\quad\quad\quad) \circ Transpose() \circ Tile(sizeN, sizeK) \$ \mathbf{B} \\
&\quad\quad) \circ Tile(sizeM, sizeK) \$ \mathbf{A}
\end{aligned}
$$

1.3.2

$$
\begin{aligned}
&TiledMultiply(\mathbf{A},\mathbf{B}) = \\
&\quad Untile() \circ \\
&\quad\quad MapWrg(1)(\overrightarrow{aRows} \mapsto \\
&\quad\quad\quad MapWrg(0)(\overrightarrow{bCols} \mapsto \\
&\quad\quad\quad\quad ReduceSeq((\mathbf{acc}, pairOfTiles) \mapsto \\
&\quad\quad\quad\quad\quad \mathbf{acc} + toLocal(pairOfTiles.\_0) \\
&\quad\quad\quad\quad\quad\quad * toLocal(pairOfTiles.\_1) \\
&\quad\quad\quad\quad) \$ Zip(\overrightarrow{aRows}, \overrightarrow{bCols}) \\
&\quad\quad\quad) \circ Transpose() \circ Tile(sizeN, sizeK) \$ \mathbf{B} \\
&\quad\quad) \circ Tile(sizeM, sizeK) \$ \mathbf{A}
\end{aligned}
$$

1.3.3

$$
\begin{aligned}
&TiledMultiply(\mathbf{A},\mathbf{B}) = \\
&\quad Untile() \circ \\
&\quad\quad MapWrg(1)(\overrightarrow{aRows} \mapsto \\
&\quad\quad\quad MapWrg(0)(\overrightarrow{bCols} \mapsto \\
&\quad\quad\quad\quad ReduceSeq((\mathbf{acc}, pairOfTiles) \mapsto \\
&\quad\quad\quad\quad\quad \mathbf{acc} + toLocal(pairOfTiles.\_0) \\
&\quad\quad\quad\quad\quad\quad * toLocal(pairOfTiles.\_1) \\
&\quad\quad\quad\quad) \$ Zip(\overrightarrow{aRows}, \overrightarrow{bCols}) \\
&\quad\quad\quad) \circ Transpose() \circ Tile(sizeN, sizeK) \$ \mathbf{B} \\
&\quad\quad) \circ Tile(sizeM, sizeK) \$ \mathbf{A}
\end{aligned}
$$

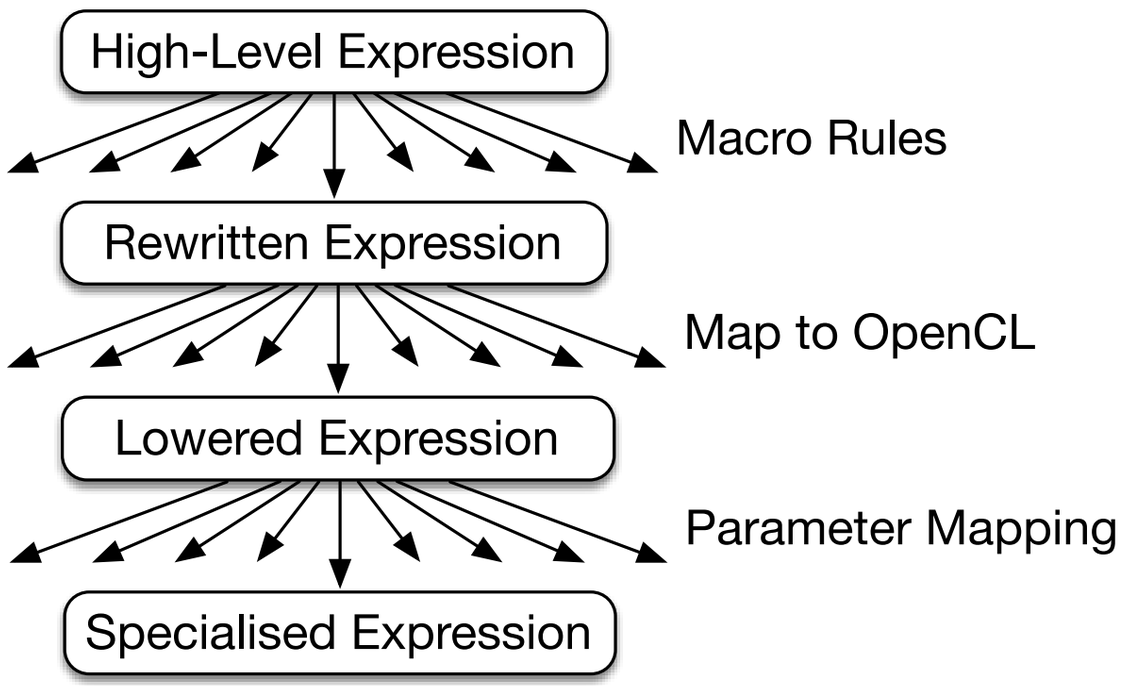
$$\overrightarrow{aRows} \mapsto Map$$

$$\overrightarrow{bCols} \mapsto Map$$

$$(\mathbf{acc}, pairOfTiles) \mapsto Reduce$$

$$\mathbf{acc} + pairOfTiles.\_0 * pairOfTiles.\_1$$

Exploration Strategy



1.3.2

$$
\begin{aligned}
&TiledMultiply(\mathbf{A},\mathbf{B}) = \\
&\quad Untile()\circ \\
&\quad MapWrg(1)(\overrightarrow{aRows} \mapsto \\
&\quad\quad MapWrg(0)(\overrightarrow{bCols} \mapsto \\
&\quad\quad\quad ReduceSeq((\mathbf{acc}, pairOfTiles) \mapsto \\
&\quad\quad\quad\quad \mathbf{acc} + toLocal(pairOfTiles.\_0) \\
&\quad\quad\quad\quad * toLocal(pairOfTiles.\_1) \\
&\quad\quad\quad) \$ Zip(\overrightarrow{aRows}, \overrightarrow{bCols}) \\
&\quad\quad) \circ Transpose() \circ Tile(128,16) \$ \mathbf{B} \\
&\quad) \circ Tile(128,16) \$ \mathbf{A}
\end{aligned}
$$

1.3.2.1

$$
\begin{aligned}
&TiledMultiply(\mathbf{A},\mathbf{B}) = \\
&\quad Untile()\circ \\
&\quad MapWrg(1)(\overrightarrow{aRows} \mapsto \\
&\quad\quad MapWrg(0)(\overrightarrow{bCols} \mapsto \\
&\quad\quad\quad ReduceSeq((\mathbf{acc}, pairOfTiles) \mapsto \\
&\quad\quad\quad\quad \mathbf{acc} + toLocal(pairOfTiles.\_0) \\
&\quad\quad\quad\quad * toLocal(pairOfTiles.\_1) \\
&\quad\quad\quad) \$ Zip(\overrightarrow{aRows}, \overrightarrow{bCols}) \\
&\quad\quad) \circ Transpose() \circ Tile(128,16) \$ \mathbf{B} \\
&\quad) \circ Tile(128,16) \$ \mathbf{A}
\end{aligned}
$$

1.3.2.2

$$
\begin{aligned}
&TiledMultiply(\mathbf{A},\mathbf{B}) = \\
&\quad Untile()\circ \\
&\quad MapWrg(1)(\overrightarrow{aRows} \mapsto \\
&\quad\quad MapWrg(0)(\overrightarrow{bCols} \mapsto \\
&\quad\quad\quad ReduceSeq((\mathbf{acc}, pairOfTiles) \mapsto \\
&\quad\quad\quad\quad \mathbf{acc} + toLocal(pairOfTiles.\_0) \\
&\quad\quad\quad\quad * toLocal(pairOfTiles.\_1) \\
&\quad\quad\quad) \$ Zip(\overrightarrow{aRows}, \overrightarrow{bCols}) \\
&\quad\quad) \circ Transpose() \circ Tile(128,16) \$ \mathbf{B} \\
&\quad) \circ Tile(128,16) \$ \mathbf{A}
\end{aligned}
$$

1.3.2.3

$$
\begin{aligned}
&TiledMultiply(\mathbf{A},\mathbf{B}) = \\
&\quad Untile()\circ \\
&\quad MapWrg(1)(\overrightarrow{aRows} \mapsto \\
&\quad\quad MapWrg(0)(\overrightarrow{bCols} \mapsto \\
&\quad\quad\quad ReduceSeq((\mathbf{acc}, pairOfTiles) \mapsto \\
&\quad\quad\quad\quad \mathbf{acc} + toLocal(pairOfTiles.\_0) \\
&\quad\quad\quad\quad * toLocal(pairOfTiles.\_1) \\
&\quad\quad\quad) \$ Zip(\overrightarrow{aRows}, \overrightarrow{bCols}) \\
&\quad\quad) \circ Transpose() \circ Tile(128,16) \$ \mathbf{B} \\
&\quad) \circ Tile(128,16) \$ \mathbf{A}
\end{aligned}
$$

1.3.2.4

$$
\begin{aligned}
&TiledMultiply(\mathbf{A},\mathbf{B}) = \\
&\quad Untile()\circ \\
&\quad MapWrg(1)(\overrightarrow{aRows} \mapsto \\
&\quad\quad MapWrg(0)(\overrightarrow{bCols} \mapsto \\
&\quad\quad\quad ReduceSeq((\mathbf{acc}, pairOfTiles) \mapsto \\
&\quad\quad\quad\quad \mathbf{acc} + toLocal(pairOfTiles.\_0) \\
&\quad\quad\quad\quad * toLocal(pairOfTiles.\_1) \\
&\quad\quad\quad) \$ Zip(\overrightarrow{aRows}, \overrightarrow{bCols}) \\
&\quad\quad) \circ Transpose() \circ Tile(128,16) \$ \mathbf{B} \\
&\quad) \circ Tile(128,16) \$ \mathbf{A}
\end{aligned}
$$

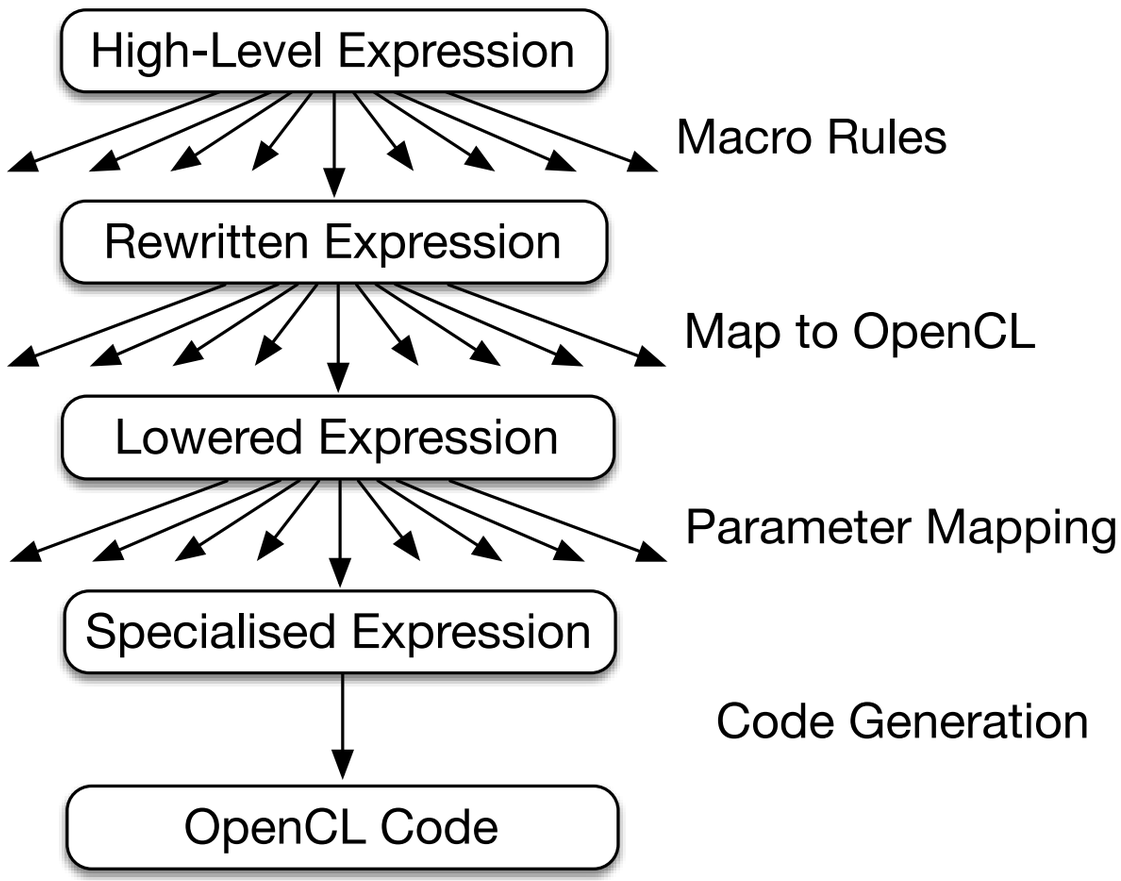
1.3.2.5

$$
\begin{aligned}
&TiledMultiply(\mathbf{A},\mathbf{B}) = \\
&\quad Untile()\circ \\
&\quad MapWrg(1)(\overrightarrow{aRows} \mapsto \\
&\quad\quad MapWrg(0)(\overrightarrow{bCols} \mapsto \\
&\quad\quad\quad ReduceSeq((\mathbf{acc}, pairOfTiles) \mapsto \\
&\quad\quad\quad\quad \mathbf{acc} + toLocal(pairOfTiles.\_0) \\
&\quad\quad\quad\quad * toLocal(pairOfTiles.\_1) \\
&\quad\quad\quad) \$ Zip(\overrightarrow{aRows}, \overrightarrow{bCols}) \\
&\quad\quad) \circ Transpose() \circ Tile(128,16) \$ \mathbf{B} \\
&\quad) \circ Tile(128,16) \$ \mathbf{A}
\end{aligned}
$$

1.3.2.6

$$
\begin{aligned}
&TiledMultiply(\mathbf{A},\mathbf{B}) = \\
&\quad Untile()\circ \\
&\quad MapWrg(1)(\overrightarrow{aRows} \mapsto \\
&\quad\quad MapWrg(0)(\overrightarrow{bCols} \mapsto \\
&\quad\quad\quad ReduceSeq((\mathbf{acc}, pairOfTiles) \mapsto \\
&\quad\quad\quad\quad \mathbf{acc} + toLocal(pairOfTiles.\_0) \\
&\quad\quad\quad\quad * toLocal(pairOfTiles.\_1) \\
&\quad\quad\quad) \$ Zip(\overrightarrow{aRows}, \overrightarrow{bCols}) \\
&\quad\quad) \circ Transpose() \circ Tile(128,16) \$ \mathbf{B} \\
&\quad) \circ Tile(128,16) \$ \mathbf{A}
\end{aligned}
$$

Exploration Strategy



1.3.2.5

```
1  kernel mm_amd_opt(global float *A, B, C,
2                    int M,N,K)
3  local float tileA [512]; tileB [512];
4
5  private float acc_0; ...; acc_31;
6  private float blockOfB_0; ...; blockOfB_3;
7  private float blockOfA_0; ...; blockOfA_7;
8
9  int lid0 = local_id(0); lid1 = local_id(1);
10 int wid0 = group_id(0); wid1 = group_id(1);
11
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
15   acc_0 = 0.0f; ...; acc_31 = 0.0f;
16   for (int i=0; i<K/8; i++) {
17    vstore4(vload4(lid1*M/4+2*i*M+16*w1+lid0,A), 16*lid1+lid0, tileA);
18    vstore4(vload4(lid1*N/4+2*i*N+16*w0+lid0,B), 16*lid1+lid0, tileB);
19    barrier (...) ;
20
21    for (int j = 0; j<8; j++) {
22     blockOfA_0 = tileA[0+64*j+lid1*8]; ...; blockOfA_7 = tileA[7+64*j+lid1*8];
23     blockOfB_0 = tileB[0 +64*j+lid0]; ...; blockOfB_3 = tileB[48+64*j+lid0];
24
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 (...) ;
31   }
32
33   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;
34   C[16+8*lid1*N+64*w0+64*w1*N+1*N+lid0]=acc_1; ...; C[16+8*lid1*N+64*w0+64*w1*N+7*N+lid0]=acc_29;
35   C[32+8*lid1*N+64*w0+64*w1*N+0*N+lid0]=acc_2; ...; C[32+8*lid1*N+64*w0+64*w1*N+7*N+lid0]=acc_30;
36   C[48+8*lid1*N+64*w0+64*w1*N+0*N+lid0]=acc_3; ...; C[48+8*lid1*N+64*w0+64*w1*N+7*N+lid0]=acc_31;
37 } } }
```

$$
\begin{aligned}
&TiledMultiply(\mathbf{A},\mathbf{B}) = \\
&Untile() \circ \\
&MapWrg(1)(\overrightarrow{aRows} \mapsto \\
&MapWrg(0)(\overrightarrow{bCols} \mapsto \\
&ReduceSeq((\mathbf{acc}, pairOfTiles) \mapsto \\
&\mathbf{acc} + toLocal(pairOfTiles.\_0) \\
&* toLocal(pairOfTiles.\_1)) \\
&) \$ Zip(\overrightarrow{aRows}, \overrightarrow{bCols}) \\
&) \circ Transpose() \circ Tile(128,16) \$ \mathbf{B} \\
&) \circ Tile(128,16) \$ \mathbf{A}
\end{aligned}
$$

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

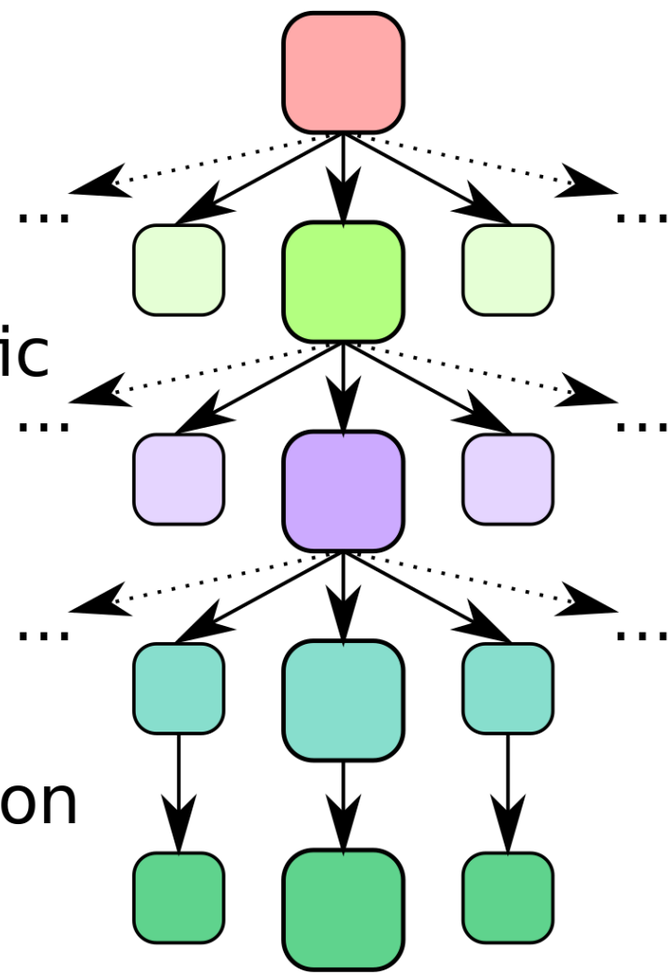
Phases:

Algorithmic Exploration

OpenCL specific Exploration

Parameter Exploration

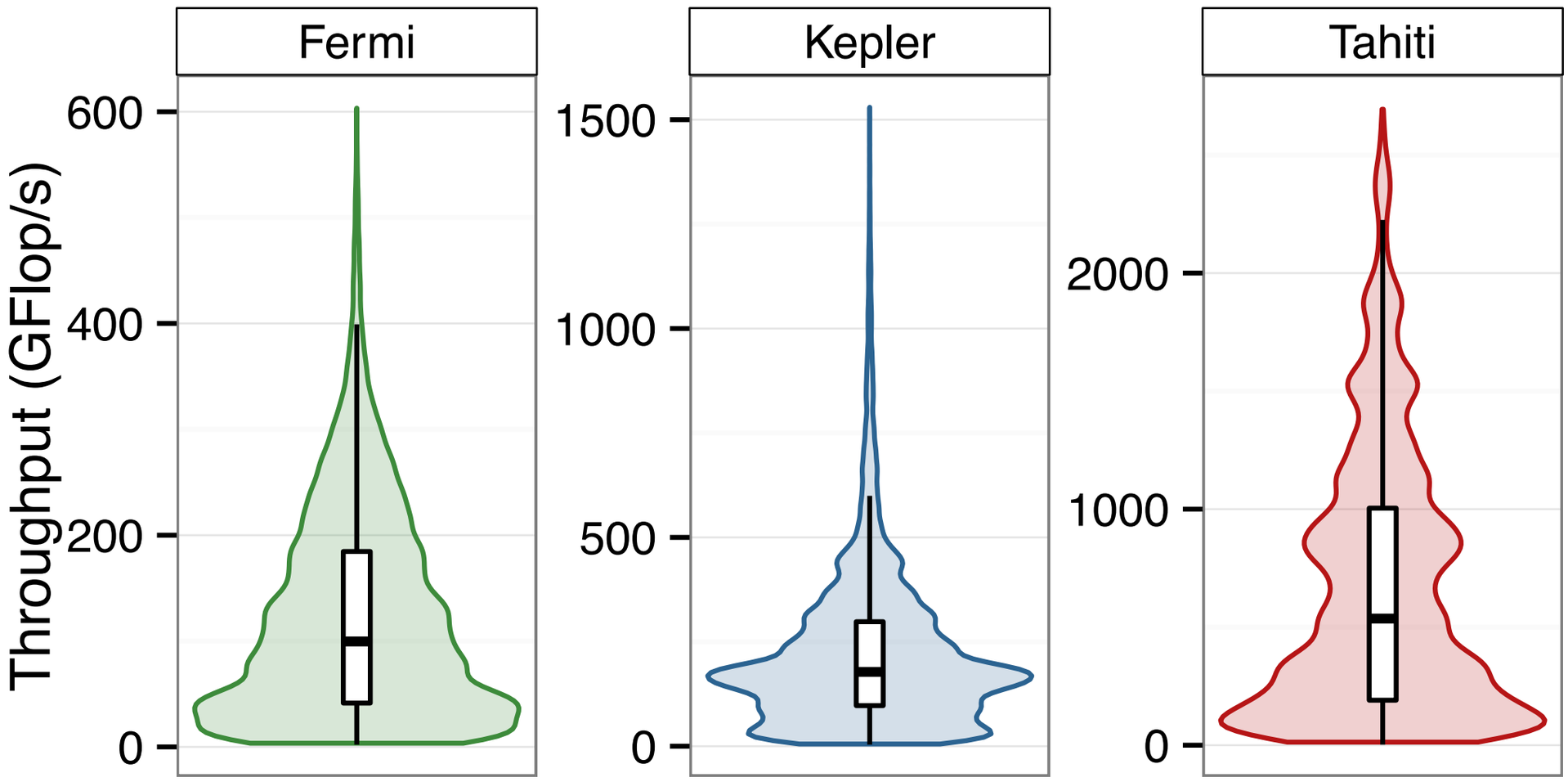
Code Generation



Program Variants:

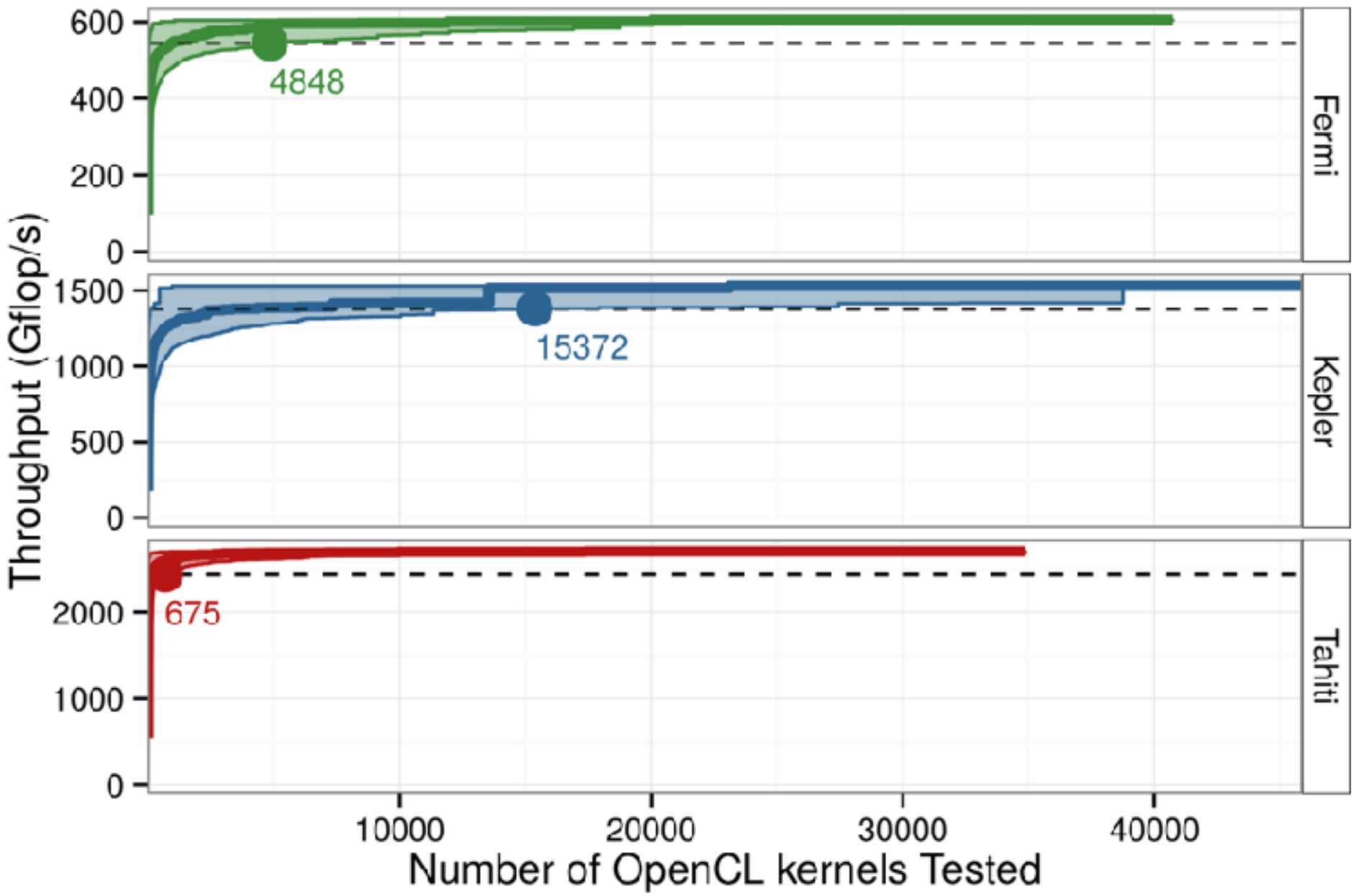
High-Level Program	1
Algorithmic Rewritten Program	8
OpenCL Specific Program	760
Fully Specialized Program	46,000
OpenCL Code	46,000

Exploration Space for Matrix Multiplication



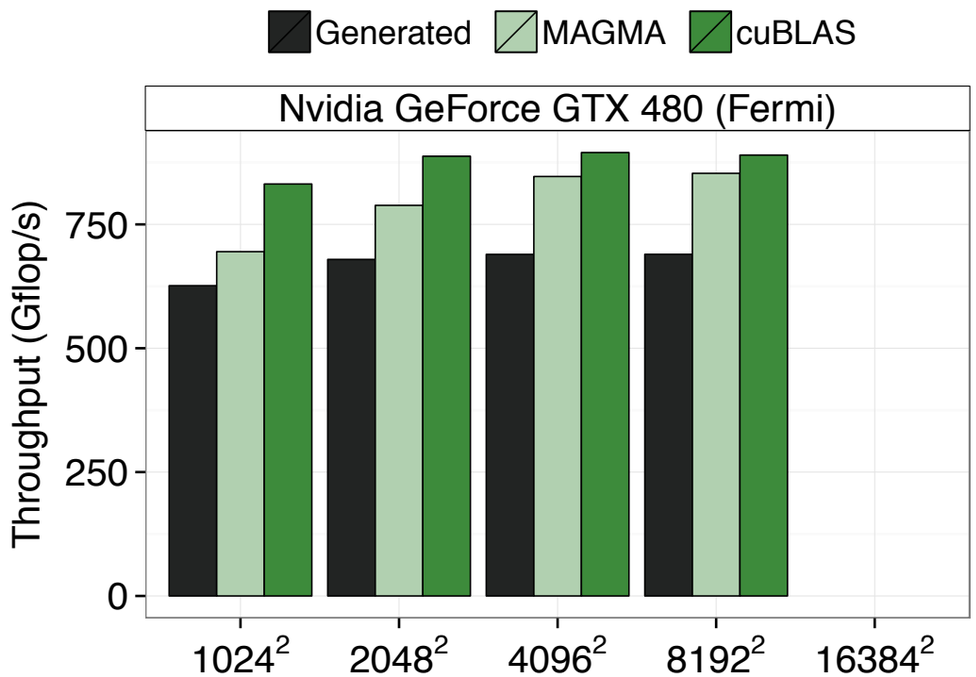
Only few OpenCL kernel with very good performance

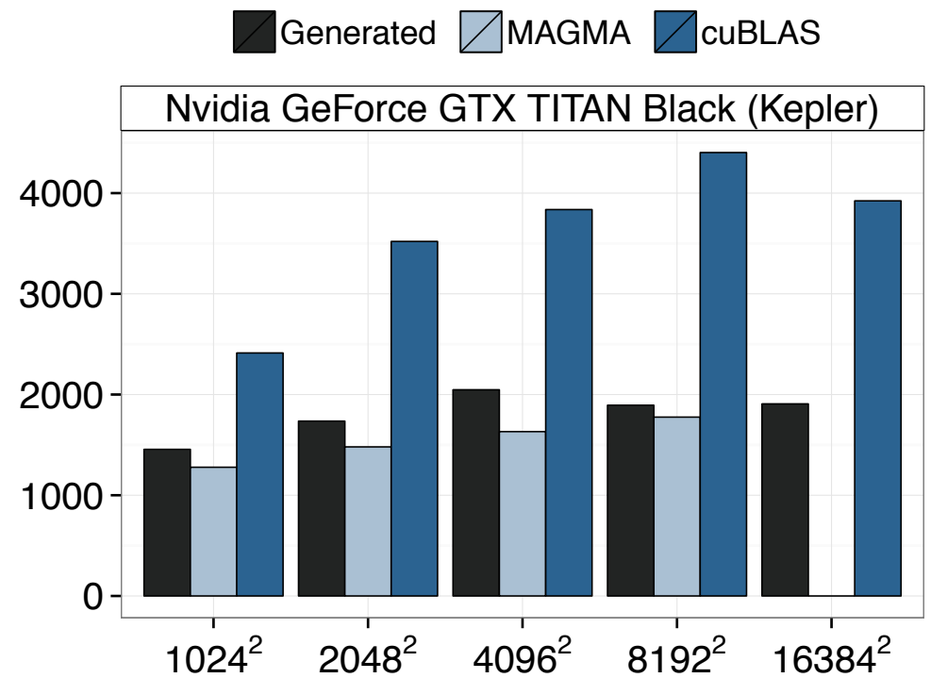
Performance Evolution for Randomised Search

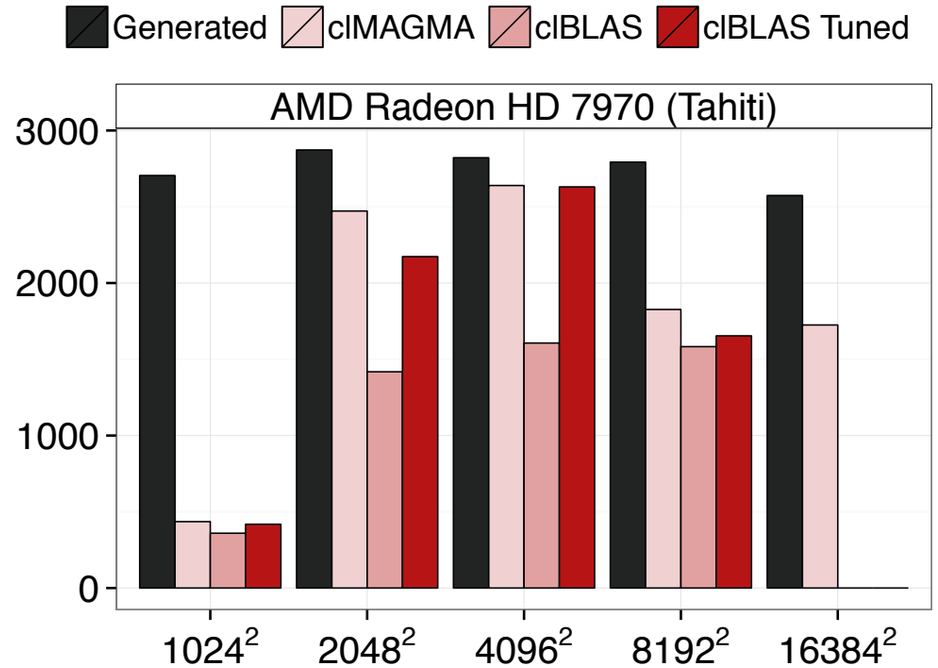


Even with a simple random search strategy one can expect to find a good performing kernel quickly

Performance Results Matrix Multiplication

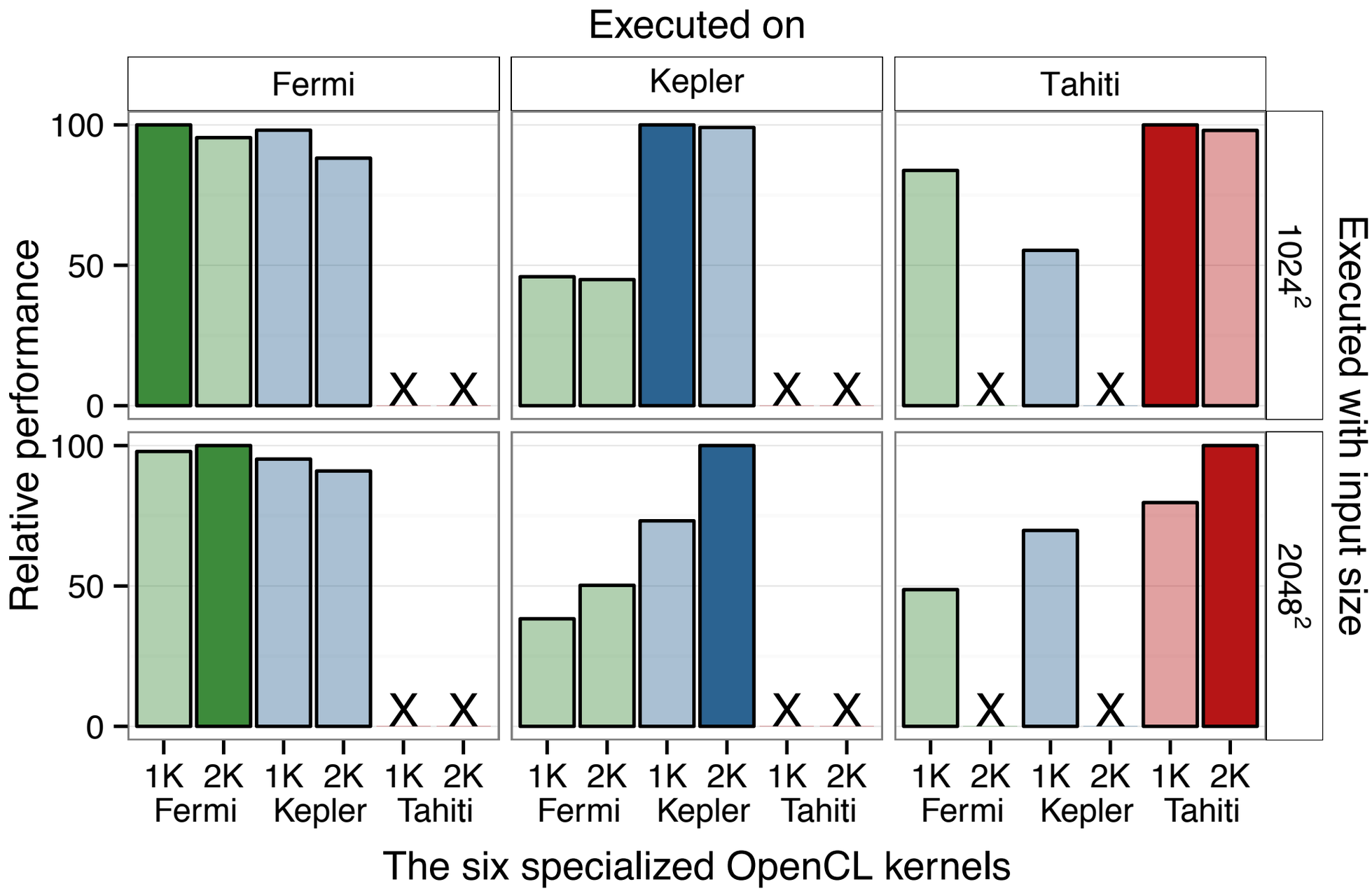






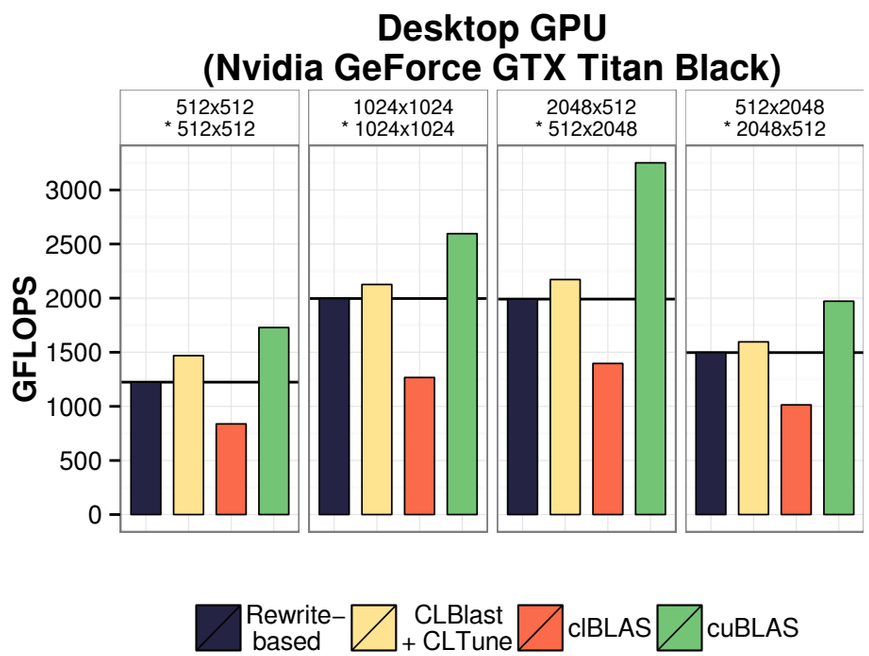
Performance close or better than hand-tuned MAGMA library

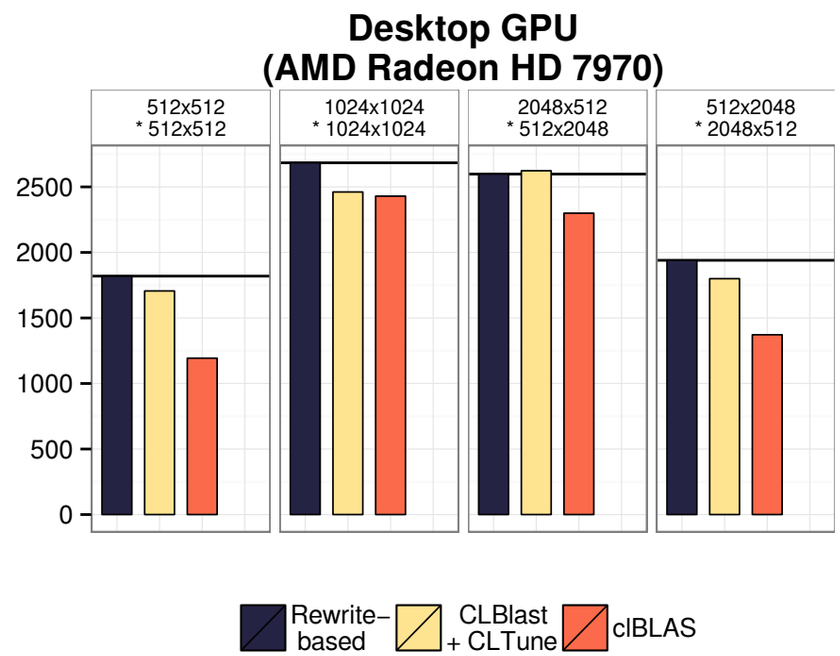
Performance Portability Matrix Multiplication

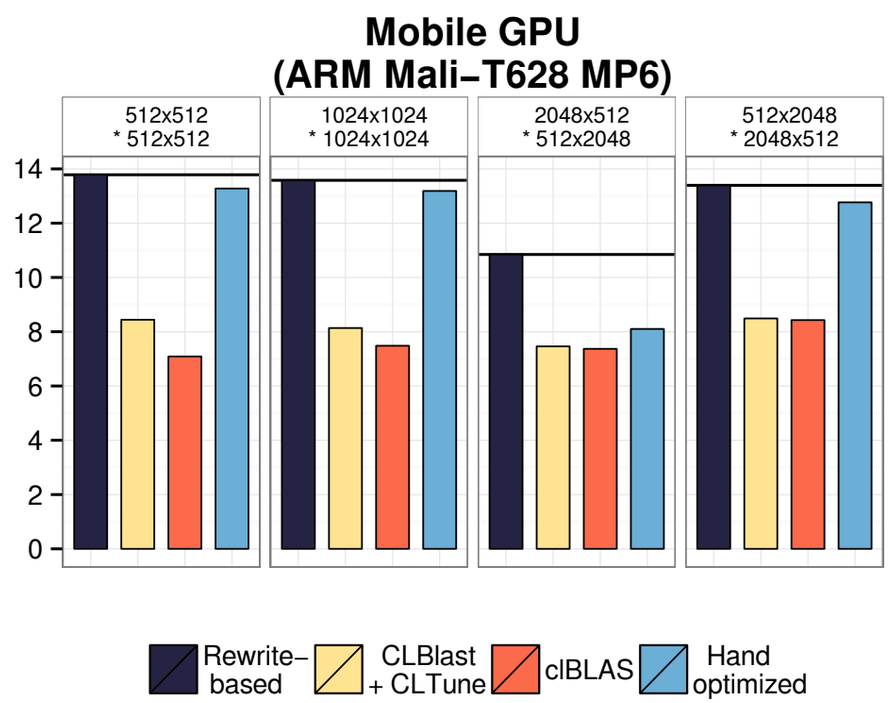


Generated kernels are specialised for device and input size

Desktop GPUs vs. Mobile GPU

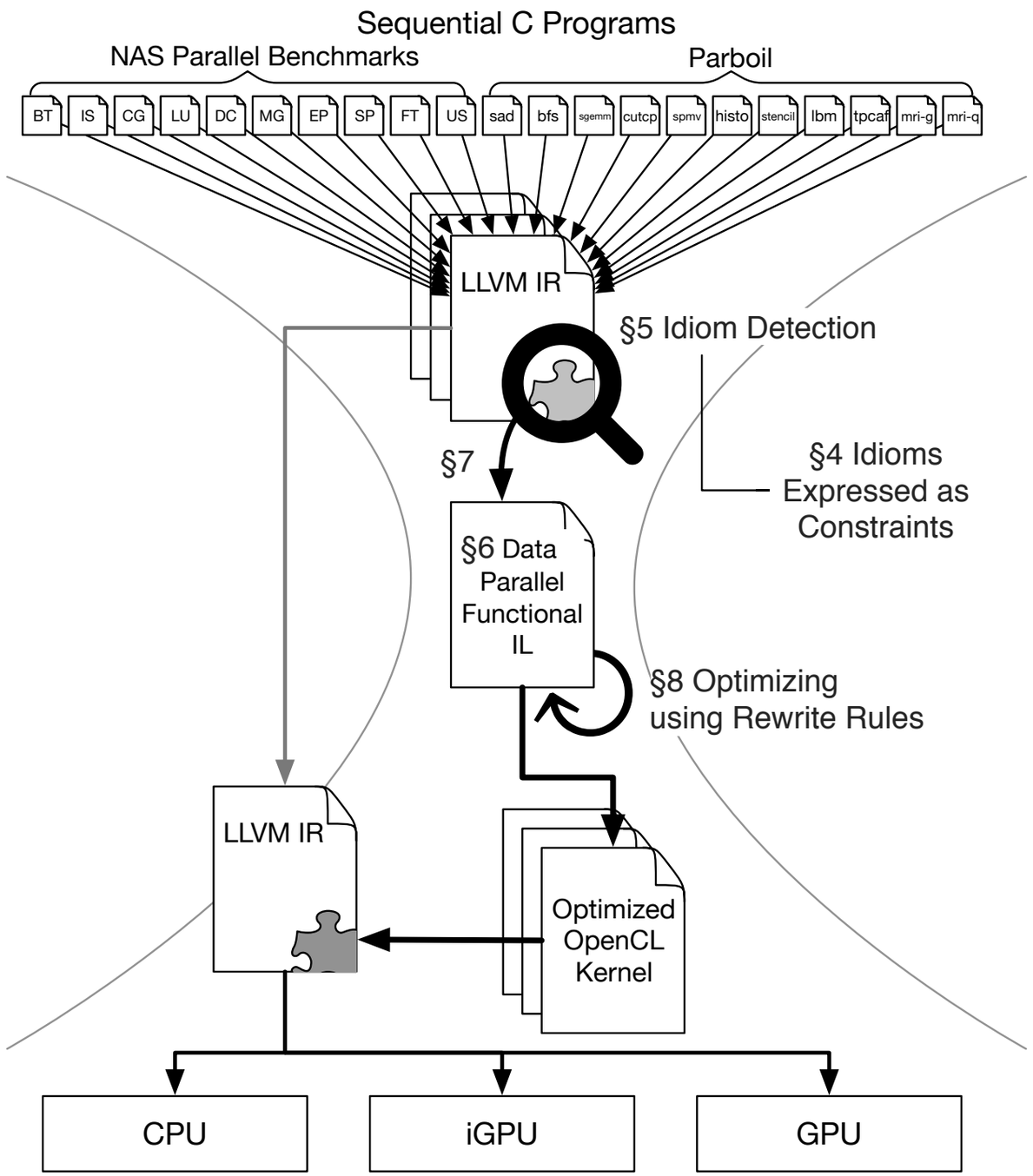




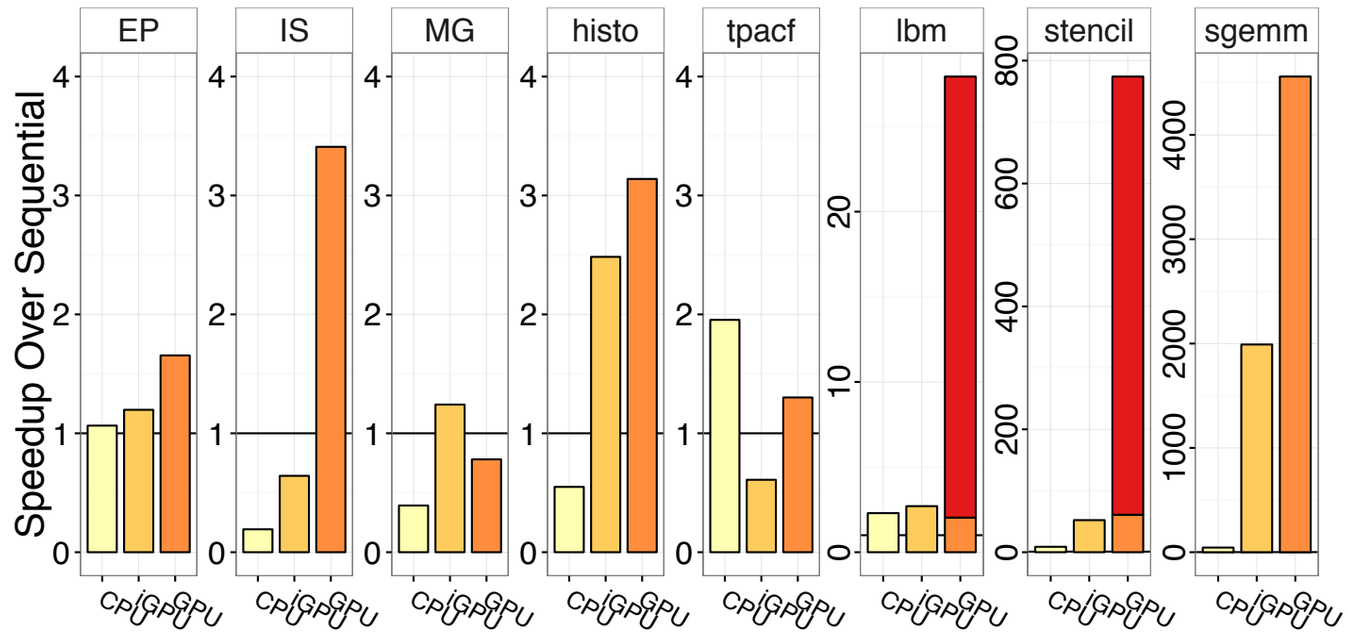


Performance portable even for mobile GPU device!

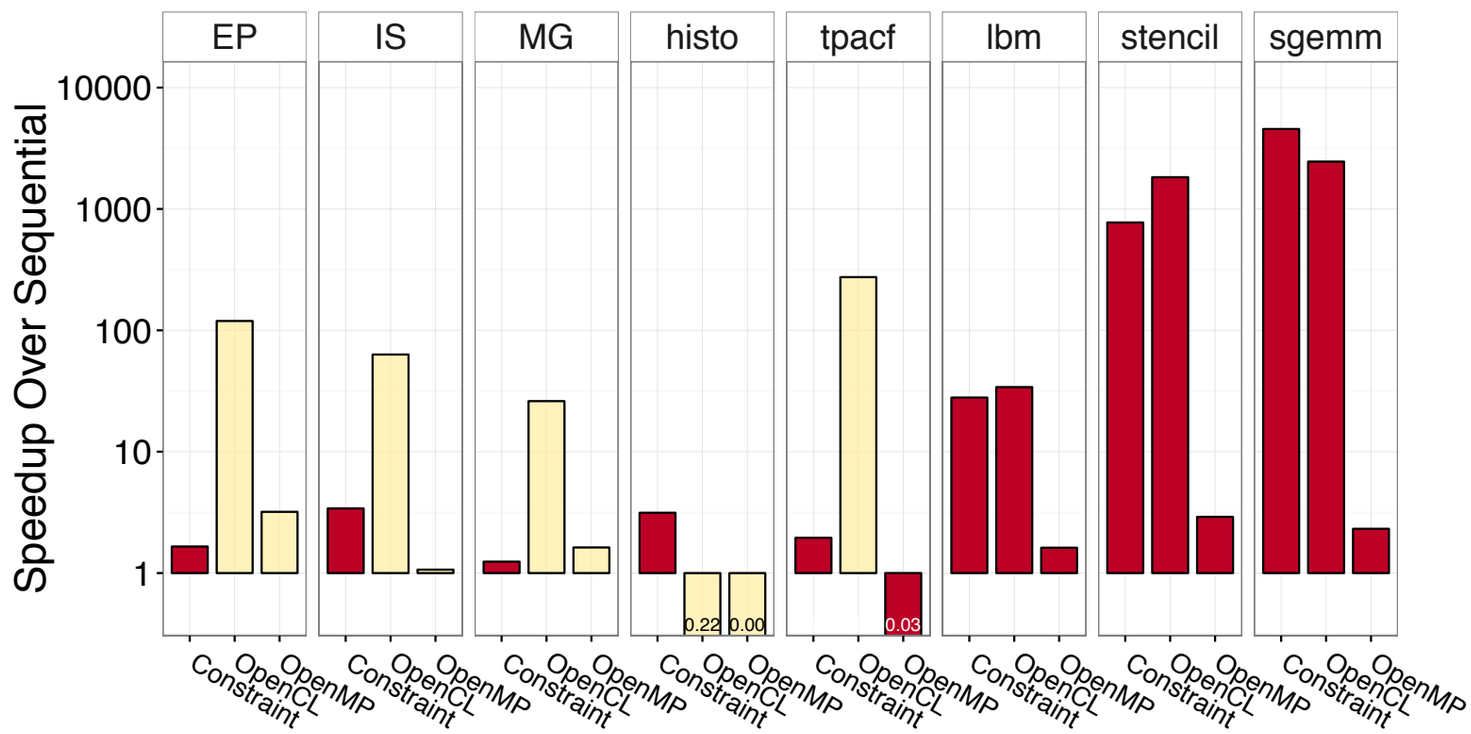
Using *Lift* as a code generation backend



Using *Lift* as a code generation backend



Heterogeneous code generation gives a speedup in all cases

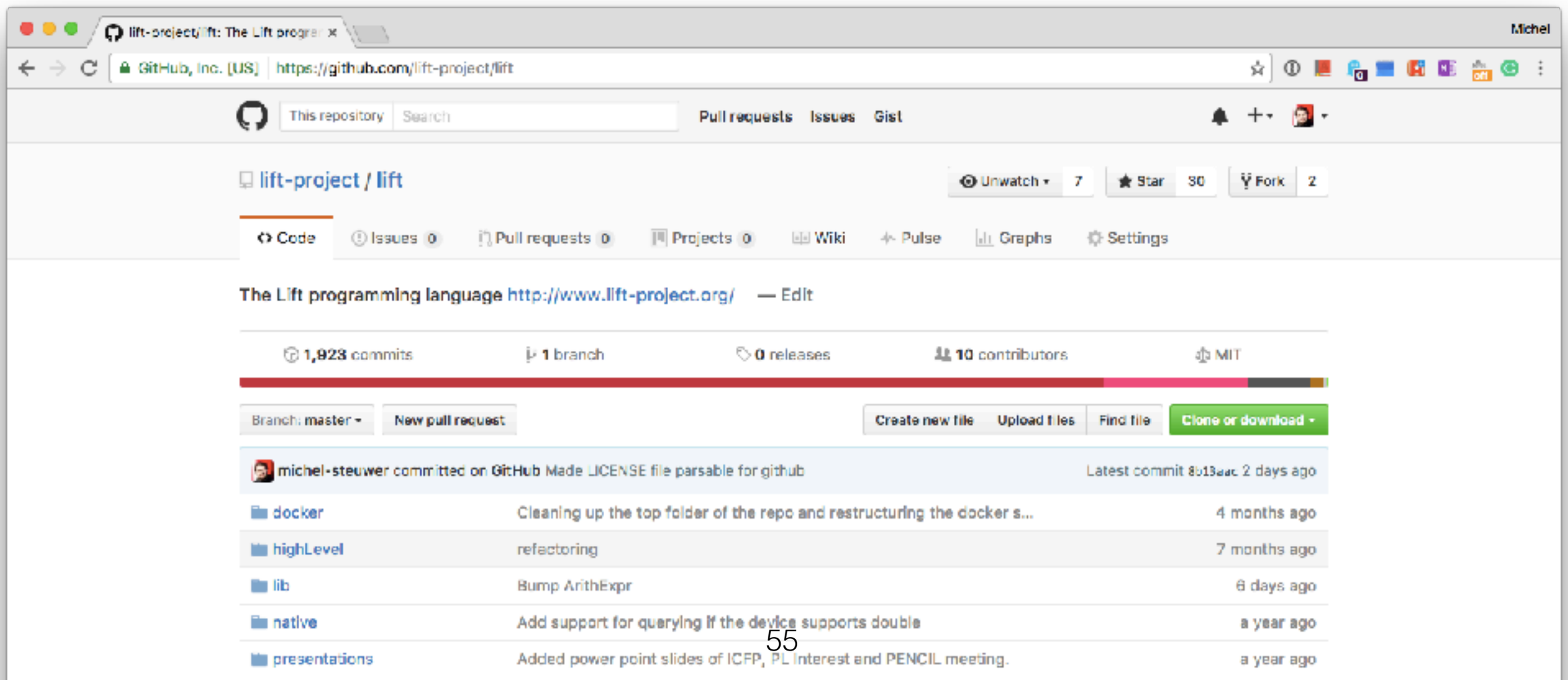


Performance close to manual written code —when parallelisation strategy is comparable

Lift is now Open-Source Software

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

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



The *Lift* Team

Christophe Dubach
Lecturer
University of Edinburgh

Michel Steuwer
Postdoc
University of Edinburgh

Toomas Remmelg
PhD Student
University of Edinburgh

Adam Harries
PhD Student
University of Edinburgh

Bastian Hagedorn
PhD Student
University of Münster

Larisa Stoltzfus
PhD Student
University of Edinburgh

Federico Pizzuti
PhD Student
University of Edinburgh

Naums Mogers
PhD Student
University of Edinburgh

The *Lift* Project: Performance Portable GPU Code Generation via Rewrite Rules

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

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

THE UNIVERSITY of EDINBURGH
informatics

icsa