Michel Steuwer

http://homepages.inf.ed.ac.uk/msteuwer/

THE UNIVERSITY of EDINBURGH

·EDINBVRGH·


THE UNIVERSITY of EDINBURGH


THE UNIVERSITY of EDINBURGH
informatics
THE UNIVERSITY of EDINBURGH
INFORMATICS FORUM


GLASGOW


GLASGOW


University
of Glasgow
VIA VERITAS VITA


University of Glasgow
School of Computing Science
VIA VERITAS VITA
Alwyn Williams Building

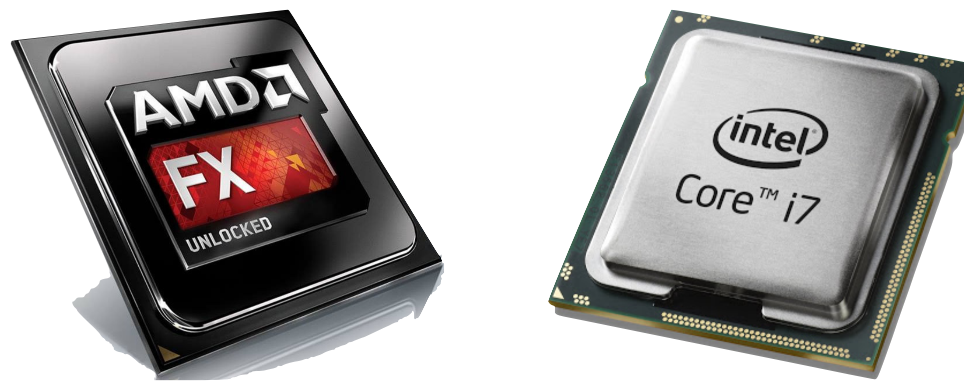
The *Lift* Project: Performance Portable Parallel Code Generation via Rewrite Rules

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

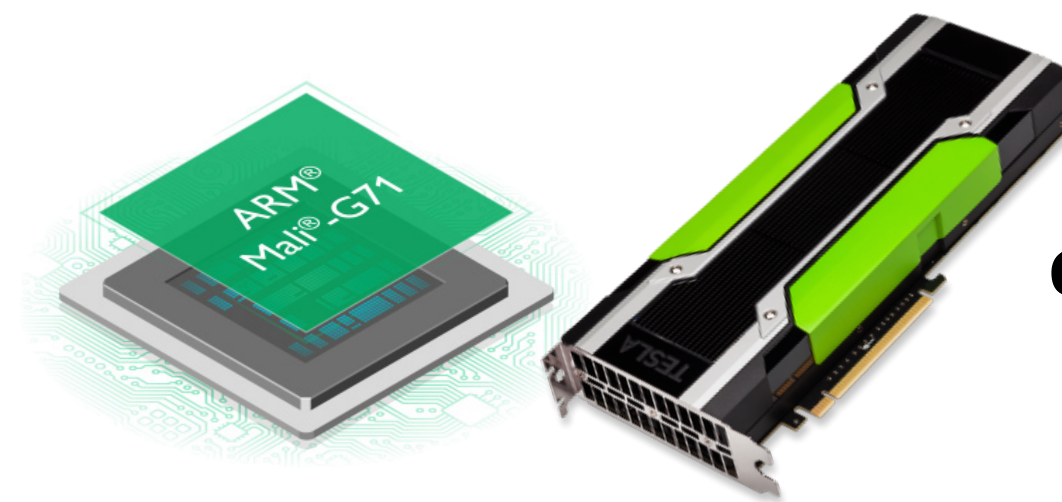
http://www.lift-project.org/

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!



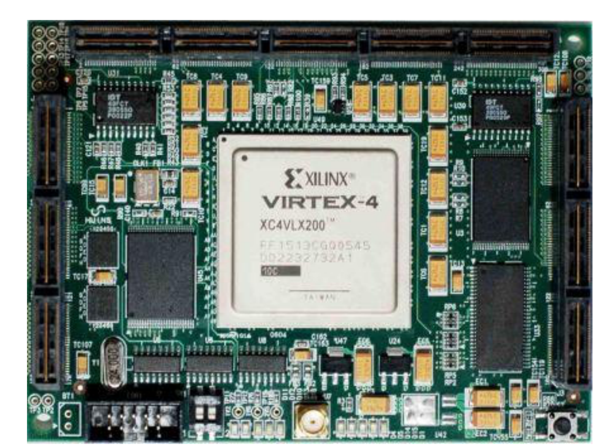
CPU



GPU

Accelerator

FPGA



Reduction Case Study

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

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

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

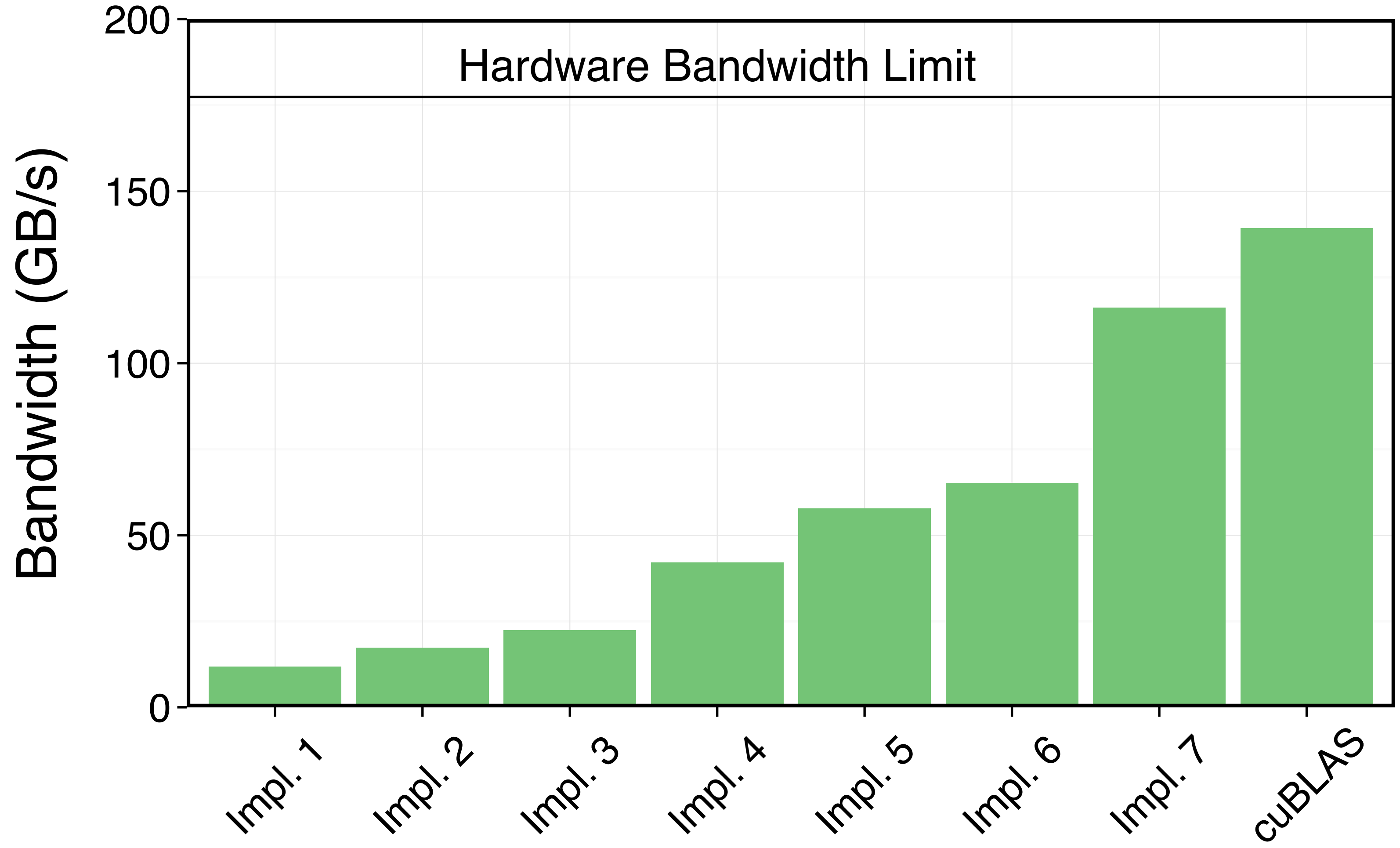
Unoptimized Implementation

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

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

Fully Optimized Implementation

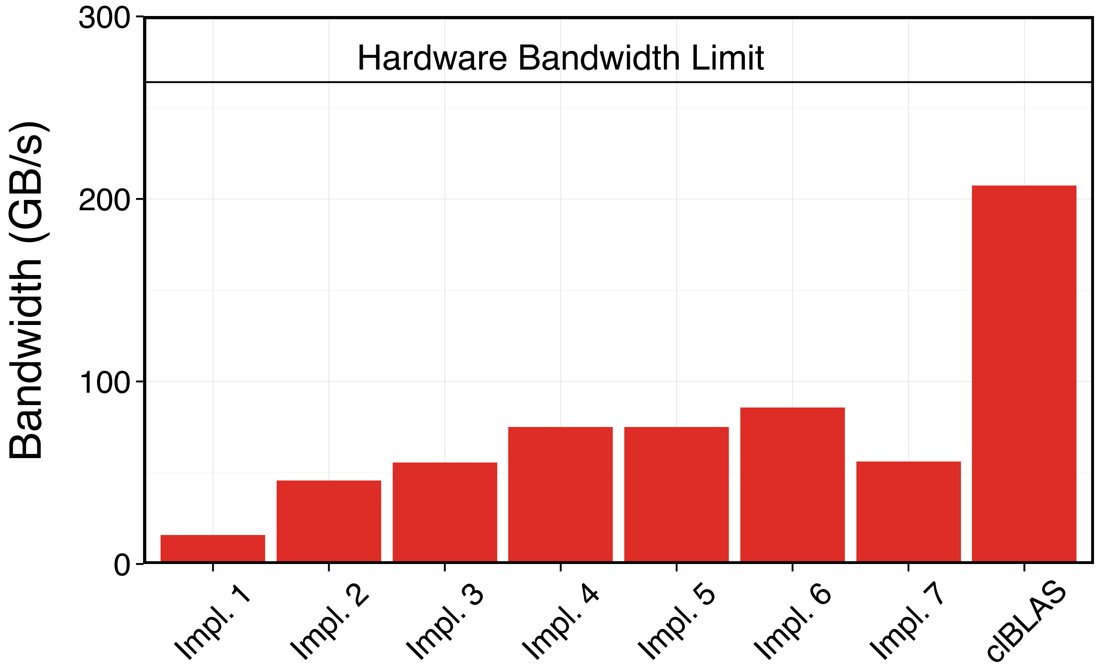
Performance Results Nvidia



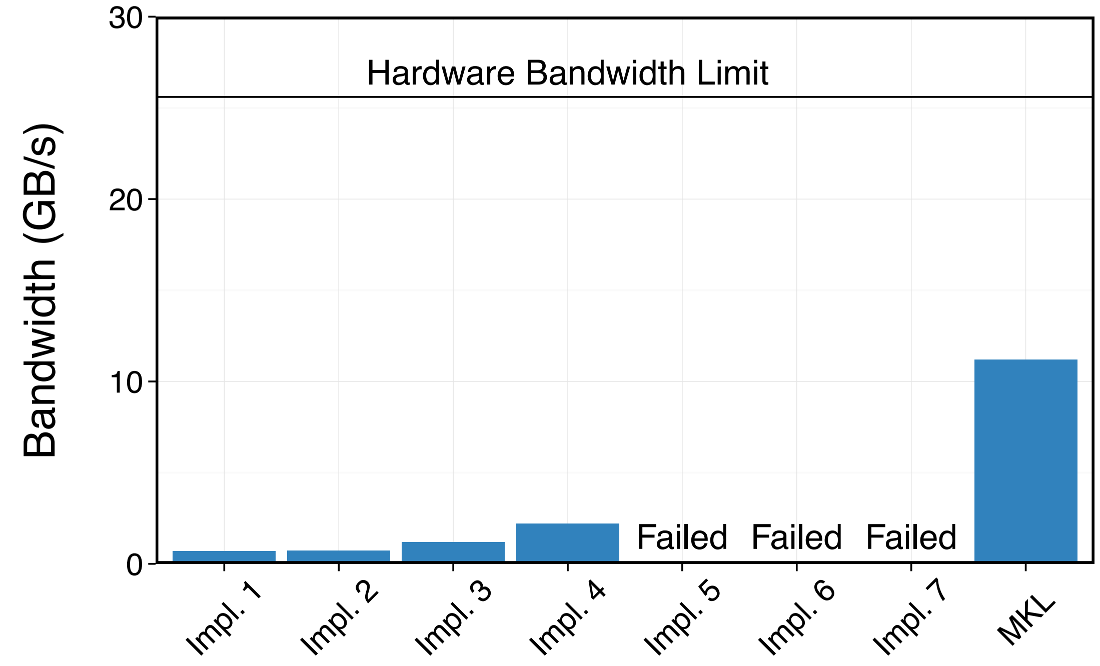
(a) Nvidia's GTX 480 GPU.

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

Performance Results AMD and Intel



(b) AMD's HD 7970 GPU.

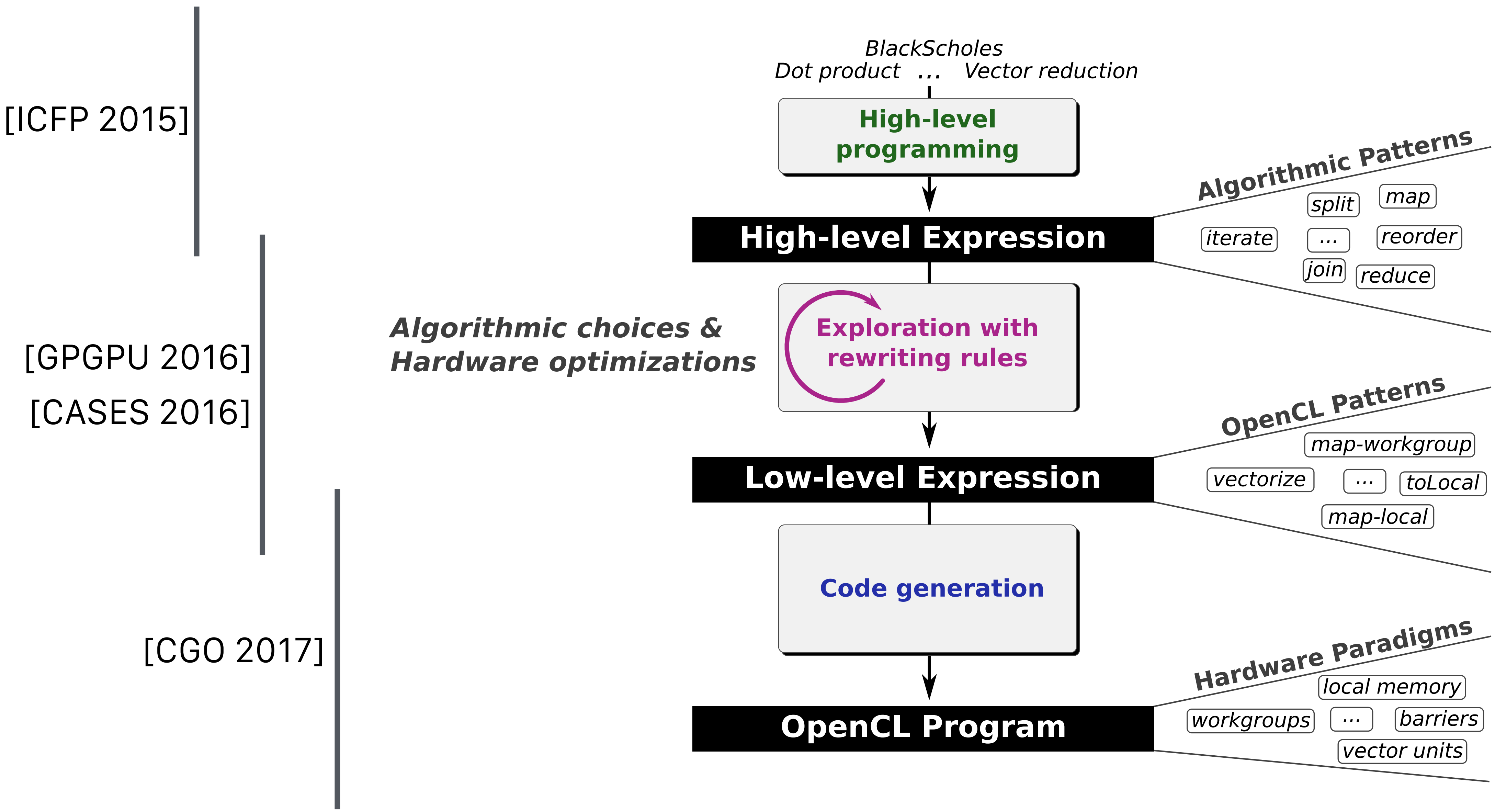


(c) Intel's E5530 dual-socket CPU.

- ... unfortunately, optimisations in OpenCL are not portable!

- **Challenge**: how to achieving portable performance?

Lift: Performance Portable GPU Code Generation via Rewrite Rules



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

Walkthrough

① $sum(vec) = reduce(+, 0, vec)$

rewrite rules

code generation

②

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

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 $sum(vec) = reduce(+, 0, vec)$

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)

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

① High-Level Programs

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

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

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

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

Walkthrough

① $sum(vec) = reduce(+, 0, vec)$

rewrite rules

code generation

②

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

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 $sum(vec) = reduce(+, 0, vec)$

rewrite rules

code generation

②

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

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:

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

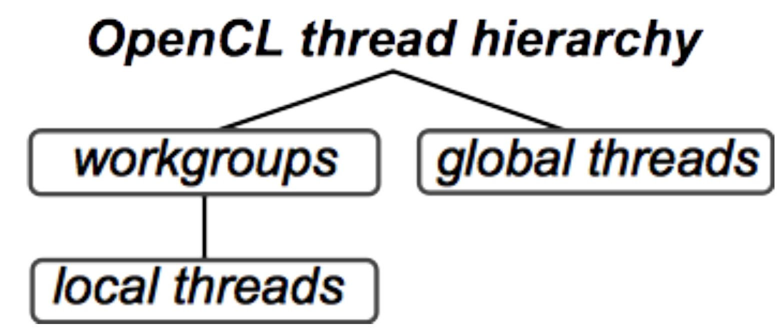
$$reducePart\ f\ \ z \rightarrow reducePart\ f\ \ z \circ reorder$$

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

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

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 $vecSum = reduce\ (+)\ 0$

rewrite rules

code generation

②

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

③

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

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

Walkthrough

① $vecSum = reduce\ (+)\ 0$

rewrite rules

code generation

②

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

③

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

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

③ Pattern based OpenCL Code Generation

- Generate OpenCL code for each OpenCL primitive

$$mapGlobal\ f\ xs \rightarrow$$

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

$$reduceSeq\ f\ z\ xs \rightarrow$$

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

$$\vdots \qquad\qquad \vdots$$

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

Walkthrough

① $vecSum = reduce\ (+)\ 0$

rewrite rules

code generation

②

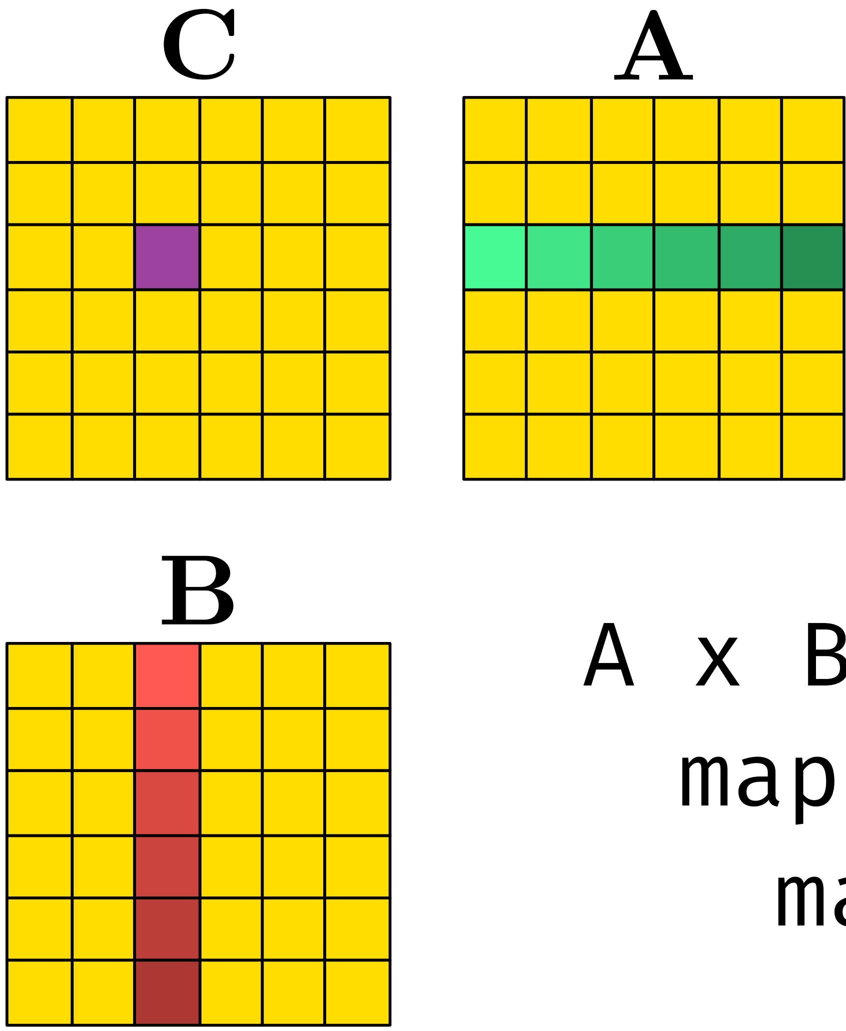
$$
\begin{array}{l}
vecSum = reduce \circ join \circ map\text{-}workgroup\ (\\
\quad join \circ toGlobal\ (map\text{-}local\ (map\text{-}seq\ id)) \circ split\ 1 \circ \\
\quad join \circ map\text{-}warp\ (\\
\quad\quad join \circ map\text{-}lane\ (reduce\text{-}seq\ (+)\ 0) \circ split\ 2\ \circ reorder\text{-}stride\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

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)
```

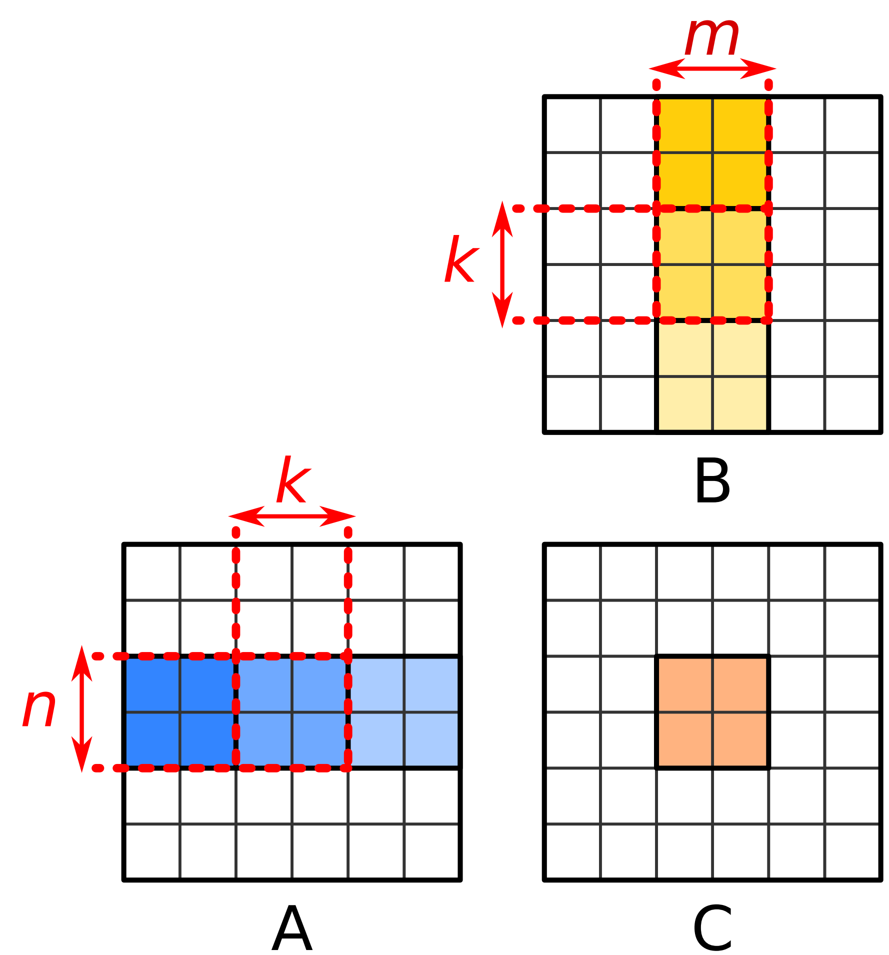
Tiling as a Rewrite Rules

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)
```

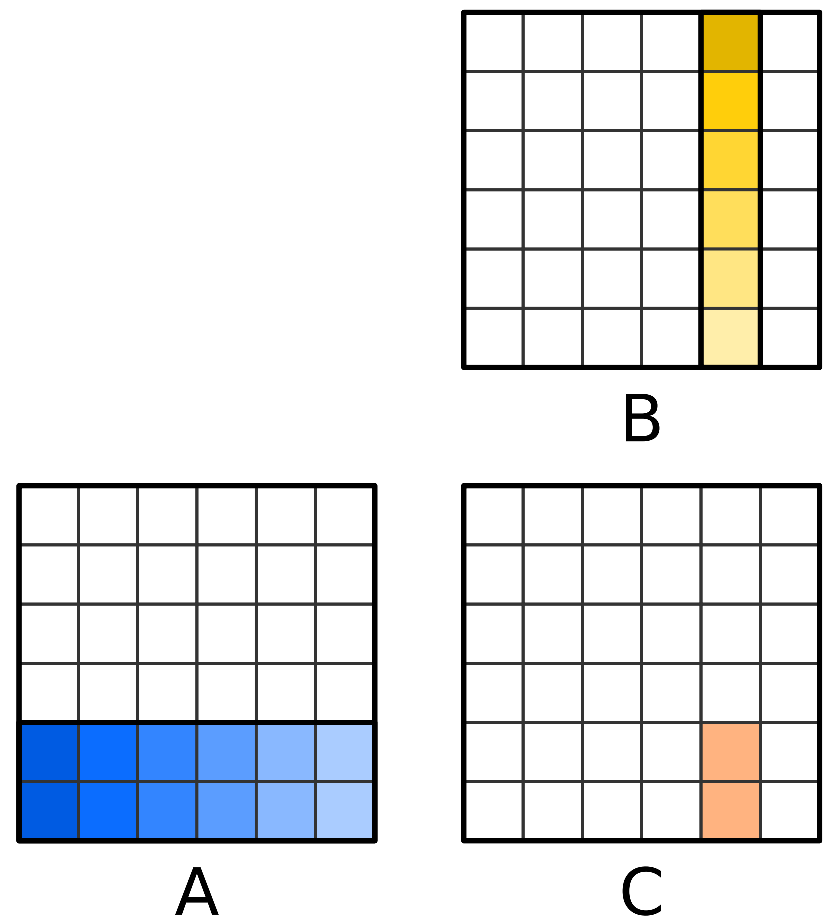


Register Blocking as a Rewrite Rules

```
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)
```

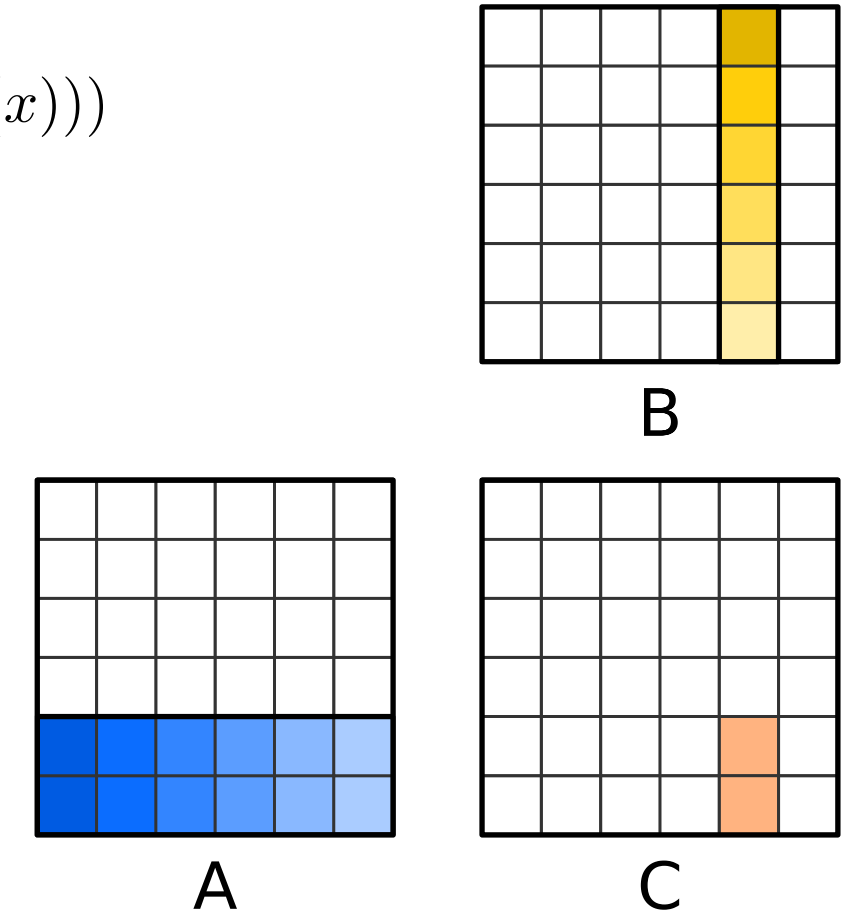
↓ Apply blocking rules

```
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)
```

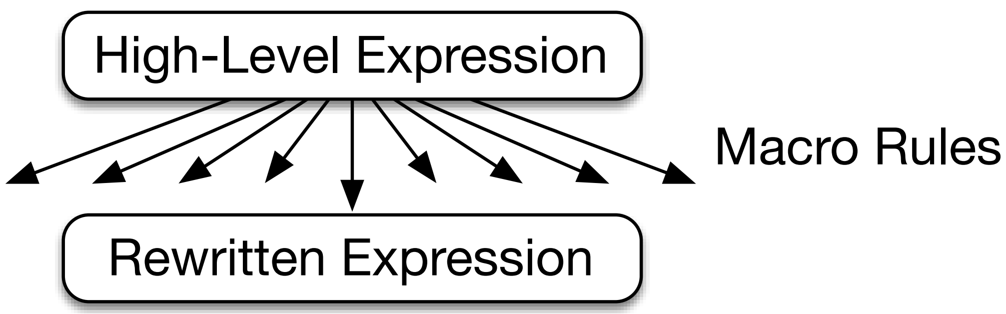


Register Blocking as a Rewrite Rules

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



Exploration Strategy



1

$$
\begin{aligned}
&\mathbf{A} * \mathbf{B} = \\
&\quad Map(\overrightarrow{rowA} \mapsto \\
&\qquad Map(\overrightarrow{colB} \mapsto \\
&\qquad\quad DotProduct(\overrightarrow{rowA}, \overrightarrow{colB}) \\
&\qquad) \circ Transpose() \ \$ \ \mathbf{B} \\
&\quad) \ \$ \ \mathbf{A}
\end{aligned}
$$

1.1

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

1.2

$$
\begin{aligned}
&BlockedMultiply(\mathbf{A}, \mathbf{B}) = \\
&\quad Join() \circ Map(Transpose()) \circ \\
&\quad Map(\overrightarrow{rowsA} \mapsto \\
&\qquad Map(\overrightarrow{colB} \mapsto \\
&\qquad\quad Transpose() \circ \\
&\qquad\quad Reduce(((\overrightarrow{acc}, rowElemPair) \mapsto \\
&\qquad\quad Map(p \mapsto p.\_0 + p.\_1 * rowElemPair.\_1) \ \$ \\
&\qquad\quad Zip(\overrightarrow{acc}, rowElemPair.\_0) \\
&\qquad\quad) \ \$ \ Zip(Transpose() \ \$ \ \overrightarrow{rowsA}, \overrightarrow{colB}) \\
&\qquad) \circ Transpose() \ \$ \ \mathbf{B} \\
&\quad) \circ Split(blockFactor) \ \$ \ \mathbf{A}
\end{aligned}
$$

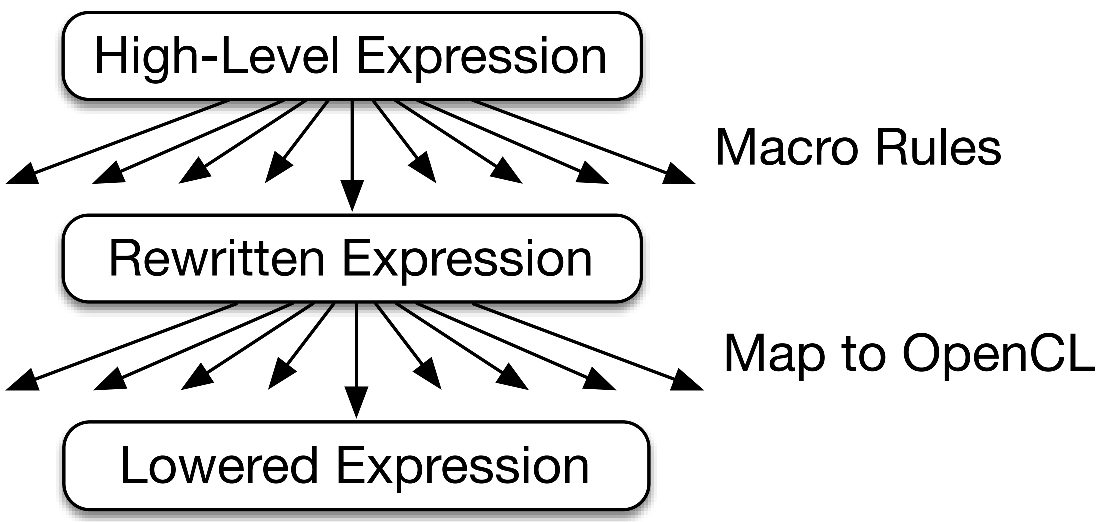
1.3

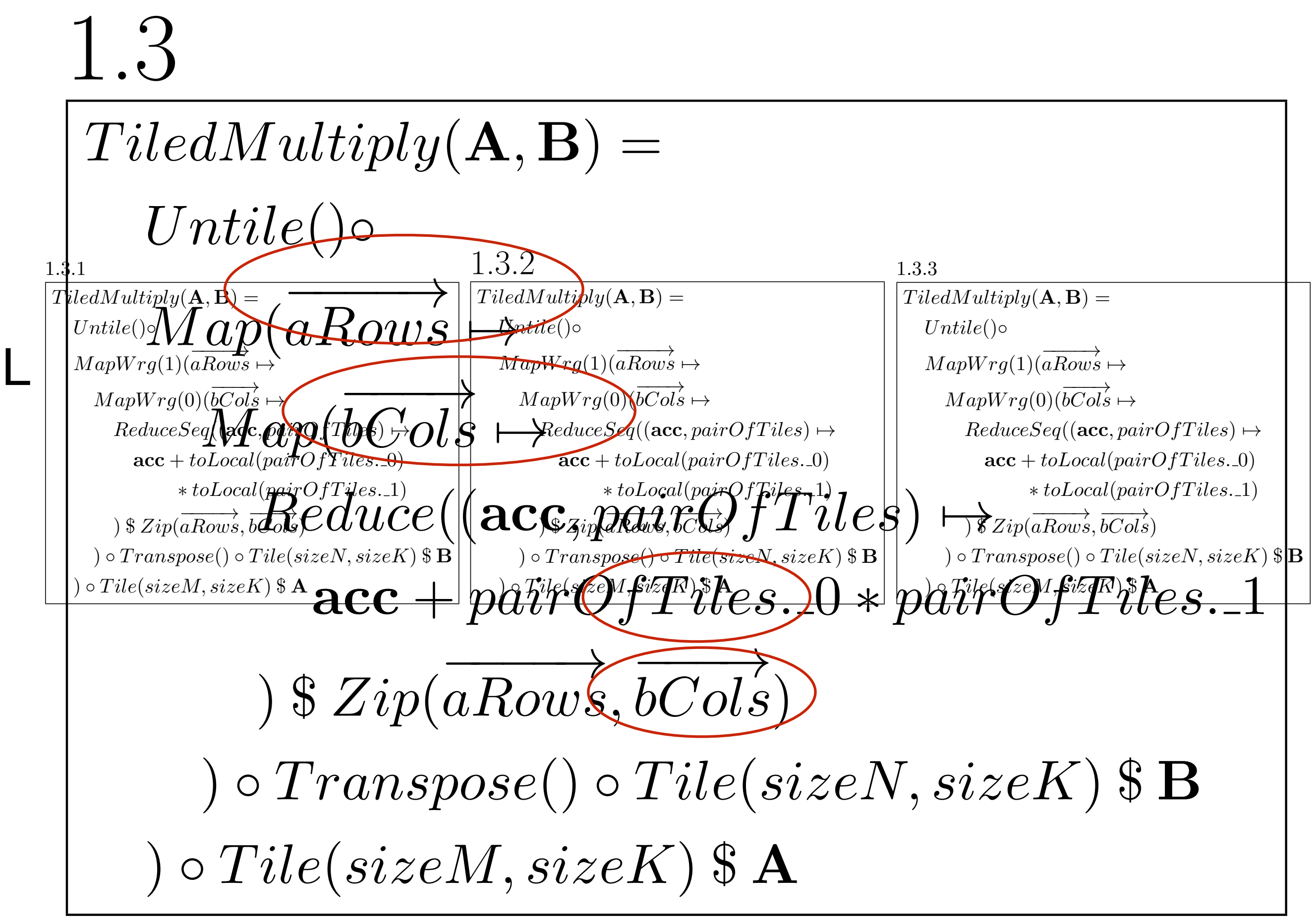
$$
\begin{aligned}
&TiledMultiply(\mathbf{A}, \mathbf{B}) = \\
&\quad Untile() \circ \\
&\quad Map(\overrightarrow{aRows} \mapsto \\
&\qquad Map(\overrightarrow{bCols} \mapsto \\
&\qquad\quad Reduce((\mathbf{acc}, pairOfTiles) \mapsto \\
&\qquad\qquad \mathbf{acc} + pairOfTiles.\_0 * pairOfTiles.\_1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

1.4

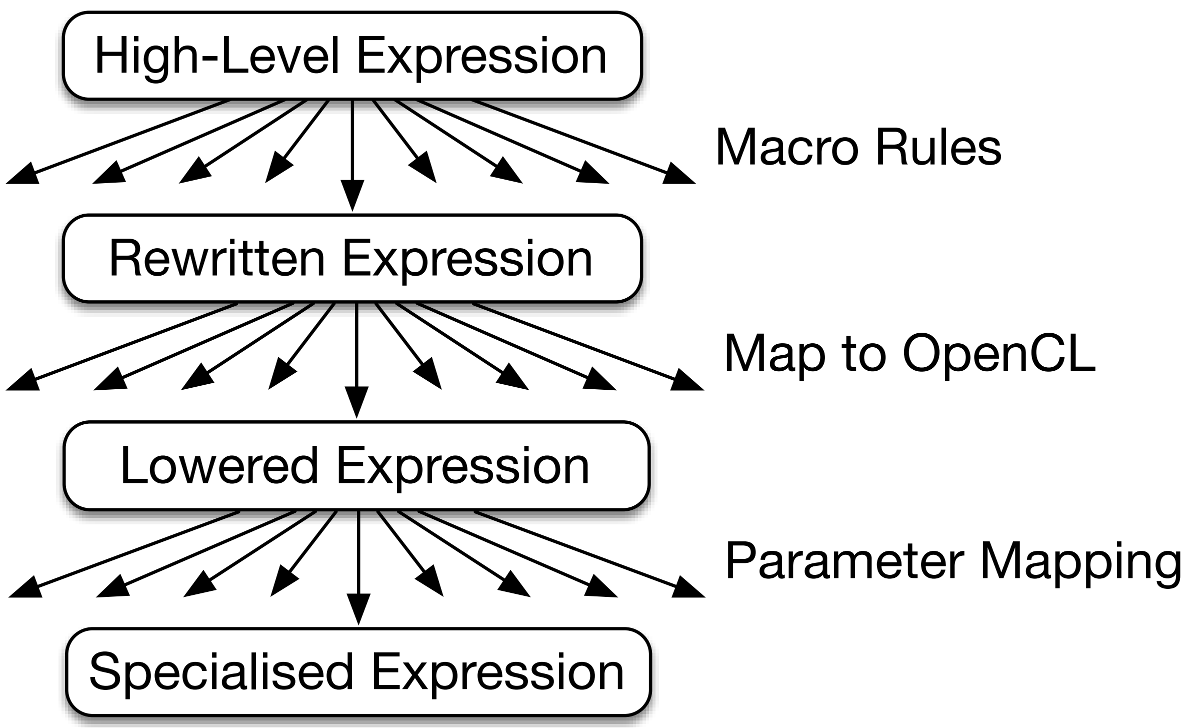
$$
\begin{aligned}
&BlockedMultiply(\mathbf{A}, \mathbf{B}) = \\
&\quad Join() \circ Map(Transpose()) \circ \\
&\quad Map(\overrightarrow{rowsA} \mapsto \\
&\qquad Map(\overrightarrow{colB} \mapsto \\
&\qquad\quad Transpose() \circ \\
&\qquad\quad Reduce(((\overrightarrow{acc}, rowElemPair) \mapsto \\
&\qquad\quad Map(p \mapsto p.\_0 + p.\_1 * rowElemPair.\_1) \ \$ \\
&\qquad\quad Zip(\overrightarrow{acc}, rowElemPair.\_0) \\
&\qquad\quad) \ \$ \ Zip(Transpose() \ \$ \ \overrightarrow{rowsA}, \overrightarrow{colB}) \\
&\qquad) \circ Transpose() \ \$ \ \mathbf{B} \\
&\quad) \circ Split(blockFactor) \ \$ \ \mathbf{A}
\end{aligned}
$$

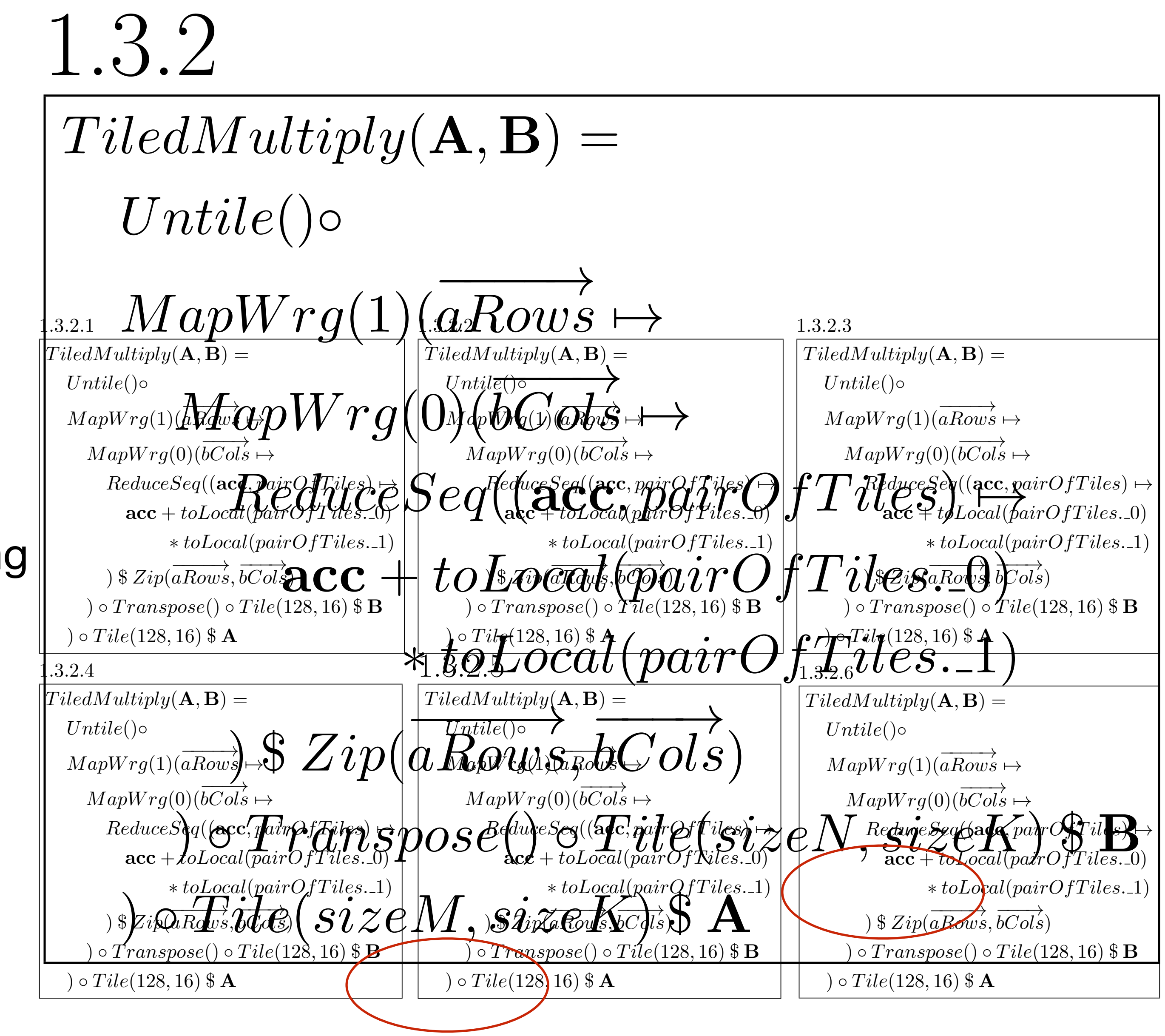
Exploration Strategy



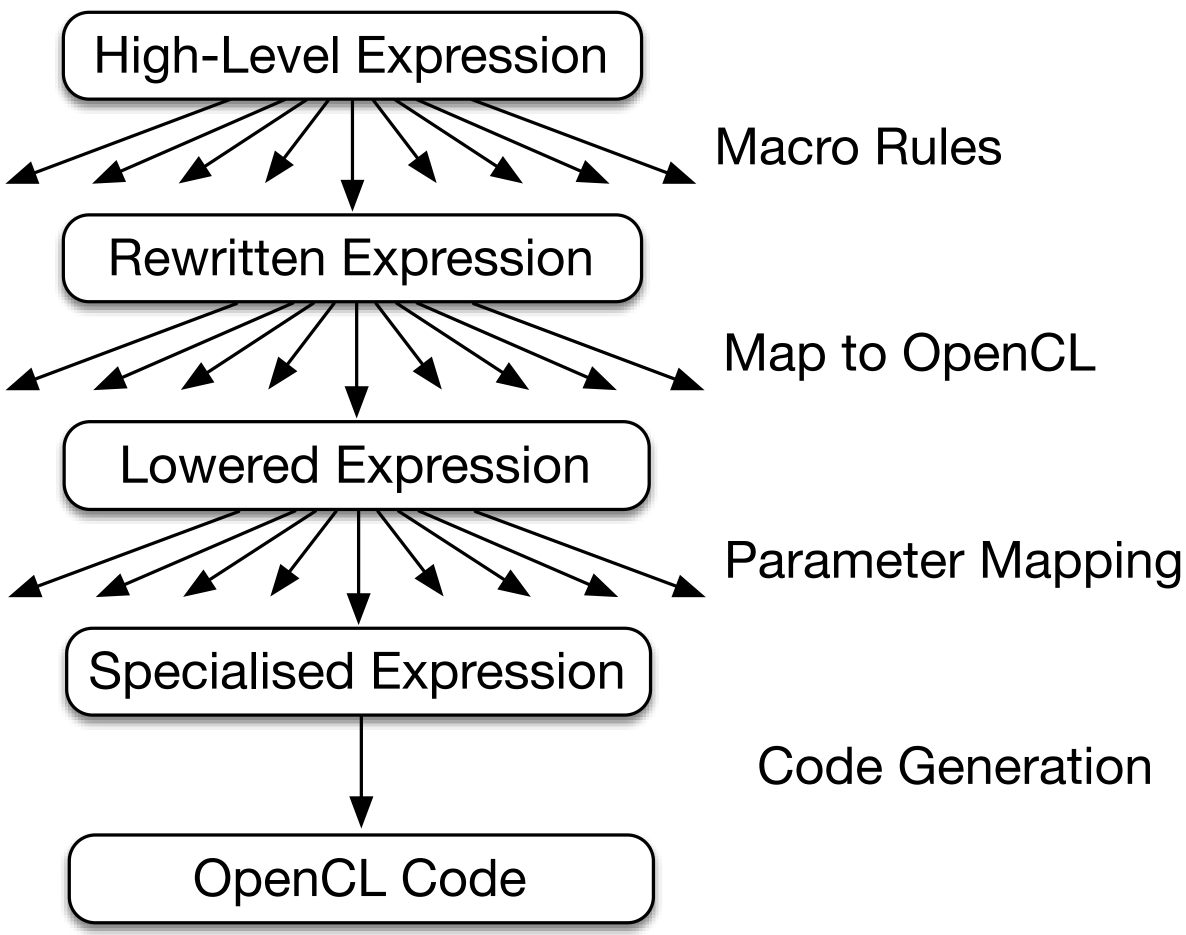


Exploration Strategy





Exploration Strategy



1.3.2.5

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

```
kernel mm_amd_opt(global float * A, B, C,
                  int K, M, N) {
local float tileA [512]; tileB [512];

private float acc_0; ...; acc_31;
private float blockOfB_0; ...; blockOfB_3;
private float blockOfA_0; ...; blockOfA_7;

int lid0 = local_id(0); lid1 = local_id(1);
int wid0 = group_id(0); wid1 = group_id(1);

for (int w1=wid1; w1<M/64; w1+=num_grps(1)) {
 for (int w0=wid0; w0<N/64; w0+=num_grps(0)) {

  acc_0 = 0.0f; ...; acc_31 = 0.0f;
  for (int i=0; i<K/8; i++) {
   vstore4(vload4(lid1*M/4+2*i*M+16*w1+lid0,A), 16*lid1+lid0, tileA);
   vstore4(vload4(lid1*N/4+2*i*N+16*w0+lid0,B), 16*lid1+lid0, tileB);
   barrier (...) ;

   for (int j = 0; j<8; j++) {
    blockOfA_0 = tileA[0+64*j+lid1*8]; ...; blockOfA_7 = tileA[7+64*j+lid1*8];
    blockOfB_0 = tileB[0 +64*j+lid0]; ...; blockOfB_3 = tileB[48+64*j+lid0];

    acc_0  += blockOfA_0 * blockOfB_0; ...; acc_28 += blockOfA_7 * blockOfB_0;
    acc_1  += blockOfA_0 * blockOfB_1; ...; acc_29 += blockOfA_7 * blockOfB_1;
    acc_2  += blockOfA_0 * blockOfB_2; ...; acc_30 += blockOfA_7 * blockOfB_2;
    acc_3  += blockOfA_0 * blockOfB_3; ...; acc_31 += blockOfA_7 * blockOfB_3;
   }
   barrier (...) ;
  }

  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;
  C[16+8*lid1*N+64*w0+64*w1*N+0*N+lid0]=acc_1; ...; C[16+8*lid1*N+64*w0+64*w1*N+7*N+lid0]=acc_29;
  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;
  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;
} } }
```

Heuristics for Matrix Multiplication

For Macro Rules:

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

For Map to OpenCL:

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

For Parameter Mapping:

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

Exploration in Numbers for Matrix Multiplication

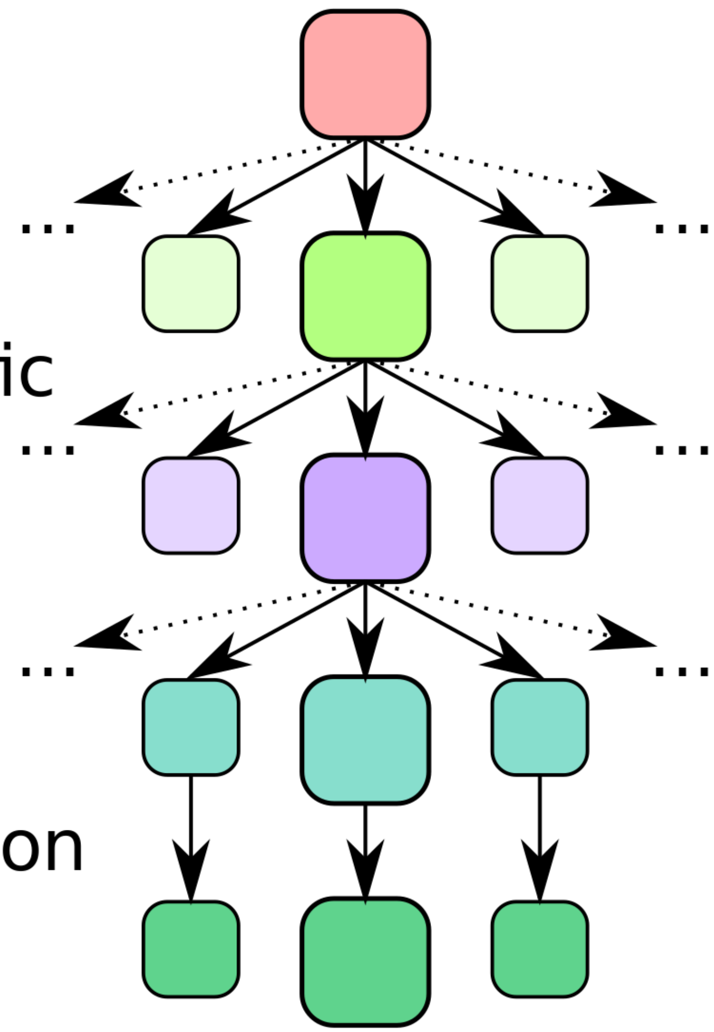
Phases:

Algorithmic Exploration

OpenCL specific Exploration

Parameter Exploration

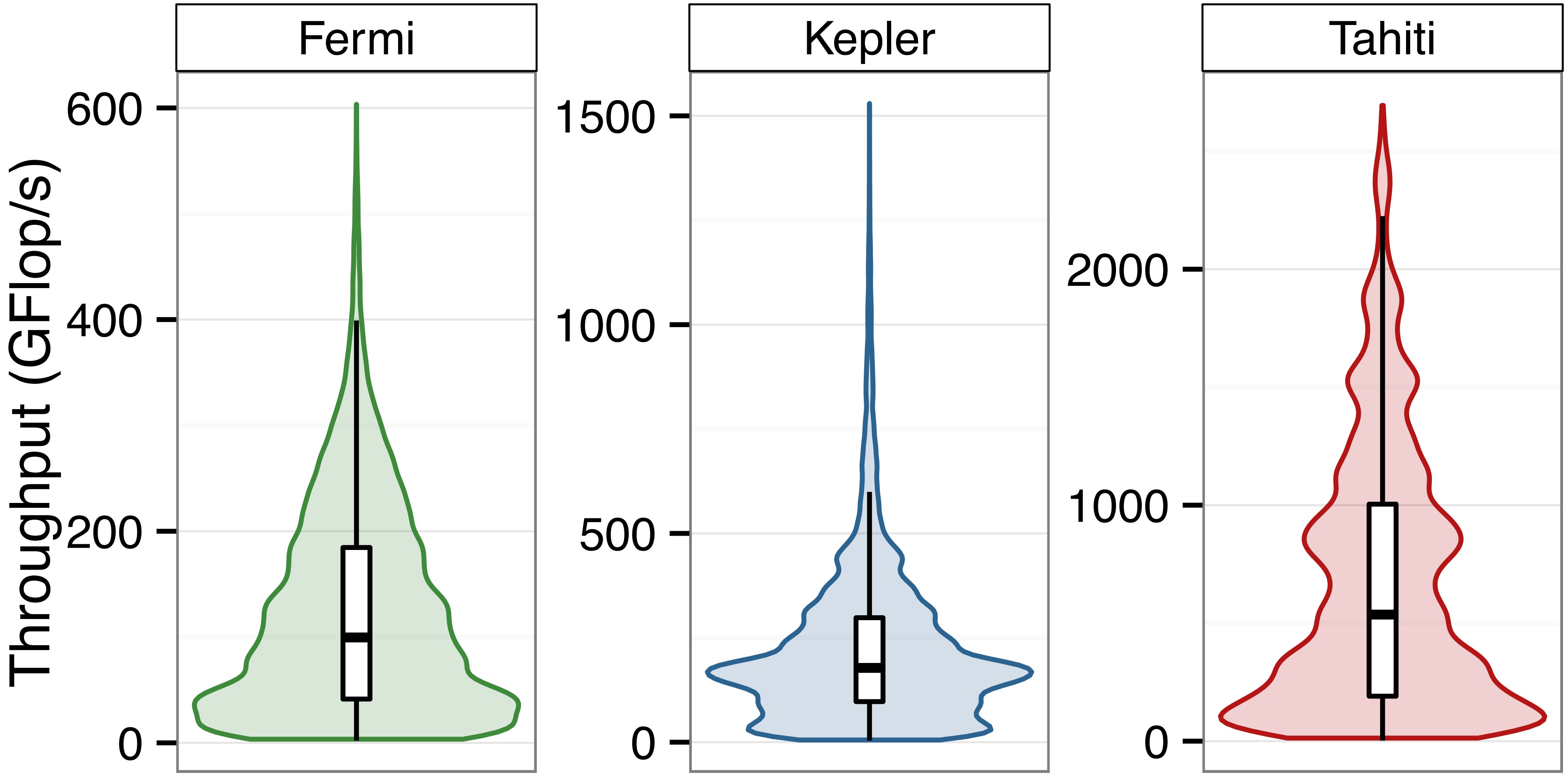
Code Generation



Program Variants:

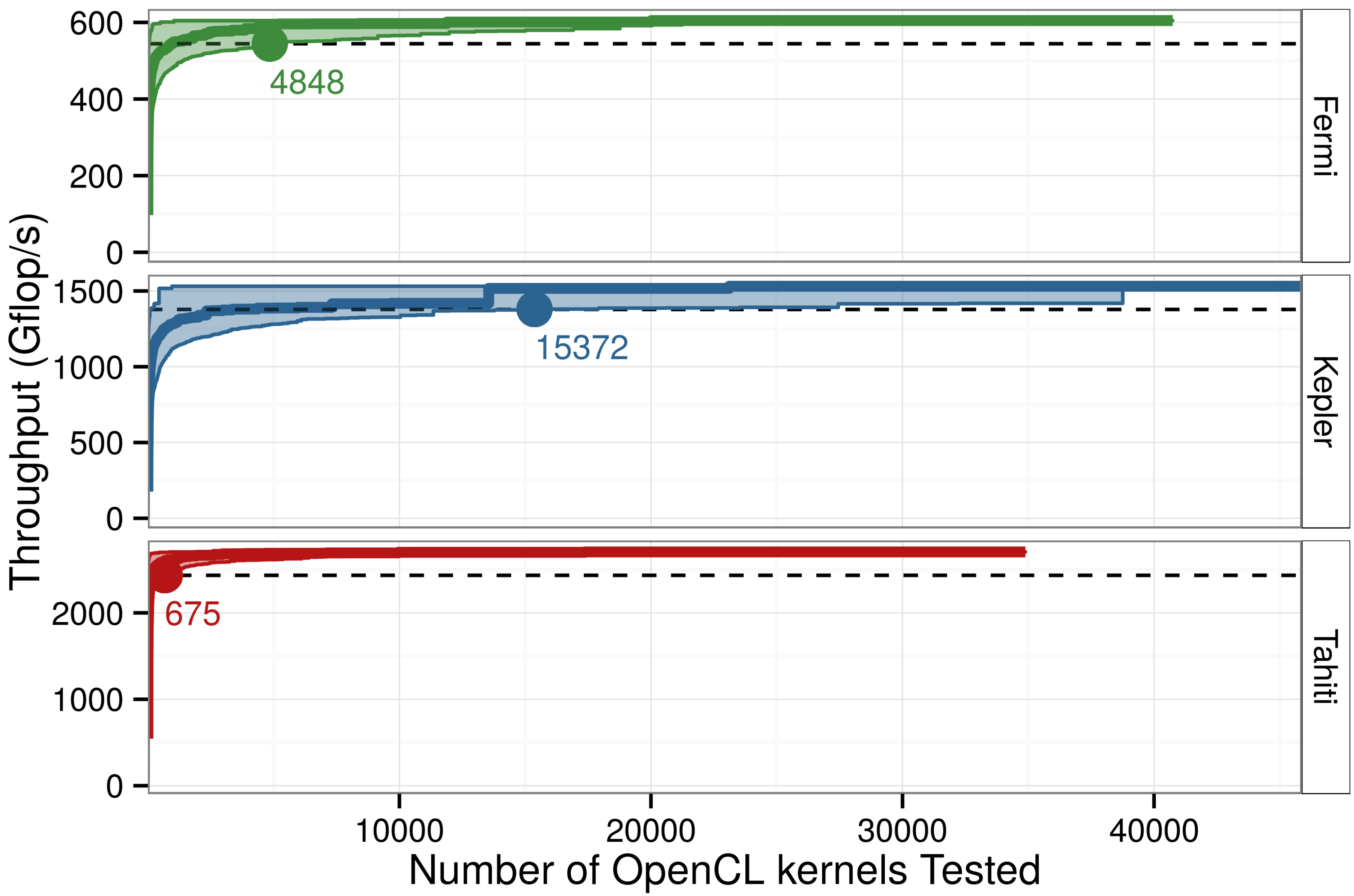
High-Level Program	1
Algorithmic Rewritten Program	8
OpenCL Specific Program	760
Fully Specialized Program	46,000
OpenCL Code	46,000

Exploration Space for Matrix Multiplication



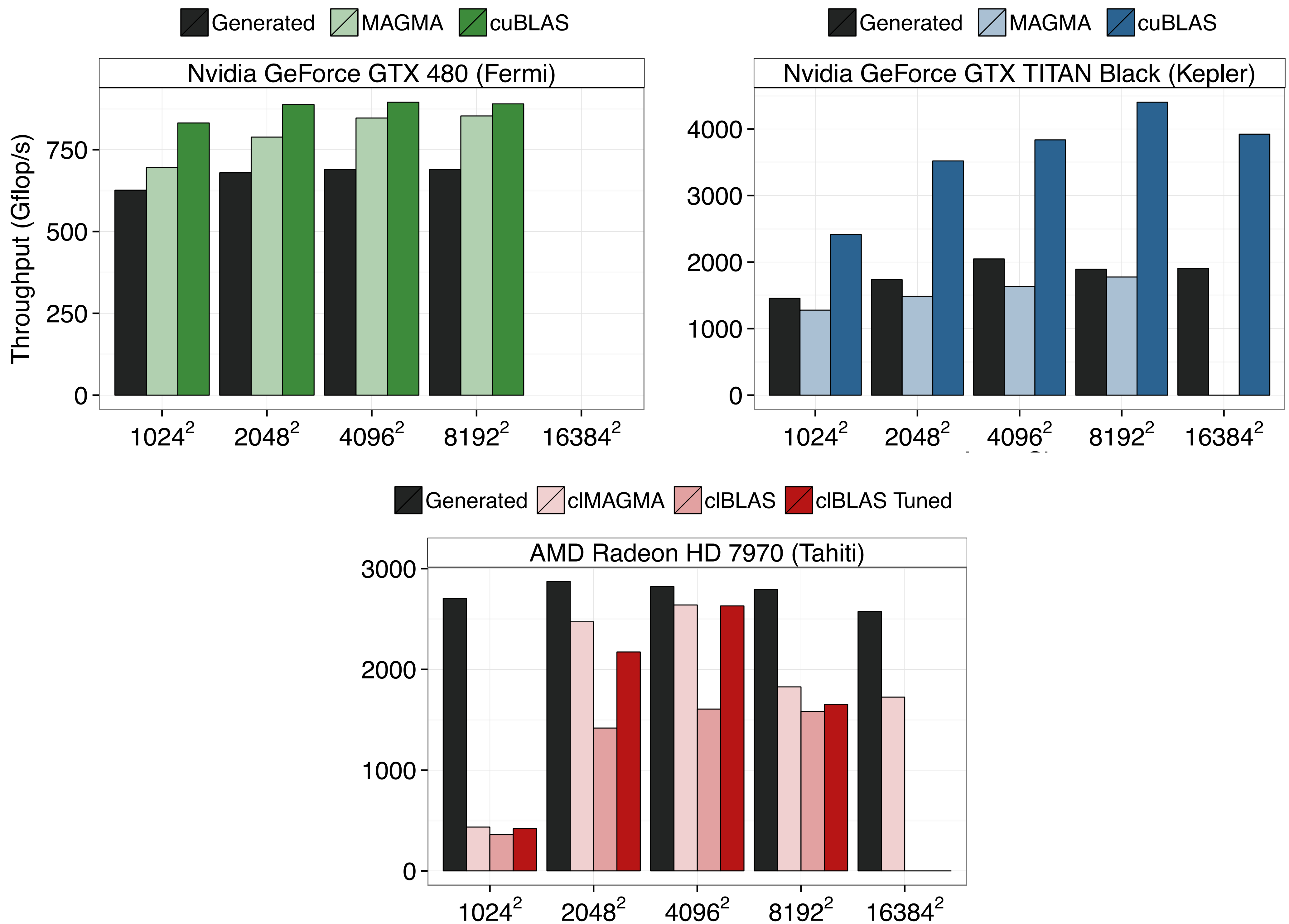
Only few OpenCL kernel with very good performance

Performance Evolution for Randomised Search



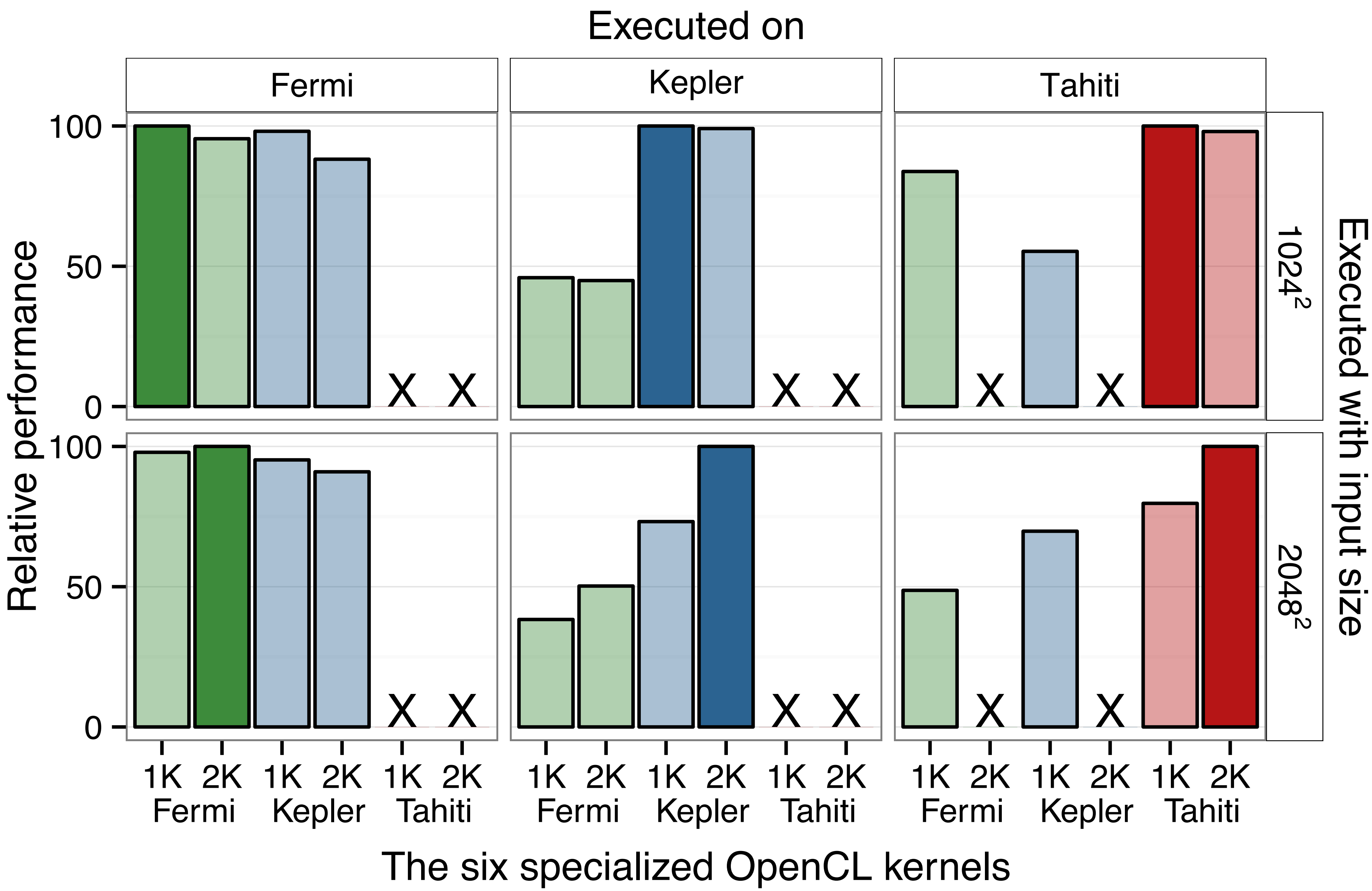
Even with a simple random search strategy one can expect to find a good performing kernel quickly

Performance Results Matrix Multiplication



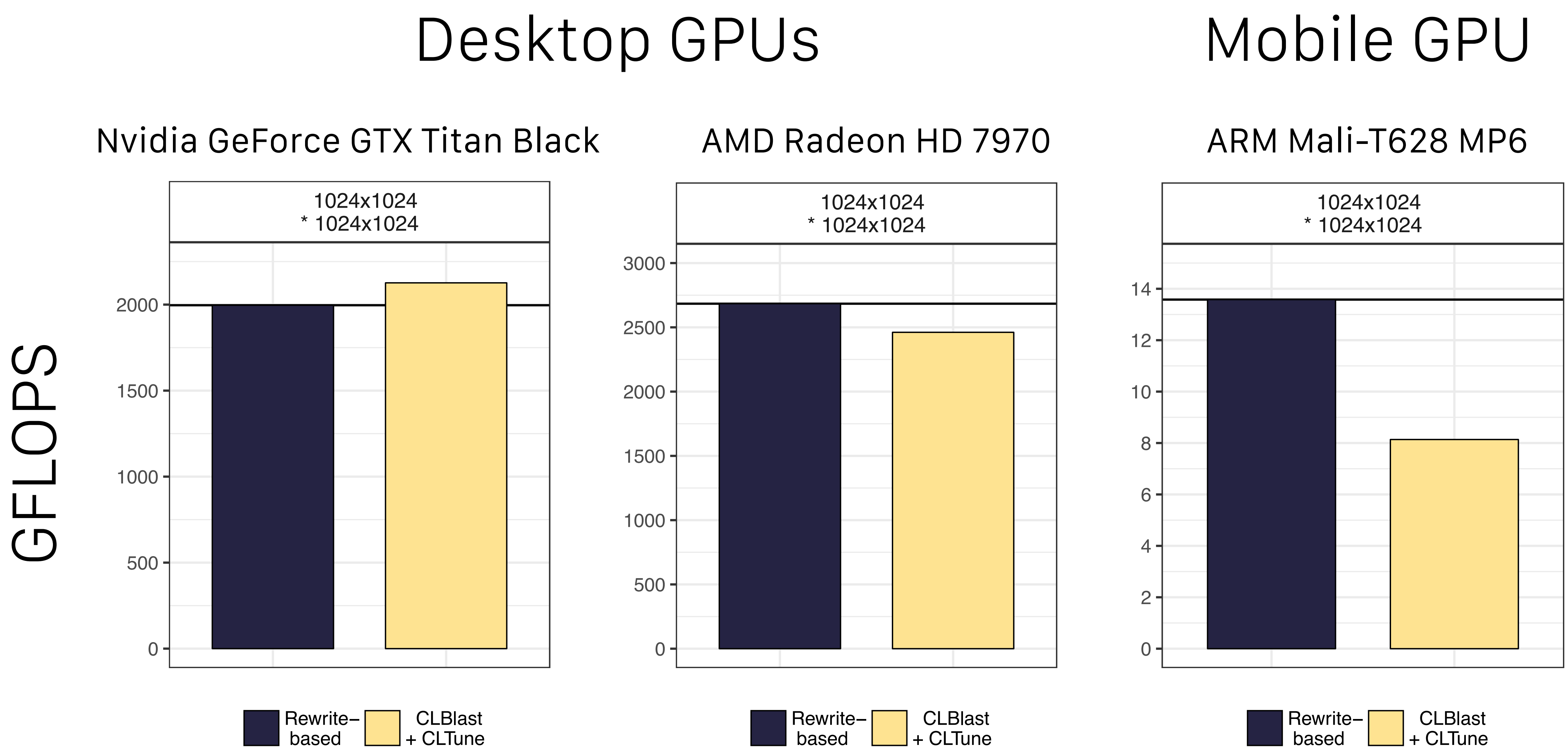
Performance close or better than hand-tuned MAGMA library

Performance Portability Matrix Multiplication



Generated kernels are specialised for device and input size

Desktop GPUs vs. Mobile GPU

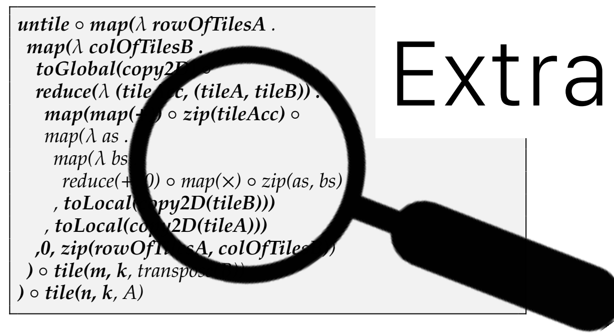


Performance portable even for mobile GPU device!

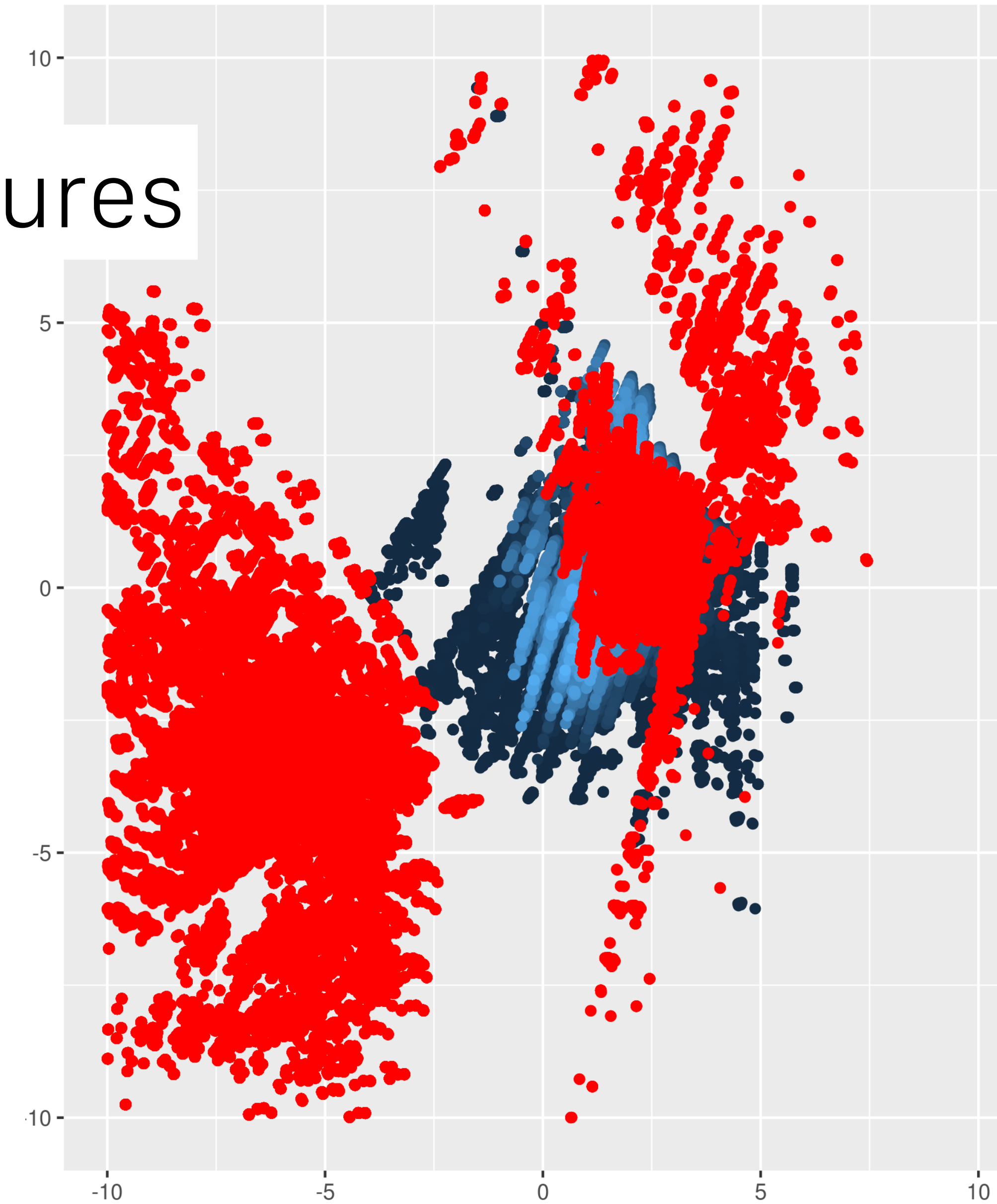
The LIFT Team

Performance Modeling of LIFT Programs

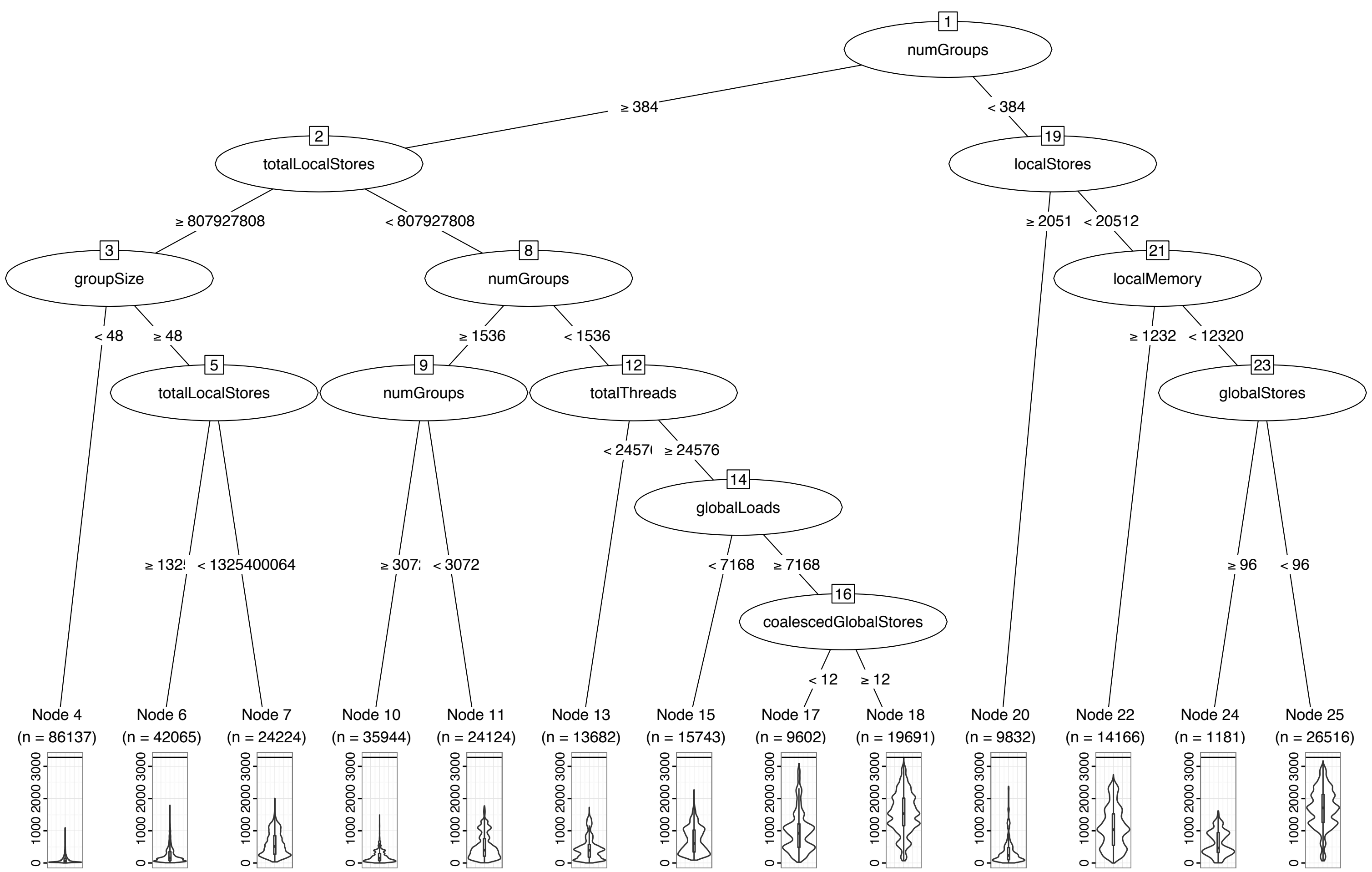
Toomas Remmelg
PhD Student
University of Edinburgh



Extract Features



Performance Model



Predictions
used to drive the
rewrite process

Adam Harries
PhD Student
University of Edinburgh

Graph Algorithms via Sparse Linear Algebra in LIFT

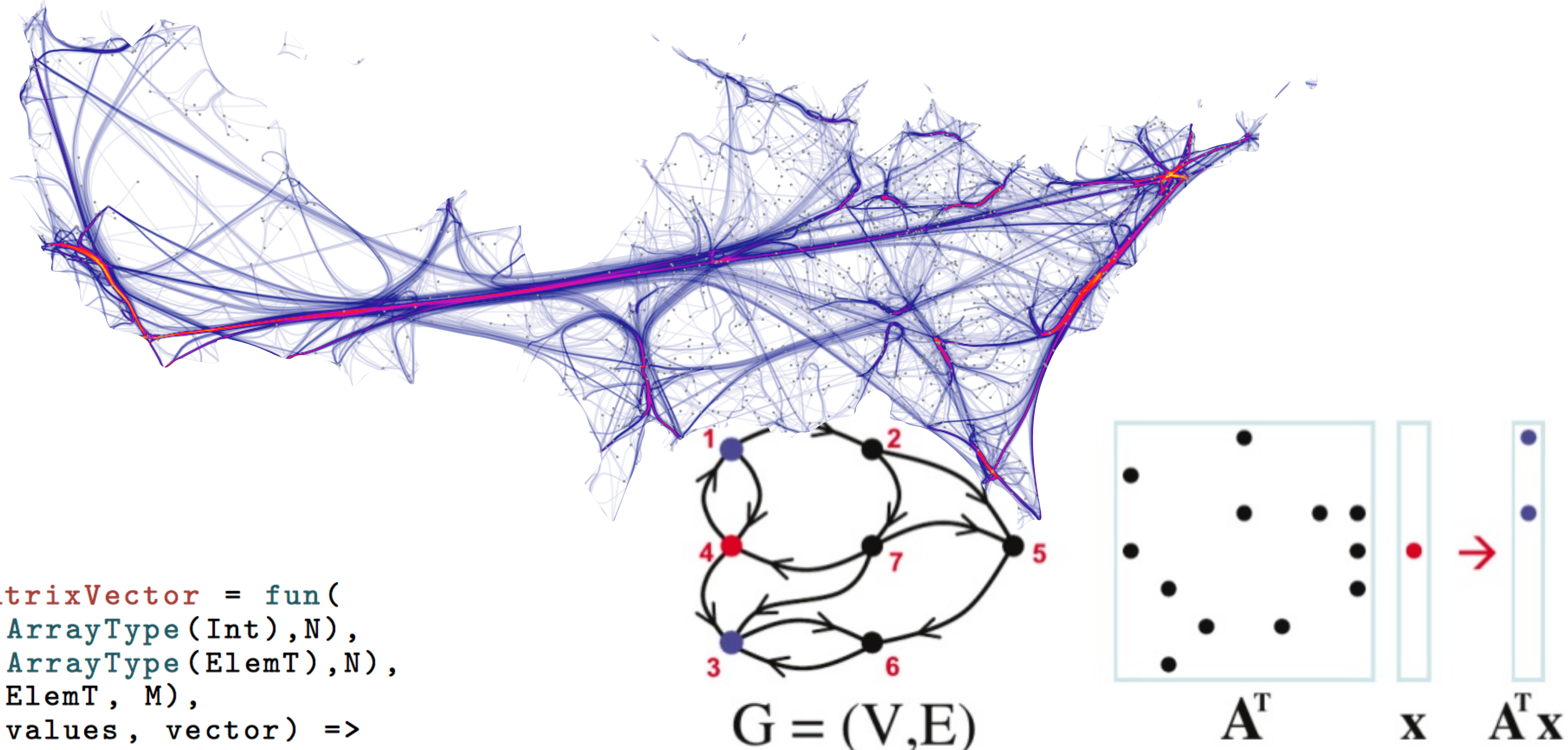
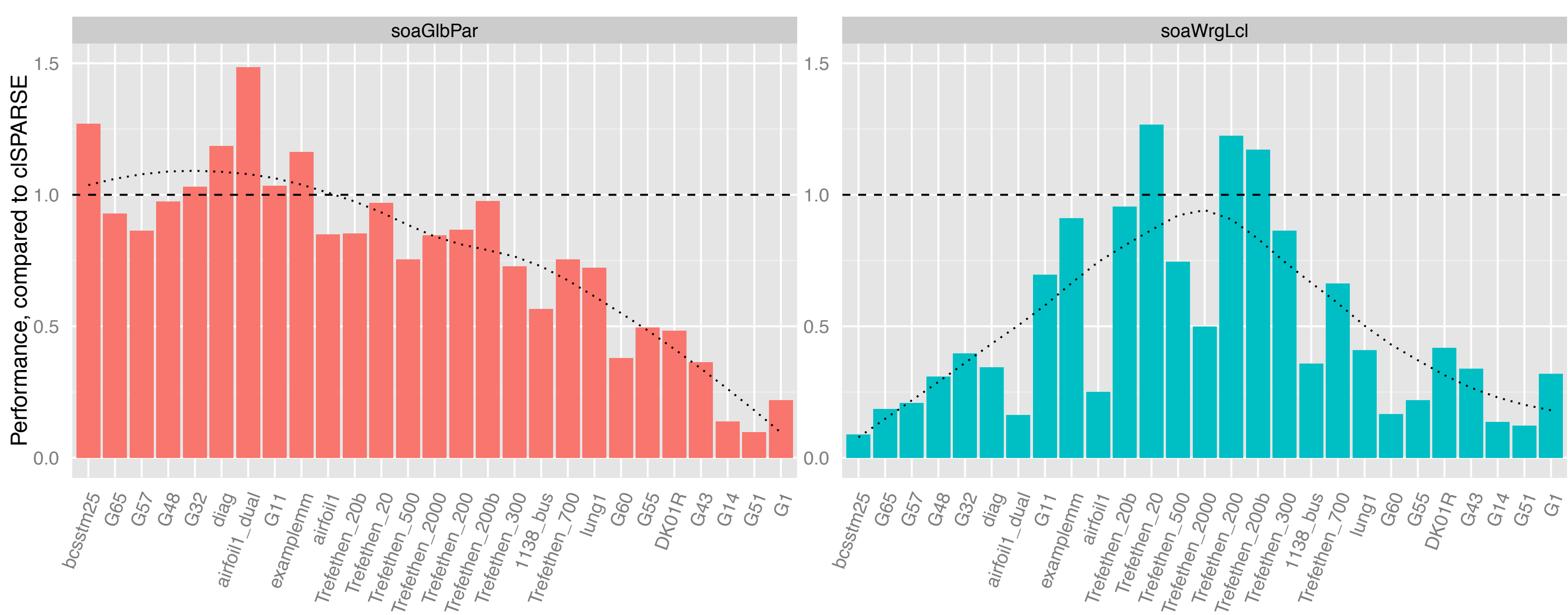


Image credit: [Kepner2011]

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



Differently optimised kernels for different inputs

Identify *hidden parallelism* in LIFT programs

Frederico Pizzuti
PhD Student
University of Edinburgh

Parallelising non-associative reductions

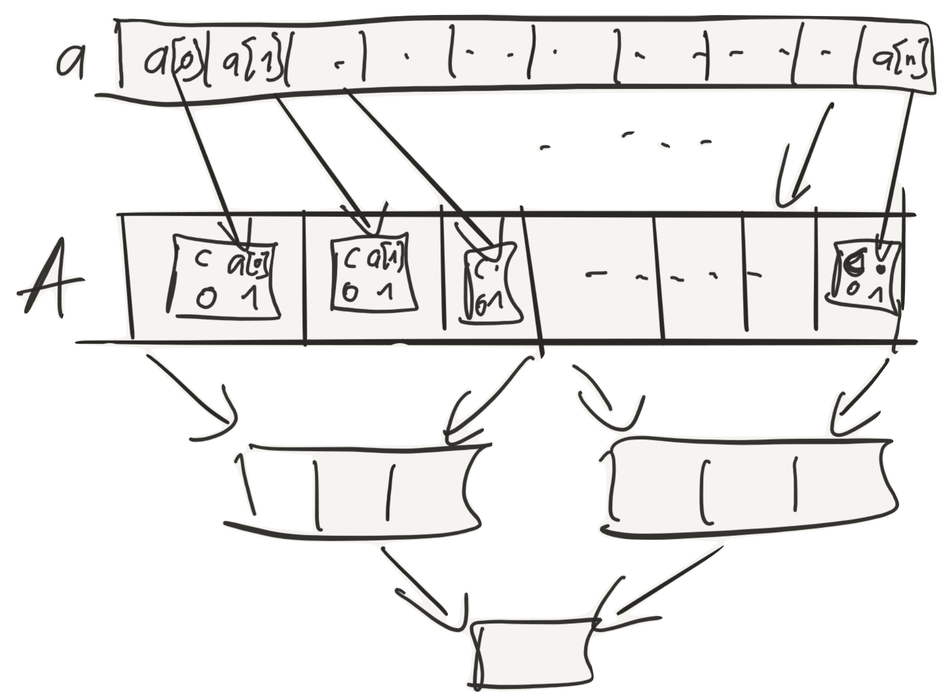
$$x \leftarrow 0;\ \textbf{for } i = 0 \textbf{ to } n \textbf{ do } x \leftarrow c \cdot x + a[i] \textbf{ done}.$$

$$\downarrow$$

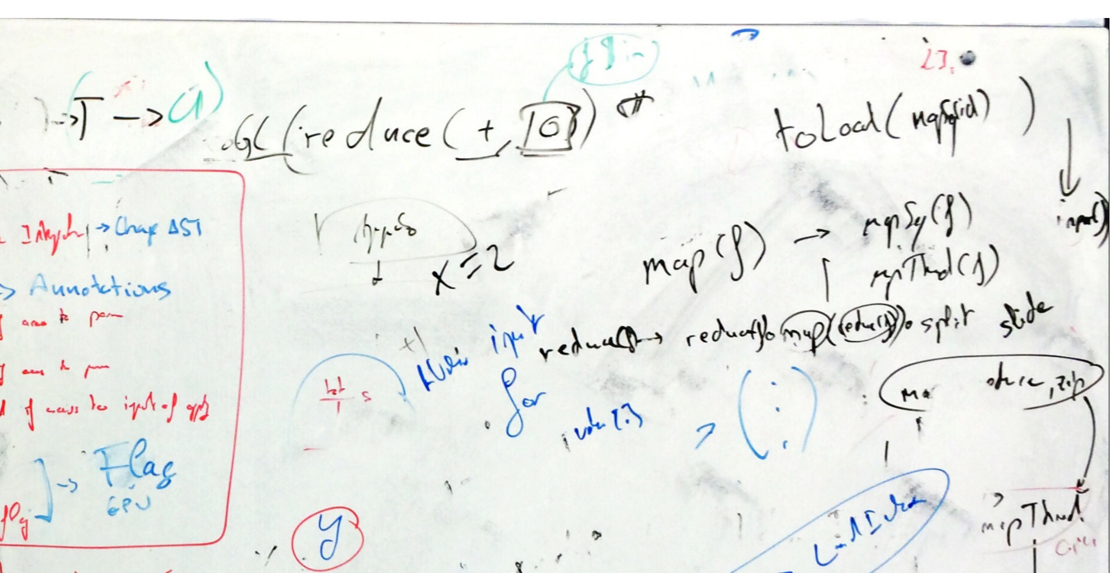
$$\boldsymbol{x} \leftarrow \boldsymbol{x}_0;\ \textbf{for } i = 0 \textbf{ to } n \textbf{ do } \boldsymbol{x} \leftarrow A_i \times \boldsymbol{x} \textbf{ done},$$

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

$$\downarrow$$

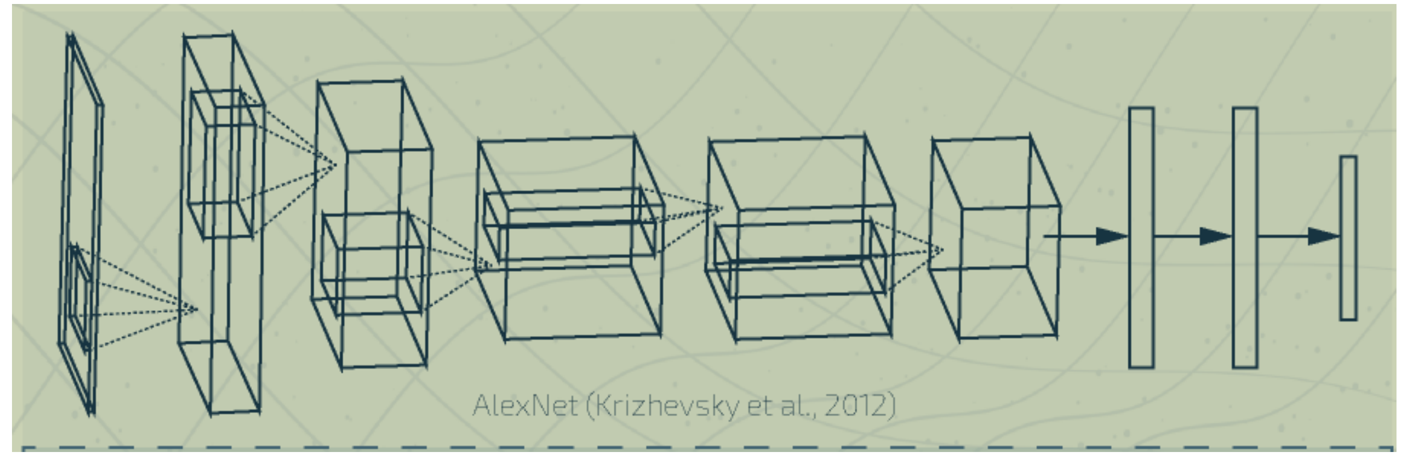


Key idea: Rearrange data as matrices to exploit associative matrix multiplication



Naums Mogers
PhD Student
University of Edinburgh

Optimizing Deep Learning with LIFT



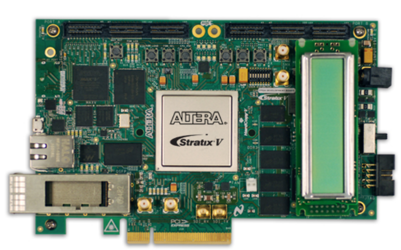
AlexNet (Krizhevsky et al., 2012)

Express layers with LIFT primitives

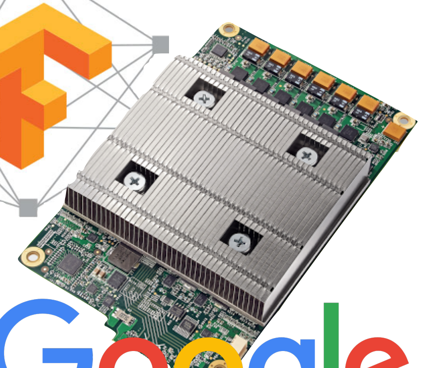
```
fully_connected(f, weights, bias, inputs) :=
    Map((neuron_weights, neuron_bias) → f() o Reduce(add, neuron_bias) o
    Map(mult) $ Zip(inputs, neuron_weights)) $ Zip(weights, bias)
```

Optimize individual layers and across layers via rewrites

FPGAs

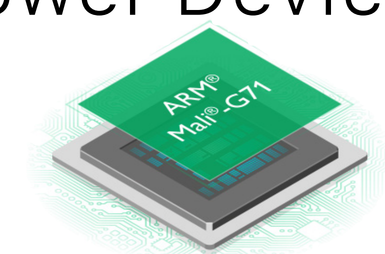


Microsoft Research



Google TPU

Low Power Devices



Stencil Computations in LIFT

Bastian Hagedorn
PhD Student
University of Münster

Larisa Stoltzfus
PhD Student
University of Edinburgh

Express Stencil with Skeletons

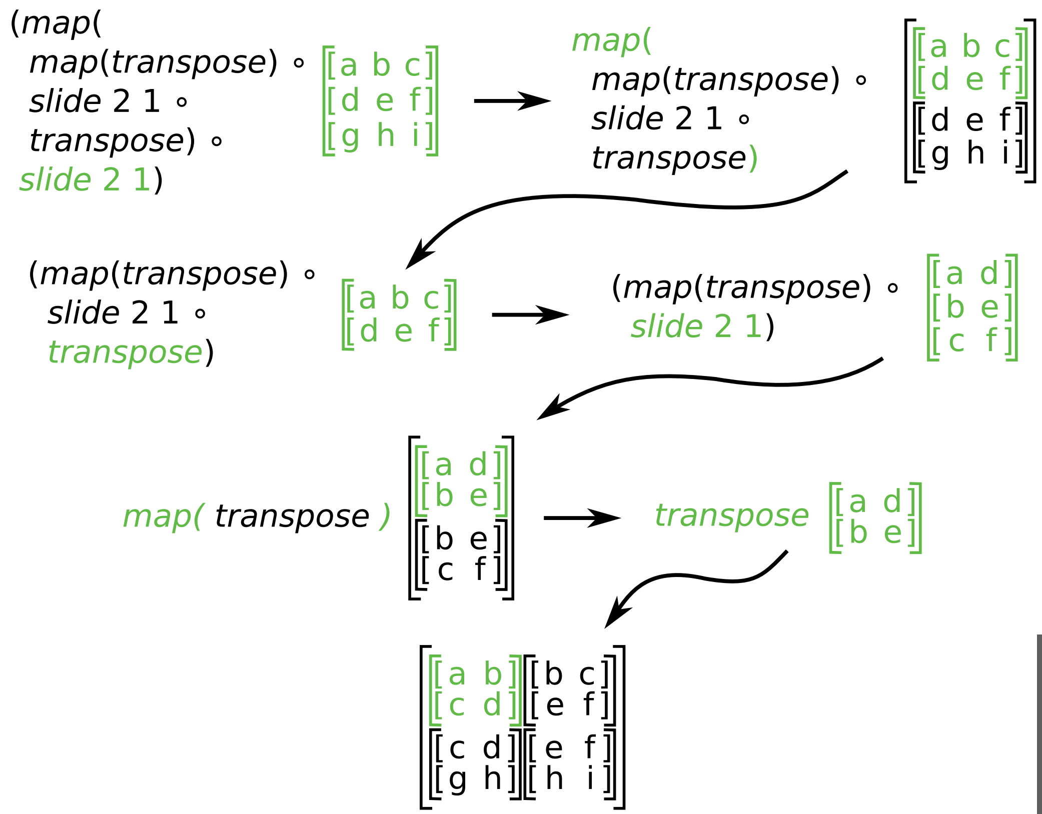
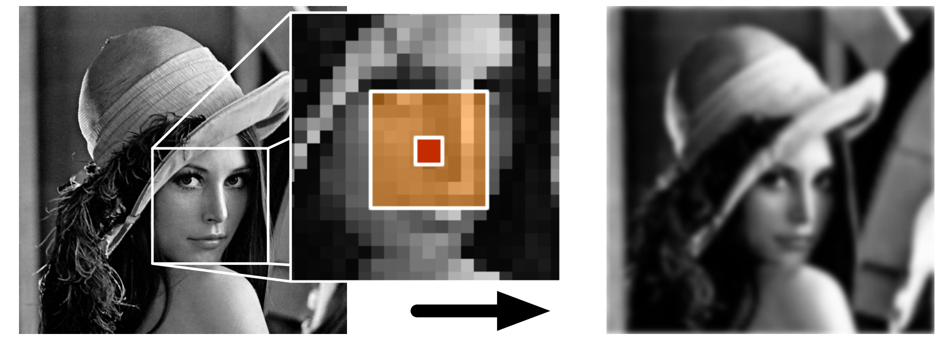
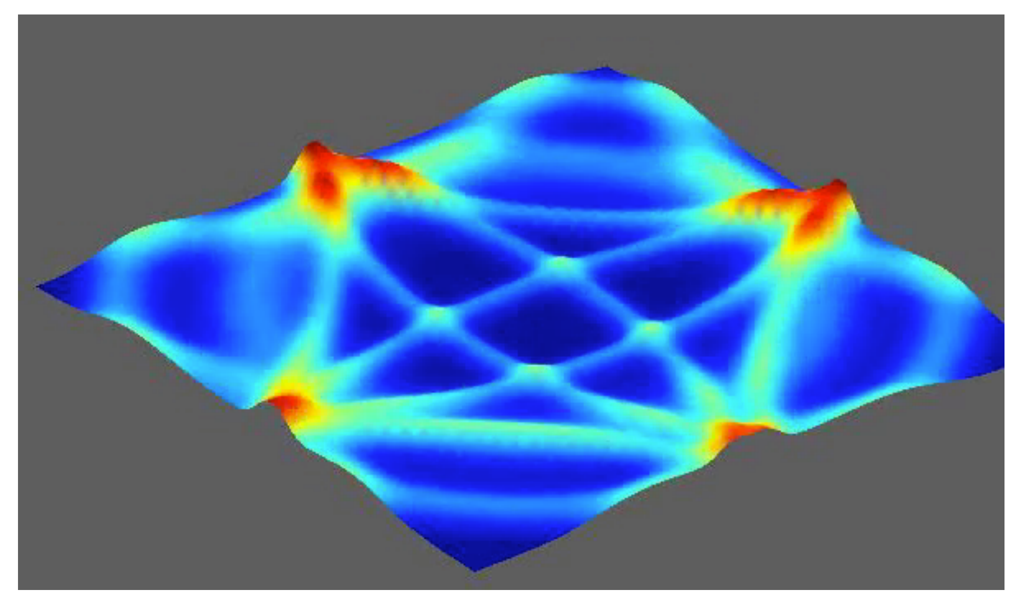


Image Processing

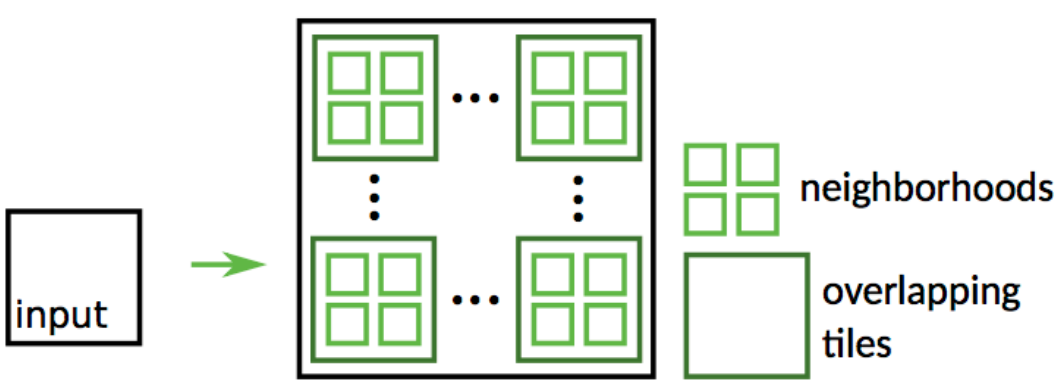


Acoustics Simulation



Video

Explore optimisations as rewrites



Data Parallel Idealised Algol: A New Foundation for LIFT

Formalisation of
strategy preserving translation
of functional into imperative code

Functional

$\textbf{reduce}\ (+)\ 0\ (\textbf{map}\ (\lambda x.\ \textbf{fst}\ x * \textbf{snd}\ x)\ (\textbf{zip}\ xs\ ys))$

$\textbf{asScalar}_4\ (\textbf{join}\ (\textbf{mapWorkgroup}\ (\lambda zs_1.\ \textbf{mapLocal}\ (\lambda zs_2.\ \textbf{reduce}\ (\lambda x\ a.\ (\textbf{fst}\ x * \textbf{snd}\ x) + a)\ 0\ (\textbf{split}\ 8192\ zs_2))\ zs_1)$
$(\textbf{split}\ 8192\ (\textbf{zip}\ (\textbf{asVector}_4\ xs)\ (\textbf{asVector}_4\ ys))))))$

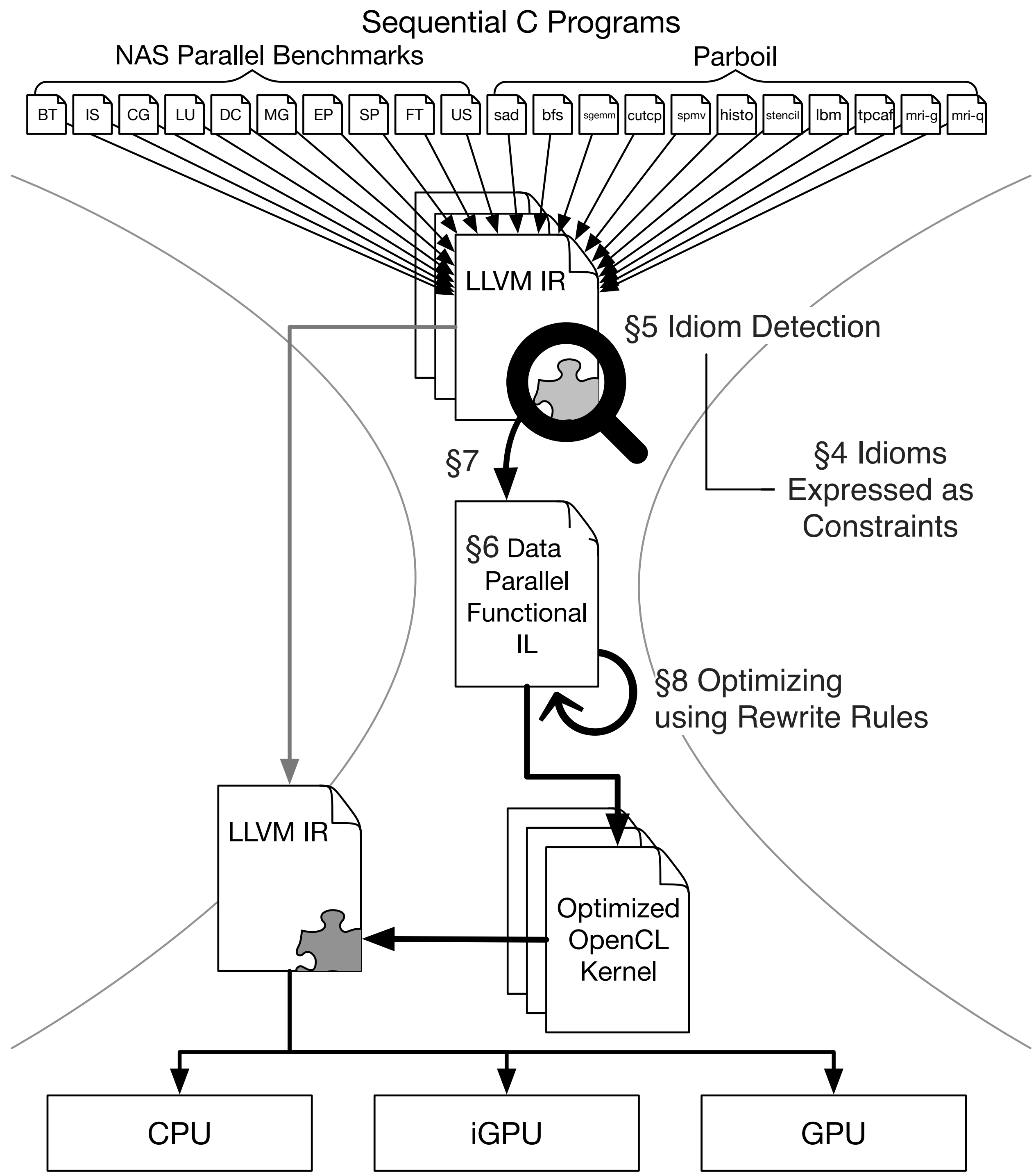
Imperative

$\textbf{parforWorkgroup}\ (N/8192)\ (\textbf{joinAcc}\ (N/8192)\ 64\ (\textbf{asScalarAcc}_4\ (N/128)\ out))\ (\lambda\ gid\ o.$
$\quad \textbf{parforLocal}\ 64\ o\ (\lambda\ lid\ o.$
$\quad\quad \textbf{newPrivate}\ \mathrm{num}\langle 4\rangle\ accum.$
$\quad\quad\quad accum.1 := 0;$
$\quad\quad\quad \textbf{for}\ 2048\ (\lambda\ i.$
$\quad\quad\quad\quad accum.1 := accum.2\ +$
$\quad\quad\quad\quad\quad (\textbf{fst}\ (\textbf{idx}\ (\textbf{idx}\ (\textbf{split}\ 2048\ (\textbf{idx}\ (\textbf{split}\ (8192 * 4)\ (\textbf{zip}\ (\textbf{asVector}_4\ xs)\ (\textbf{asVector}_4\ ys)))\ gid))\ lid)\ i)) *$
$\quad\quad\quad\quad\quad (\textbf{snd}\ (\textbf{idx}\ (\textbf{idx}\ (\textbf{split}\ 2048\ (\textbf{idx}\ (\textbf{split}\ (8192 * 4)\ (\textbf{zip}\ (\textbf{asVector}_4\ xs)\ (\textbf{asVector}_4\ ys)))\ gid))\ lid)\ i))\ );$
$\quad\quad\quad out := accum.2\ ))$

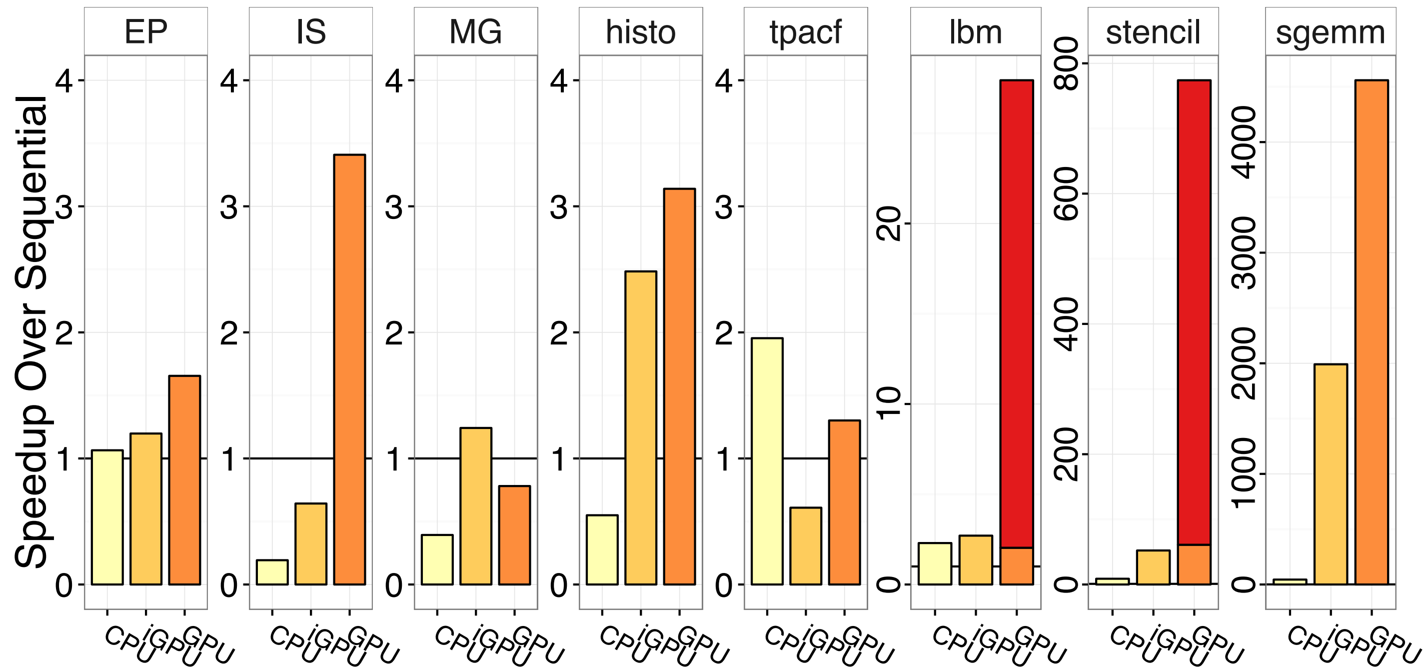
```
kernel void KERNEL(global float *out, const global float *restrict xs,
                   const global float *restrict ys, int N) {
  for (int g_id = get_group_id(0); g_id < N / 8192; g_id += get_num_groups(0)) {
    for (int l_id = get_local_id(0); l_id < 64; l_id += get_local_size(0)) {
      float4 accum;
      accum = (float4)(0.0, 0.0, 0.0, 0.0);
      for (int i = 0; i < 2048; i += 1) {
        accum = (accum +
                 (vload4(((2048 * l_id) + (8192 * 4 * g_id) + i), xs) *
                  vload4(((2048 * l_id) + (8192 * 4 * g_id) + i), ys))); }
      vstore4(accum, ((64 * g_id) + l_id), out); } } }
```

- Collaboration with Robert Atkey (Strathclyde), Christophe Dubach, and Sam Lindley (Edinburgh)
- More further down the line: formalisation of OpenCL and similar low-level models to enforce them via type- and effect-systems

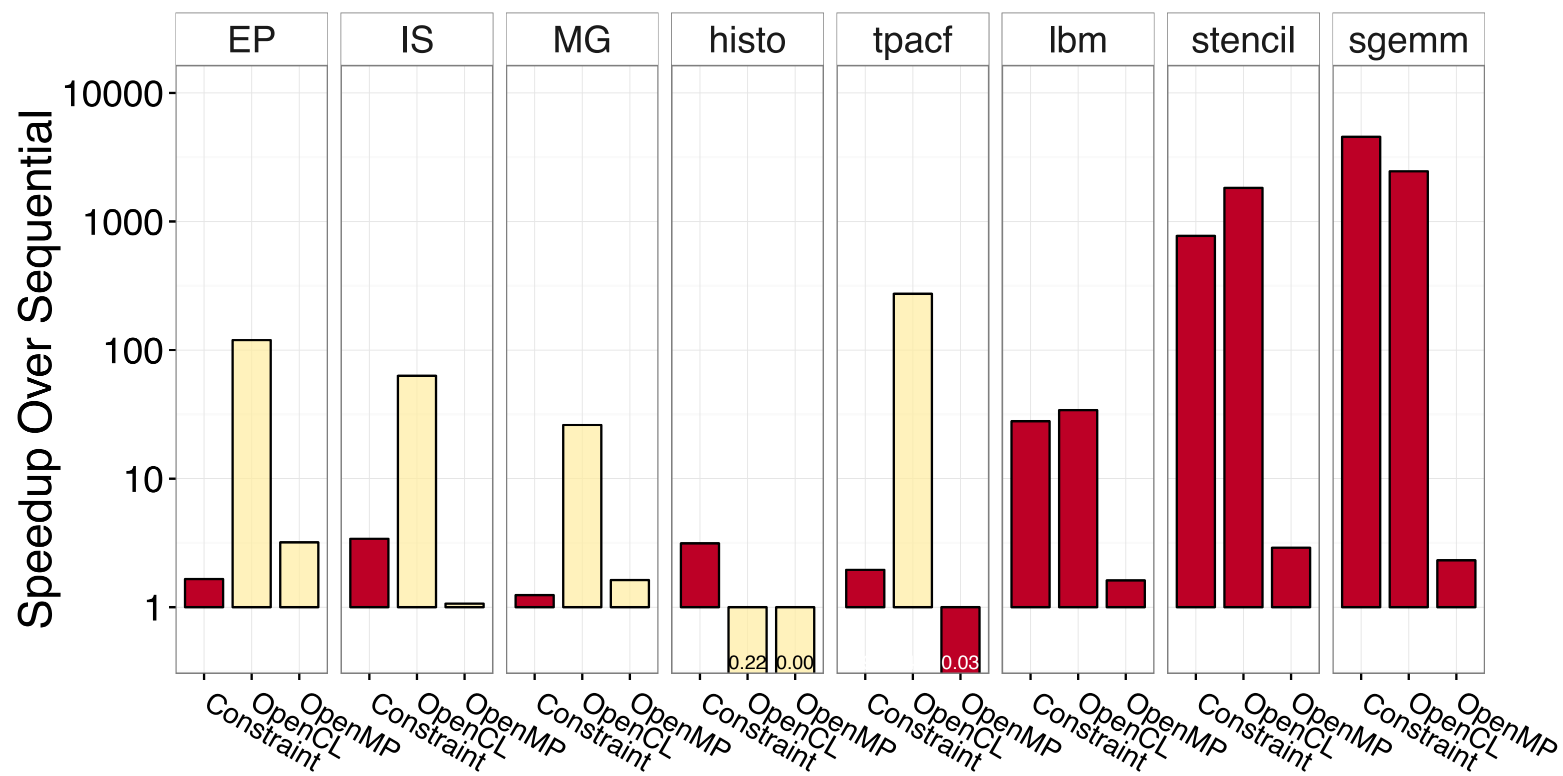
Application of *Lift* as a code generation backend



Preliminary results



Heterogeneous code generation gives a speedup in all cases

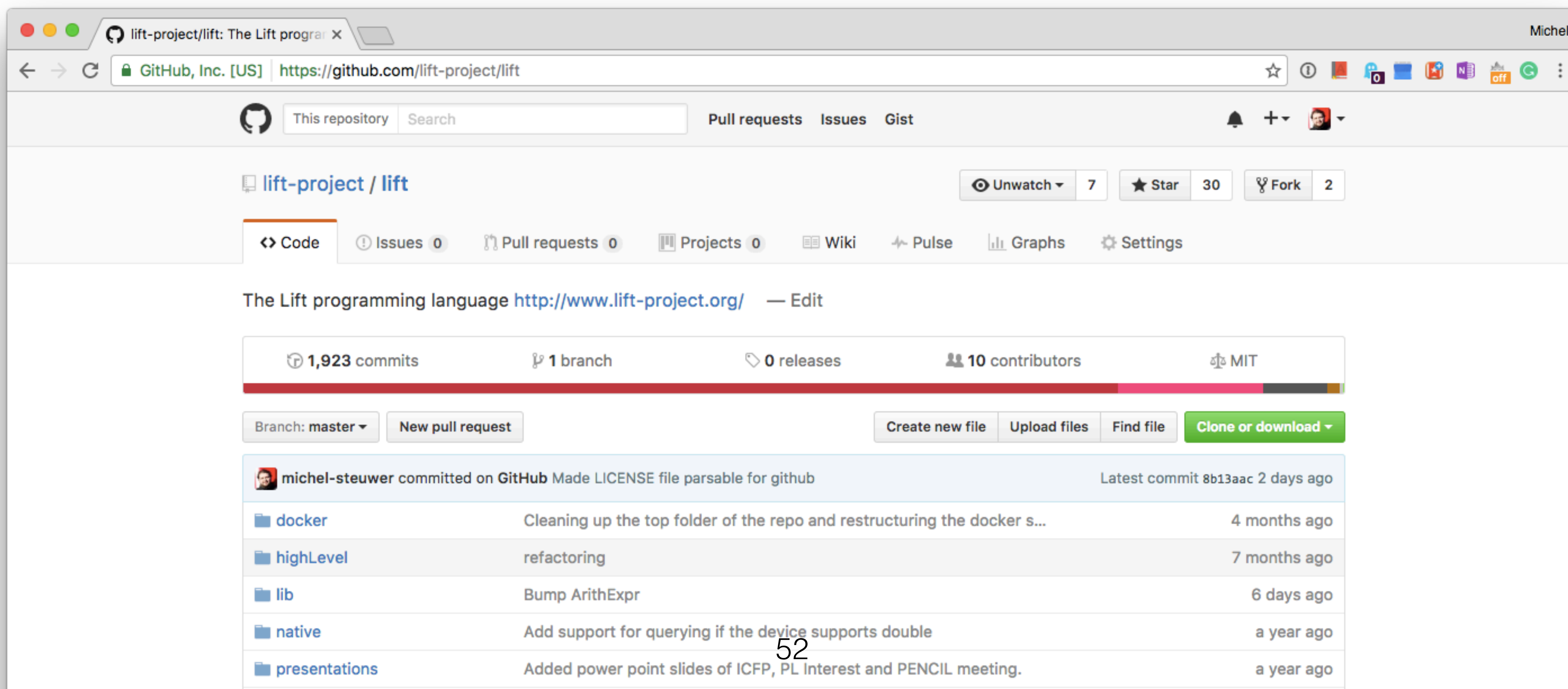


Performance close to manual written code —when parallelisation strategy is comparable

Lift is Open-Source Software

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

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



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