Generating Performance Portable OpenCL Code

From High-Level Functional Expressions

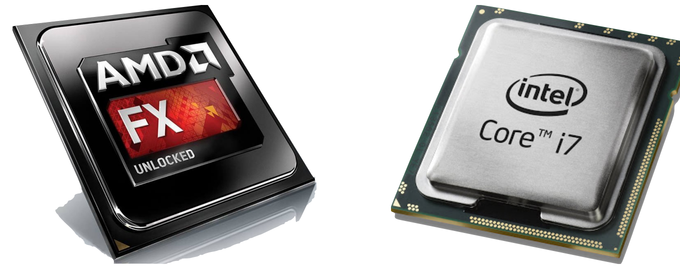
Michel Steuwer

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

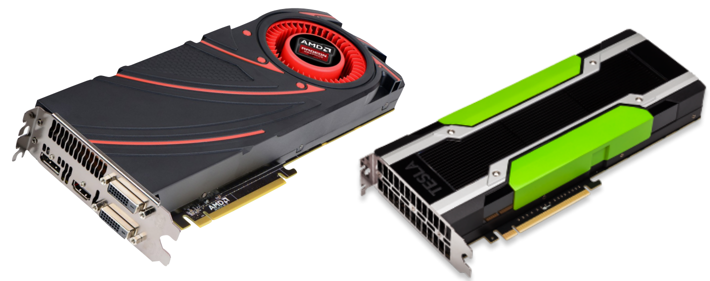
THE UNIVERSITY *of* EDINBURGH

The Problem(s)

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



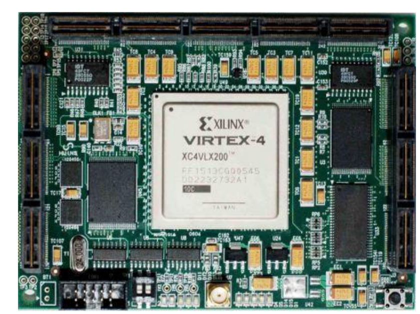
CPU



GPU

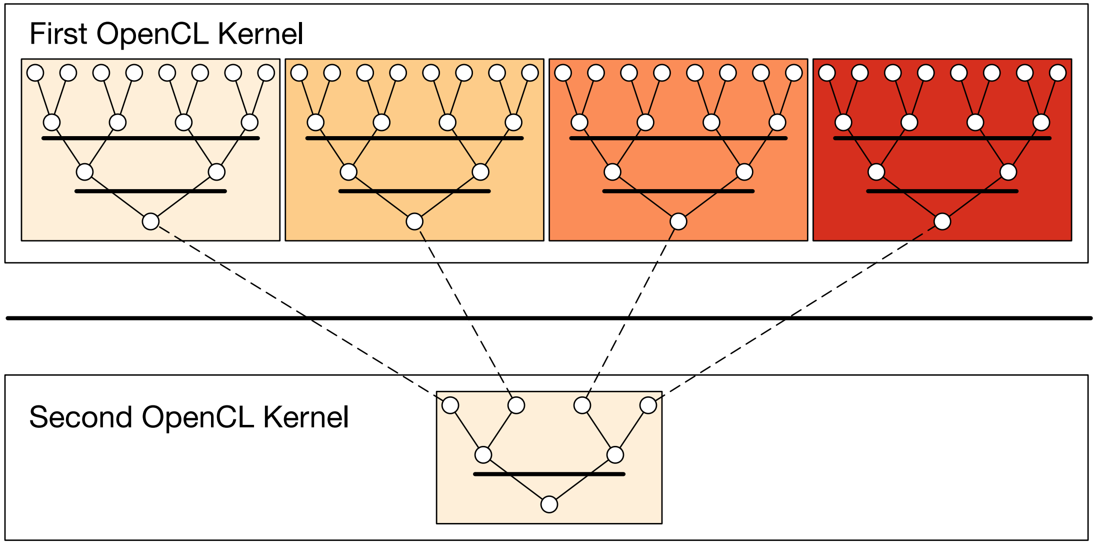
Accelerator

FPGA



Case Study: Parallel Reduction in OpenCL

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



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];
}
```

OpenCL Programming Model

```
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];
}
```

- Multiple *work-items* (threads) execute the same *kernel* function
- *Work-items* are organised for execution in *work-groups*

OpenCL Programming Model

```
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];
}
```

- Multiple *work-items* (threads) execute the same *kernel* function
- *Work-items* are organised for execution in *work-groups*

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];
}
```

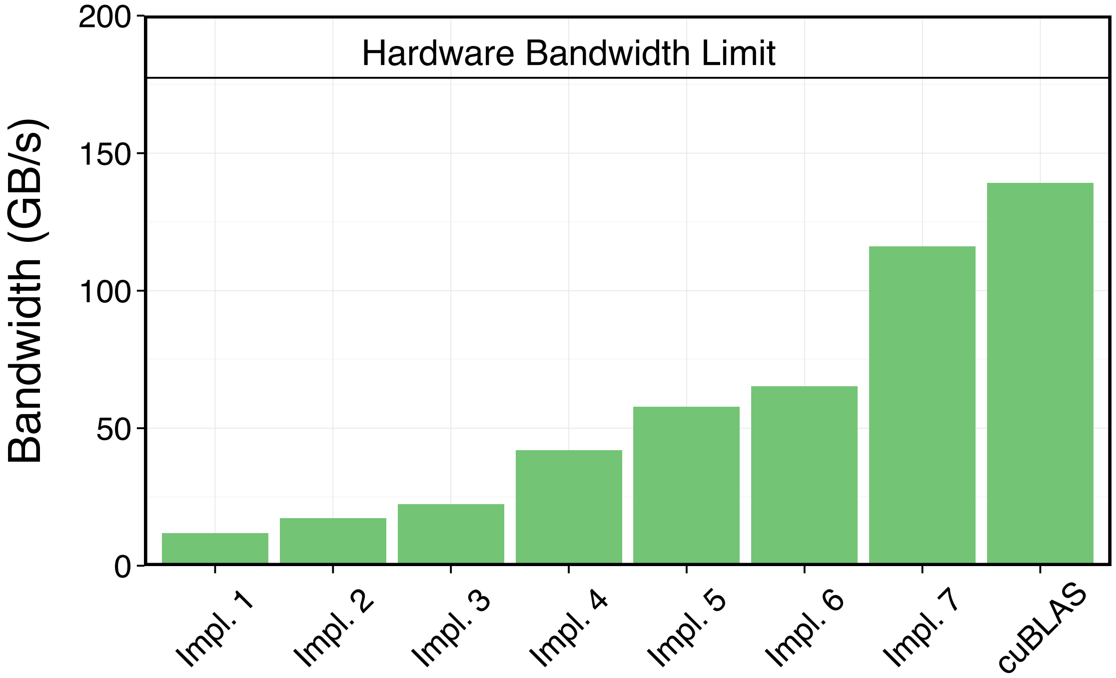
Unoptimized Implementation

```
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

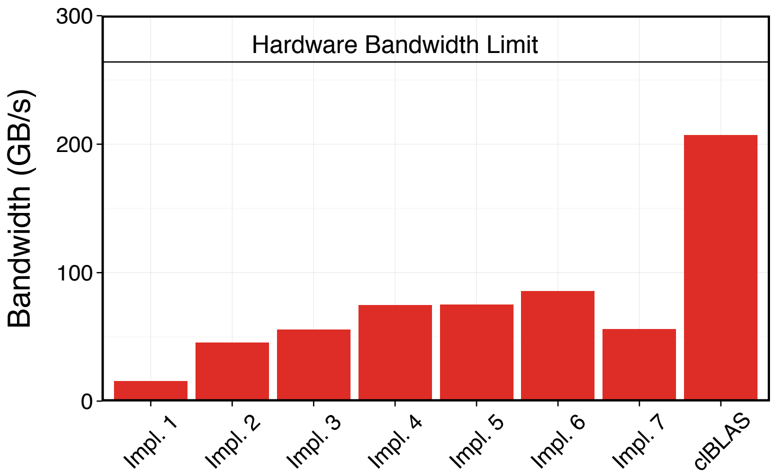
Performance Results Nvidia



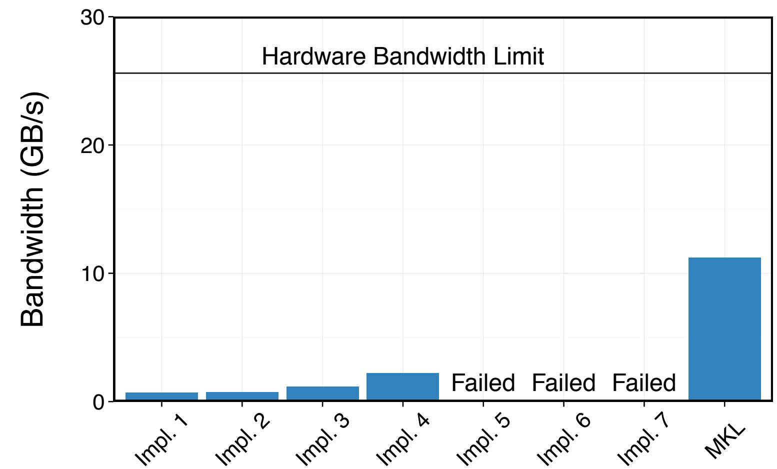
(a) Nvidia's GTX 480 GPU.

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

Performance Results AMD and Intel



(b) AMD's HD 7970 GPU.

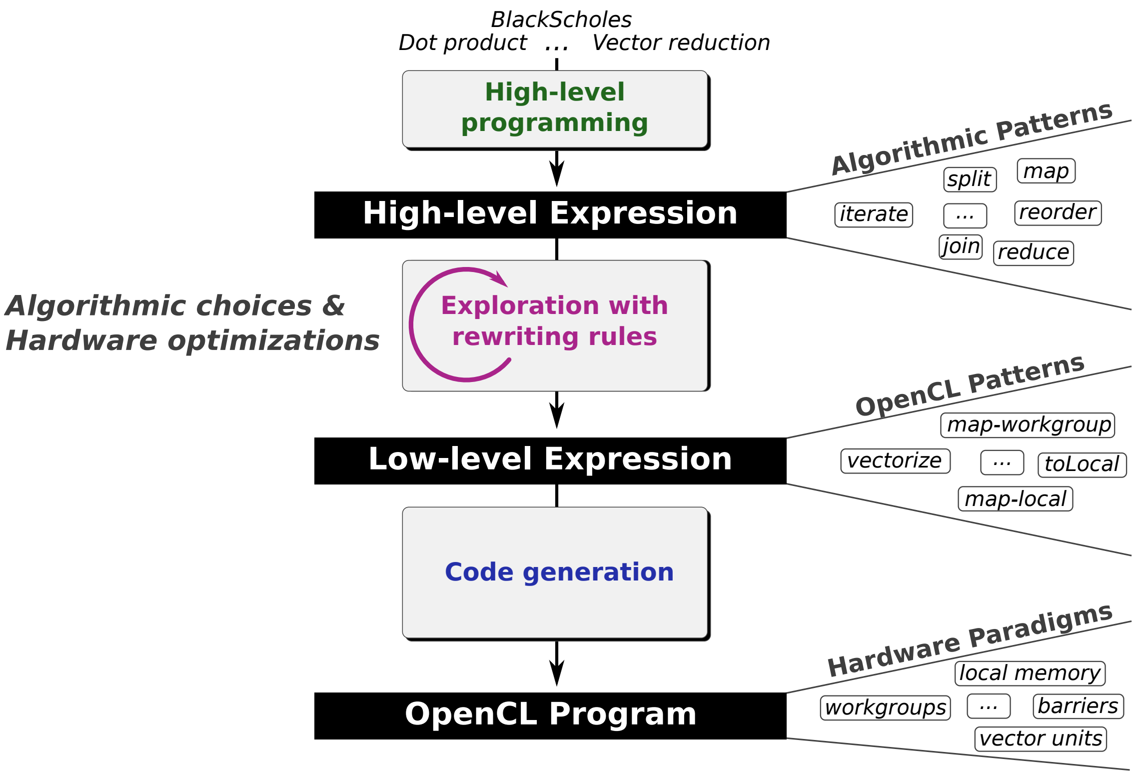


(c) Intel's E5530 dual-socket CPU.

- … unfortunately, optimisations in OpenCL are not portable!

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

Generating Performance Portable Code using Rewrite Rules



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

Walkthrough

① $sum(vec) = reduce(+, 0, vec)$

rewrite rules

code generation

②

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

③

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

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

Walkthrough

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

rewrite rules   code generation

②

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

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];
}
```

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

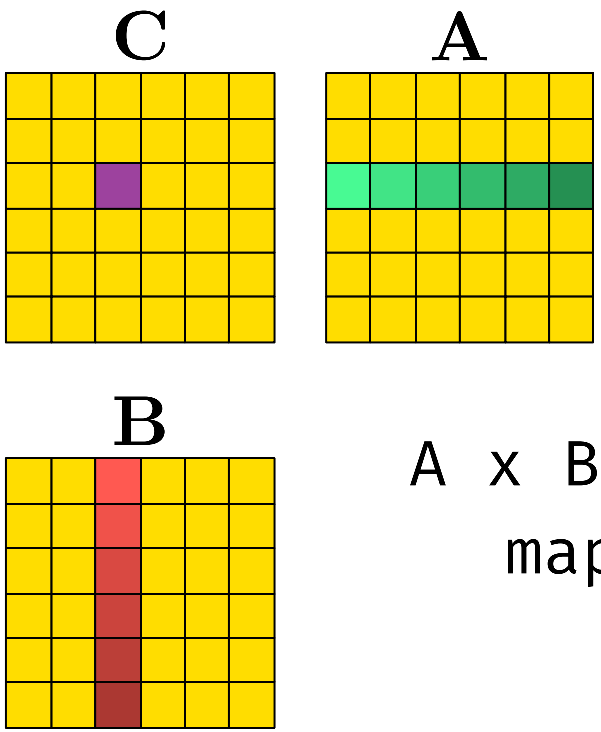
```
scal(a, vec) = map(λ x ↦ x*a, vec)

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) ) )
```

① High-Level Programs




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

Walkthrough

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

rewrite rules

code generation

②

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

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];
}
```

Walkthrough

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

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];
}
```

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:

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

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

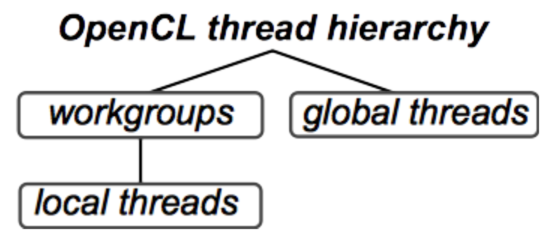
$$reducePart\ f\ z \rightarrow join\ \circ map\ (reducePart\ f\ z) \circ split\ n$$

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

THE UNIVERSITY *of* EDINBURGH informatics

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:

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

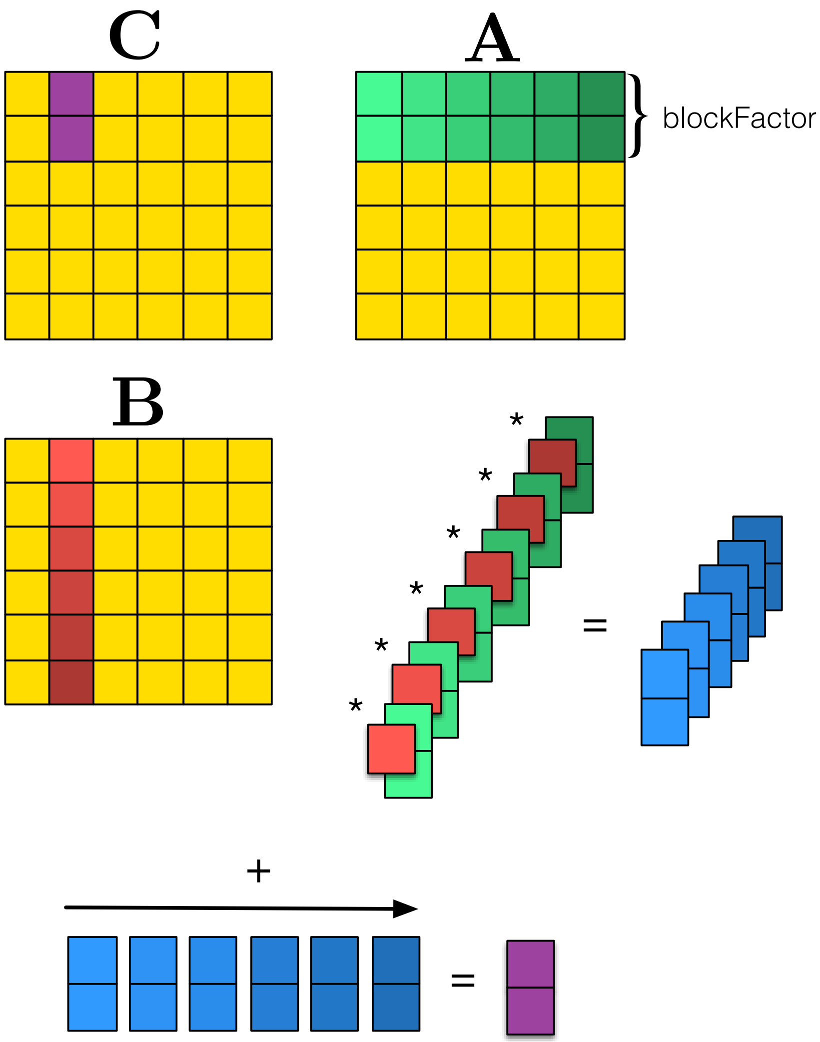
Vectorisation rule:

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

Fusion rule:

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

② Optimisations Expressed using Rewrite Rules
Register Blocking



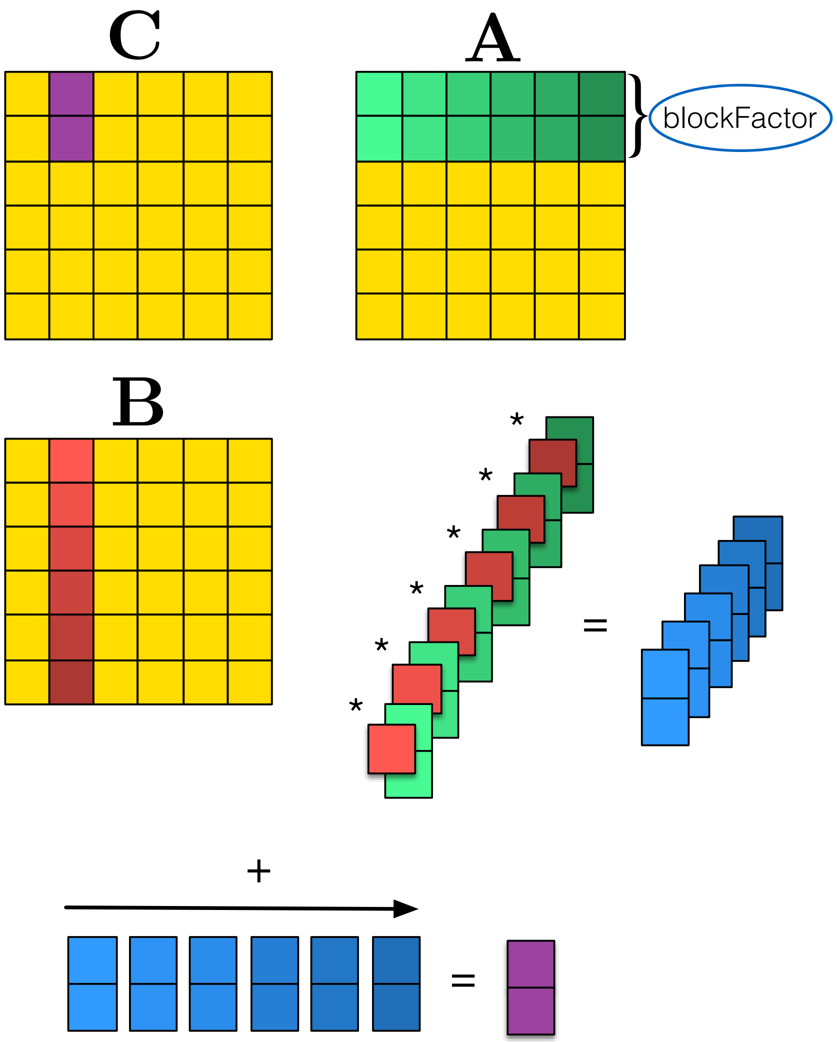

```
kernel void KERNEL(
  const global float* restrict A,
  const global float* restrict B,
  global float* C, int K, int M, int N)
{
  float acc[blockFactor];

  for (int glb_id_1 = get_global_id(1);
       glb_id_1 < M / blockFactor;
       glb_id_1 += get_global_size(1)) {
    for (int glb_id_0 = get_global_id(0); glb_id_0 < N;
         glb_id_0 += get_global_size(0)) {

      for (int i = 0; i < K; i += 1)
        float temp = B[i * N + glb_id_0];
        for (int j = 0; j < blockFactor; j+= 1)
          acc[j] +=
            A[blockFactor * glb_id_1 * K + j * K + i]
              * temp;

      for (int j = 0; j < blockFactor; j += 1)
        C[blockFactor * glb_id_1 * N + j * N + glb_id_0]
            = acc[j];
    }
  }
}
```

② Optimisations Expressed using Rewrite Rules
Register Blocking



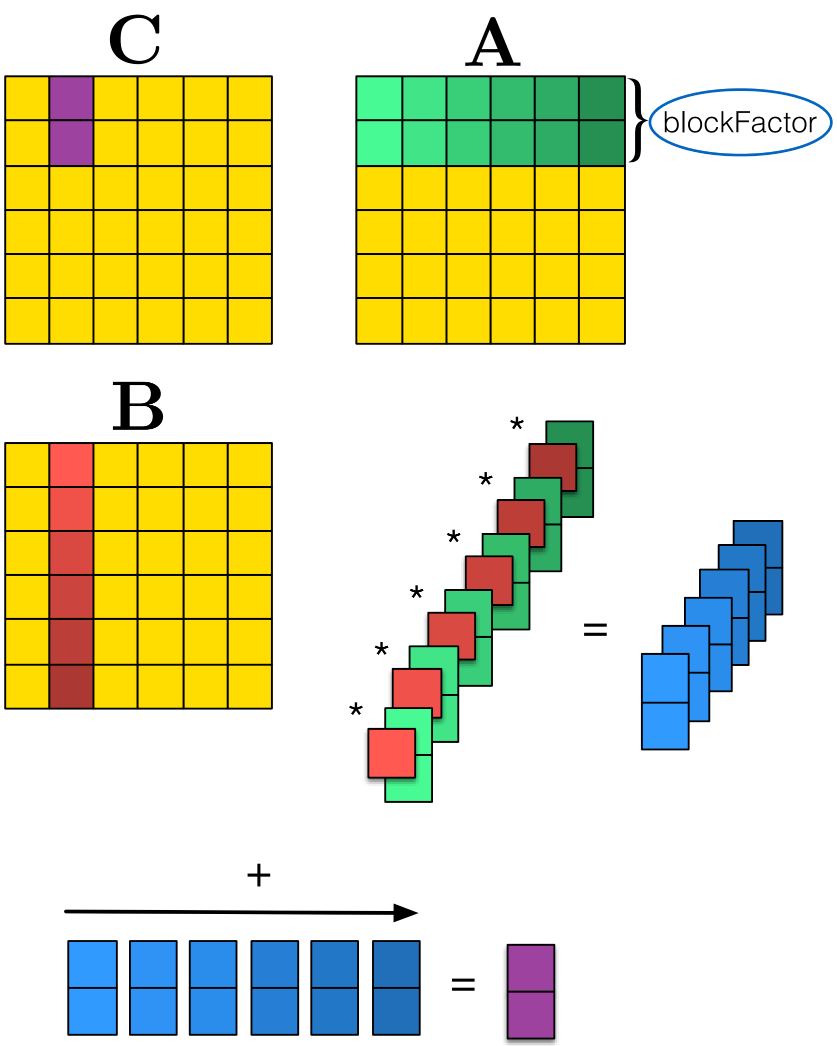

```
kernel void KERNEL(
  const global float* restrict A,
  const global float* restrict B,
  global float* C, int K, int M, int N)
{
  float acc[blockFactor];

  for (int glb_id_1 = get_global_id(1);
        glb_id_1 < M / blockFactor;
        glb_id_1 += get_global_size(1)) {
    for (int glb_id_0 = get_global_id(0); glb_id_0 < N;
        glb_id_0 += get_global_size(0)) {

      for (int i = 0; i < K; i += 1)
        float temp = B[i * N + glb_id_0];
        for (int j = 0; j < blockFactor; j+= 1)
          acc[j] +=
            A[blockFactor * glb_id_1 * K + j * K + i]
              * temp;

      for (int j = 0; j < blockFactor; j += 1)
        C[blockFactor * glb_id_1 * N + j * N + glb_id_0]
            = acc[j];
    }
  }
}
```

② Optimisations Expressed using Rewrite Rules
Register Blocking




```
kernel void KERNEL(
  const global float* restrict A,
  const global float* restrict B,
  global float* C, int K, int M, int N)
{
  float acc[blockFactor];

  for (int glb_id_1 = get_global_id(1);
         glb_id_1 < M / blockFactor;
         glb_id_1 += get_global_size(1)) {
    for (int glb_id_0 = get_global_id(0); glb_id_0 < N;
         glb_id_0 += get_global_size(0)) {

      for (int i = 0; i < K; i += 1)
        float temp = B[i * N + glb_id_0];
        for (int j = 0; j < blockFactor; j+= 1)
          acc[j] +=
            A[blockFactor * glb_id_1 * K + j * K + i]
              * temp;

      for (int j = 0; j < blockFactor; j += 1)
        C[blockFactor * glb_id_1 * N + j * N + glb_id_0]
            = acc[j];
    }
  }
}
```

② Register Blocking as a Macro Rule

- Optimisations are expressed as *Macro Rules:*
  - Series of Rewrites applied to achieve an optimisation goal

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

② Register Blocking as a Series of Rewrites

$$
\begin{aligned}
&Map(\overrightarrow{rowA} \mapsto \\
&\quad Map(\overrightarrow{colB} \mapsto \\
&\quad\quad Reduce(+) \circ Map(*) \\
&\quad\quad\quad \$\ Zip(\overrightarrow{rowA}, \overrightarrow{colB}) \\
&\quad) \circ Transpose()\ \$\ \mathbf{B} \\
&)\ \$\ \mathbf{A}
\end{aligned}
$$

$$Map(f) \Rightarrow Join() \circ Map(Map(f)) \circ Split(k)$$

② Register Blocking as a Series of Rewrites

$$
\begin{aligned}
& Map(\overrightarrow{rowA} \mapsto \\
& \quad Map(\overrightarrow{colB} \mapsto \\
& \quad\quad Reduce(+) \circ Map(*) \\
& \quad\quad\quad \$\ Zip(\overrightarrow{rowA}, \overrightarrow{colB}) \\
& \quad) \circ Transpose()\ \$\ \mathbf{B} \\
&)\ \$\ \mathbf{A}
\end{aligned}
$$

$$
\begin{aligned}
& Join() \circ Map(rowsA \mapsto \\
& \quad Map(\overrightarrow{rowA} \mapsto \\
& \quad\quad Map(\overrightarrow{colB} \mapsto \\
& \quad\quad\quad Reduce(+) \circ Map(*) \\
& \quad\quad\quad\quad \$\ Zip(\overrightarrow{rowA}, \overrightarrow{colB}) \\
& \quad\quad) \circ Transpose()\ \$\ \mathbf{B} \\
& \quad)\ \$\ rowsA \\
&) \circ Split(blockFactor)\ \$\ \mathbf{A}
\end{aligned}
$$

$$Map(f) \Rightarrow Join() \circ Map(Map(f)) \circ Split(k)$$

② Register Blocking as a Series of Rewrites

$$
\begin{aligned}
&Join() \circ Map(rowsA \mapsto \\
&\quad Map(\overrightarrow{rowA} \mapsto \\
&\quad\quad Map(\overrightarrow{colB} \mapsto \\
&\quad\quad\quad Reduce(+) \circ Map(*) \\
&\quad\quad\quad\quad \$\ Zip(\overrightarrow{rowA}, \overrightarrow{colB}) \\
&\quad\quad) \circ Transpose()\ \$\ \mathbf{B} \\
&\quad)\ \$\ rowsA \\
&) \circ Split(blockFactor)\ \$\ \mathbf{A}
\end{aligned}
$$

$$
Map(a \mapsto Map(b \mapsto f(a,b))) \Rightarrow
$$

$$
Transpose() \circ Map(b \mapsto Map(a \mapsto f(a,b)))
$$

② Register Blocking as a Series of Rewrites

$$
\begin{aligned}
&Join() \circ Map(rowsA \mapsto \\
&\quad Map(\overrightarrow{rowA} \mapsto \\
&\quad\quad Map(\overrightarrow{colB} \mapsto \\
&\quad\quad\quad Reduce(+) \circ Map(*) \\
&\quad\quad\quad\quad \$\ Zip(\overrightarrow{rowA}, \overrightarrow{colB}) \\
&\quad\quad) \circ Transpose()\ \$\ \mathbf{B} \\
&\quad)\ \$\ rowsA \\
&) \circ Split(blockFactor)\ \$\ \mathbf{A}
\end{aligned}
$$

$$
\begin{aligned}
&Join() \circ Map(rowsA \mapsto \\
&\quad Transpose() \circ Map(\overrightarrow{colB} \mapsto \\
&\quad\quad Map(\overrightarrow{rowA} \mapsto \\
&\quad\quad\quad Reduce(+) \circ Map(*) \\
&\quad\quad\quad\quad \$\ Zip(\overrightarrow{rowA}, \overrightarrow{colB}) \\
&\quad\quad)\ \$\ rowsA \\
&\quad) \circ Transpose()\ \$\ \mathbf{B} \\
&) \circ Split(blockFactor)\ \$\ \mathbf{A}
\end{aligned}
$$

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

② Register Blocking as a Series of Rewrites

$$
\begin{aligned}
&Join() \circ Map(rowsA \mapsto \\
&\quad Transpose() \circ Map(\overrightarrow{colB} \mapsto \\
&\qquad Map(\overrightarrow{rowA} \mapsto \\
&\qquad\quad Reduce(+) \circ Map(*) \\
&\qquad\qquad \$\ Zip(\overrightarrow{rowA}, \overrightarrow{colB}) \\
&\qquad)\ \$\ rowsA \\
&\quad) \circ Transpose()\ \$\ \mathbf{B} \\
&) \circ Split(blockFactor)\ \$\ \mathbf{A}
\end{aligned}
$$

$$Map(f \circ g) \Rightarrow Map(f) \circ Map(g)$$

② Register Blocking as a Series of Rewrites

$$
\begin{array}{l}
Join() \circ Map(rowsA \mapsto \\
\quad Transpose() \circ Map(\overrightarrow{colB} \mapsto \\
\quad\quad Map(\overrightarrow{rowA} \mapsto \\
\quad\quad\quad Reduce(+) \circ Map(*) \\
\quad\quad\quad\quad \$\ Zip(\overrightarrow{rowA}, \overrightarrow{colB}) \\
\quad\quad)\ \$\ rowsA \\
\quad) \circ Transpose()\ \$\ \mathbf{B} \\
) \circ Split(blockFactor)\ \$\ \mathbf{A}
\end{array}
$$

$$
\begin{array}{l}
Join() \circ Map(rowsA \mapsto \\
\quad Transpose() \circ Map(\overrightarrow{colB} \mapsto \\
\quad\quad Map(\\
\quad\quad\quad Reduce(+) \\
\quad\quad) \circ Map(\overrightarrow{rowA} \mapsto \\
\quad\quad\quad Map(*)\ \$\ Zip(\overrightarrow{rowA}, \overrightarrow{colB}) \\
\quad\quad)\ \$\ rowsA \\
\quad) \circ Transpose()\ \$\ \mathbf{B} \\
) \circ Split(blockFactor)\ \$\ \mathbf{A}
\end{array}
$$

$$Map(f \circ g) \Rightarrow Map(f) \circ Map(g)$$

② Register Blocking as a Series of Rewrites

$$
\begin{aligned}
&Join() \circ Map(rowsA \mapsto \\
&\quad Transpose() \circ Map(\overrightarrow{colB} \mapsto \\
&\qquad Map(\\
&\qquad\quad Reduce(+) \\
&\qquad) \circ Map(\overrightarrow{rowA} \mapsto \\
&\qquad\quad Map(*) \ \$ \ Zip(\overrightarrow{rowA}, \overrightarrow{colB}) \\
&\qquad) \ \$ \ rowsA \\
&\quad) \circ Transpose() \ \$ \ \mathbf{B} \\
&) \circ Split(blockFactor) \ \$ \ \mathbf{A}
\end{aligned}
$$

$$
Map(Reduce(f)) \Rightarrow
$$

$$
Transpose() \circ Reduce(Map(f) \circ Zip())
$$

② Register Blocking as a Series of Rewrites

$$
\begin{aligned}
&Join() \circ Map(rowsA \mapsto \\
&\quad Transpose() \circ Map(\overrightarrow{colB} \mapsto \\
&\qquad Map(\\
&\qquad\quad Reduce(+) \\
&\qquad) \circ Map(\overrightarrow{rowA} \mapsto \\
&\qquad\quad Map(*) \; \$ \; Zip(\overrightarrow{rowA}, \overrightarrow{colB}) \\
&\qquad) \; \$ \; rowsA \\
&\quad) \circ Transpose() \; \$ \; \mathbf{B} \\
&) \circ Split(blockFactor) \; \$ \; \mathbf{A}
\end{aligned}
$$

$$
\begin{aligned}
&Join() \circ Map(rowsA \mapsto \\
&\quad Transpose() \circ Map(\overrightarrow{colB} \mapsto \\
&\qquad Transpose() \circ Reduce((\overrightarrow{acc}, \overrightarrow{next}) \mapsto \\
&\qquad\quad Map(+) \; \$ \; Zip(\overrightarrow{acc}, \overrightarrow{next}) \\
&\qquad) \circ Transpose() \circ Map(\overrightarrow{rowA} \mapsto \\
&\qquad\quad Map(*) \$ \; Zip(\overrightarrow{rowA}, \overrightarrow{colB}) \\
&\qquad) \; \$ \; rowsA \\
&\quad) \circ Transpose() \; \$ \; \mathbf{B} \\
&) \circ Split(blockFactor) \; \$ \; \mathbf{A}
\end{aligned}
$$

$$
\begin{aligned}
&Map(Reduce(f)) \Rightarrow \\
&Transpose() \circ Reduce(Map(f) \circ Zip())
\end{aligned}
$$

② Register Blocking as a Series of Rewrites

$$
\begin{aligned}
&Join() \circ Map(rowsA \mapsto \\
&\quad Transpose() \circ Map(\overrightarrow{colB} \mapsto \\
&\qquad Transpose() \circ Reduce((\overrightarrow{acc}, \overrightarrow{next}) \mapsto \\
&\qquad\quad Map(+)\ \$\ Zip(\overrightarrow{acc}, \overrightarrow{next}) \\
&\qquad) \circ Transpose() \circ Map(\overrightarrow{rowA} \mapsto \\
&\qquad\quad Map(*) \$\ Zip(\overrightarrow{rowA}, \overrightarrow{colB}) \\
&\qquad)\ \$\ rowsA \\
&\quad) \circ Transpose()\ \$\ \mathbf{B} \\
&) \circ Split(blockFactor)\ \$\ \mathbf{A}
\end{aligned}
$$

$$
\begin{gathered}
Map(Map(f)) \Rightarrow \\
Transpose() \circ Map(Map(f)) \circ Transpose()
\end{gathered}
$$

② Register Blocking as a Series of Rewrites

$$
\begin{aligned}
&Join() \circ Map(rowsA \mapsto \\
&\quad Transpose() \circ Map(\overrightarrow{colB} \mapsto \\
&\quad\quad Transpose() \circ Reduce((\overrightarrow{acc}, \overrightarrow{next}) \mapsto \\
&\quad\quad\quad Map(+) \$ Zip(\overrightarrow{acc}, \overrightarrow{next}) \\
&\quad\quad) \circ Transpose() \circ Map(\overrightarrow{rowA} \mapsto \\
&\quad\quad\quad Map(*) \$ Zip(\overrightarrow{rowA}, \overrightarrow{colB}) \\
&\quad\quad) \$ rowsA \\
&\quad) \circ Transpose() \$ \mathbf{B} \\
&) \circ Split(blockFactor) \$ \mathbf{A}
\end{aligned}
$$

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

$$Map(Map(f)) \Rightarrow$$

$$Transpose() \circ Map(Map(f)) \circ Transpose()$$

② Register Blocking as a Series of Rewrites

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

$$Transpose() \circ Transpose() \Rightarrow id$$

② Register Blocking as a Series of Rewrites

$$
\begin{array}{l}
Join() \circ Map(rowsA \mapsto \\
\quad Transpose() \circ Map(\overrightarrow{colB} \mapsto \\
\quad\quad Transpose() \circ Reduce((\overrightarrow{acc}, \overrightarrow{next}) \mapsto \\
\quad\quad\quad Map(+) \ \$ \ Zip(\overrightarrow{acc}, \overrightarrow{next}) \\
\quad\quad) \circ Transpose() \\
\quad\quad \circ Transpose() \circ Map(pair \mapsto \\
\quad\quad\quad Map(x \mapsto x * pair.\_1) \ \$ \ pair.\_0 \\
\quad\quad) \ \$ \ Zip(Transpose() \ \$ \ rowsA, \overrightarrow{colB}) \\
\quad) \circ Transpose() \ \$ \ \mathbf{B} \\
) \circ Split(blockFactor) \ \$ \ \mathbf{A}
\end{array}
$$

$$
\begin{array}{l}
Join() \circ Map(rowsA \mapsto \\
\quad Transpose() \circ Map(\overrightarrow{colB} \mapsto \\
\quad\quad Transpose() \circ Reduce((\overrightarrow{acc}, \overrightarrow{next}) \mapsto \\
\quad\quad\quad Map(+) \ \$ \ Zip(\overrightarrow{acc}, \overrightarrow{next}) \\
\quad\quad) \circ Map(pair \mapsto \\
\quad\quad\quad Map(x \mapsto x * pair.\_1) \ \$ \ pair.\_0 \\
\quad\quad) \ \$ \ Zip(Transpose() \ \$ \ rowsA, \overrightarrow{colB}) \\
\quad) \circ Transpose() \ \$ \ \mathbf{B} \\
) \circ Split(blockFactor) \ \$ \ \mathbf{A}
\end{array}
$$

$$Transpose() \circ Transpose() \Rightarrow id$$

② Register Blocking as a Series of Rewrites

$$
\begin{array}{l}
Join() \circ Map(rowsA \mapsto \\
\quad Transpose() \circ Map(\overrightarrow{colB} \mapsto \\
\quad\quad Transpose() \circ Reduce((\overrightarrow{acc}, \overrightarrow{next}) \mapsto \\
\quad\quad\quad Map(+) \; \$ \; Zip(\overrightarrow{acc}, \overrightarrow{next}) \\
\quad\quad) \circ Map(pair \mapsto \\
\quad\quad\quad Map(x \mapsto x * pair.\_1) \; \$ \; pair.\_0 \\
\quad\quad) \; \$ \; Zip(Transpose() \; \$ \; rowsA, \overrightarrow{colB}) \\
\quad) \circ Transpose() \; \$ \; \mathbf{B} \\
) \circ Split(blockFactor) \; \$ \; \mathbf{A}
\end{array}
$$

$$
Reduce(f) \circ Map(g) \Rightarrow
$$

$$
Reduce((acc, x) \mapsto f(acc, g(x)))
$$

② Register Blocking as a Series of Rewrites

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

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

$$Reduce(f) \circ Map(g) \Rightarrow$$

$$Reduce((acc, x) \mapsto f(acc, g(x)))$$

② Register Blocking as a Series of Rewrites

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

$$Map(f) \circ Map(g) \Rightarrow Map(f \circ g)$$

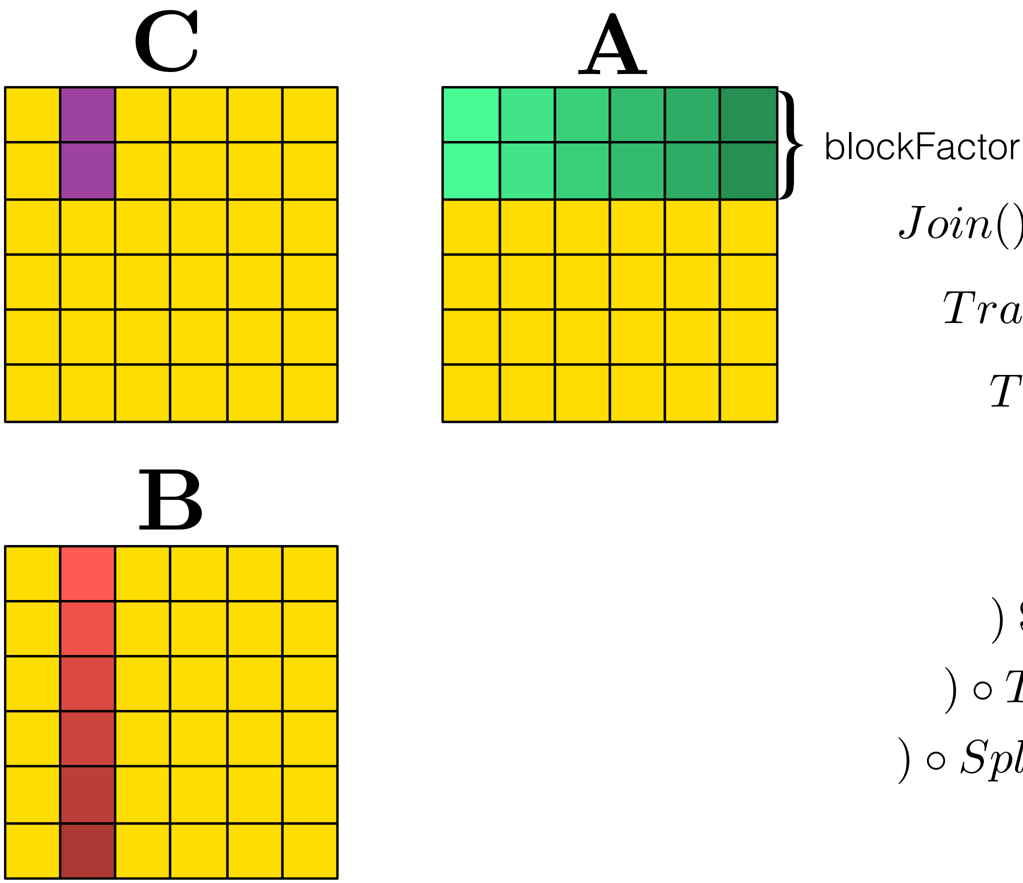
② Register Blocking as a Series of Rewrites

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

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

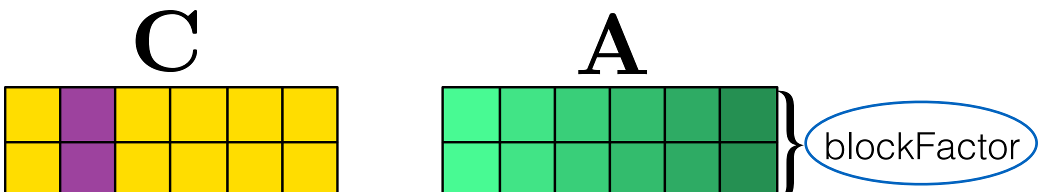
$$Map(f) \circ Map(g) \Rightarrow Map(f \circ g)$$

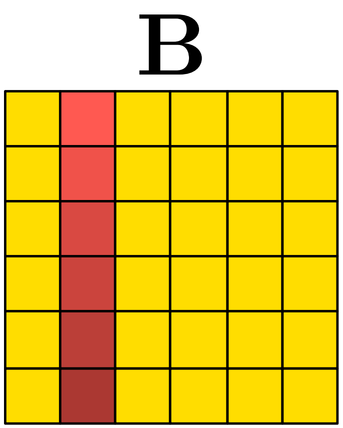
② Register Blocking Functionally Expressed



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

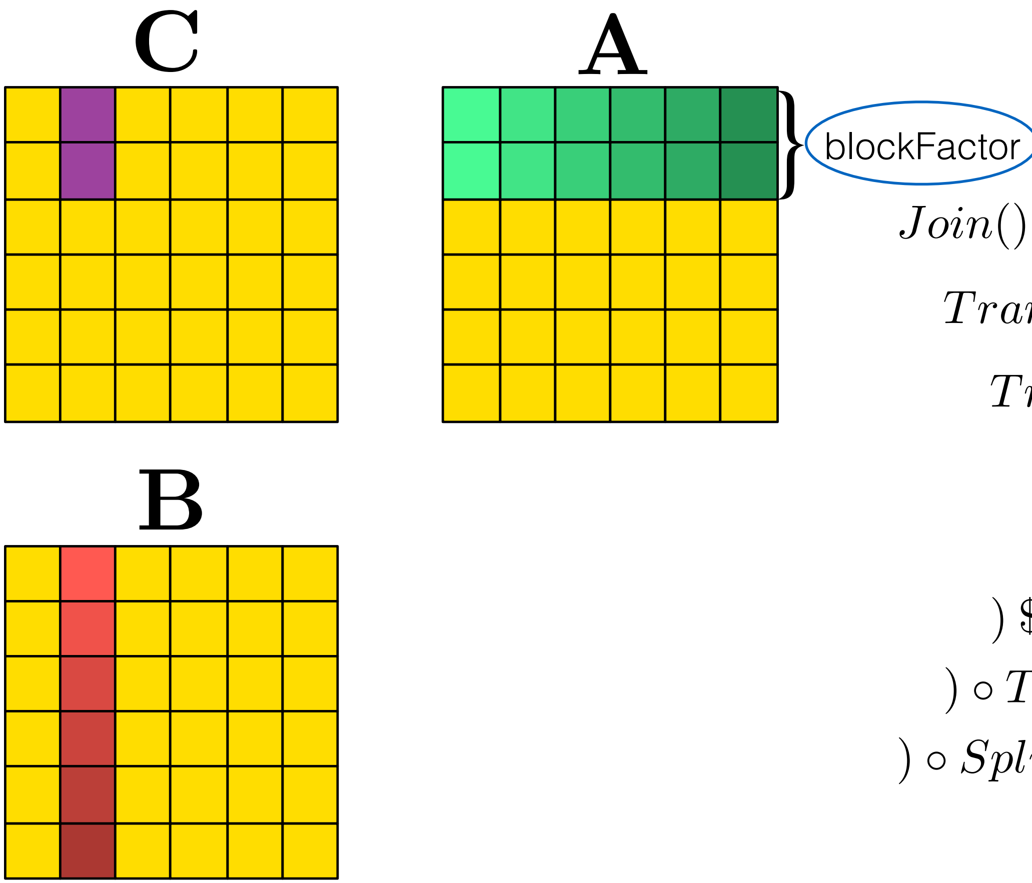
② Register Blocking Functionally Expressed





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

② Register Blocking Functionally Expressed



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

Walkthrough

① $vecSum = reduce\ (+)\ 0$

rewrite rules

code generation

②

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

③

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

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

Walkthrough

① $vecSum = reduce\ (+)\ 0$

rewrite rules

code generation

②

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

③

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

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

③ Pattern based OpenCL Code Generation

- Generate OpenCL code for each OpenCL primitive

$$mapGlobal\ f\ xs \rightarrow$$

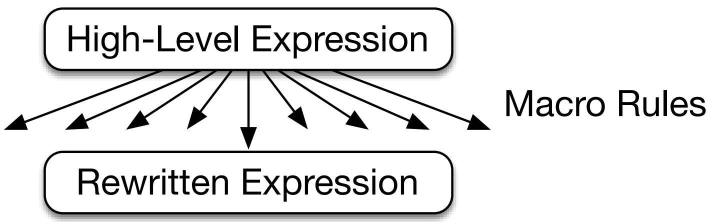
```
for (int g_id = get_global_id(0); g_id < n;
     g_id += get_global_size(0)) {
  output[g_id]  = f(xs[g_id]);
}
```

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

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

$\vdots$ $\vdots$

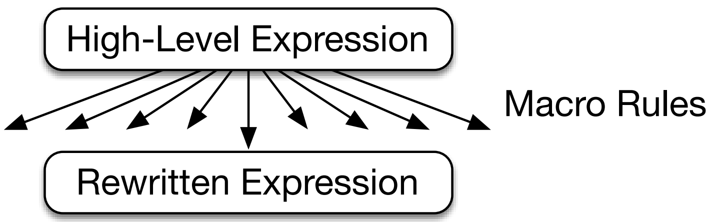
Exploration Strategy



1

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

Exploration Strategy



1

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

1.1

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

1.2

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

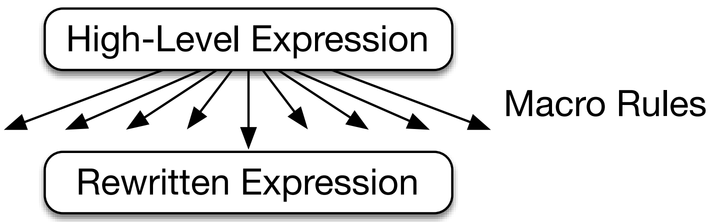
1.3

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

1.4

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

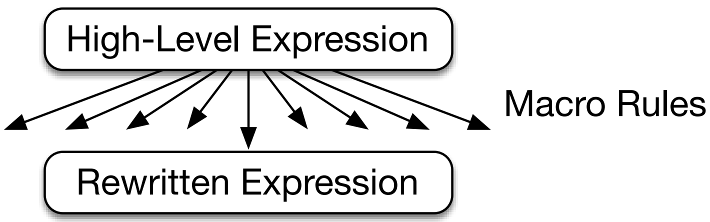
Exploration Strategy



1.3

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

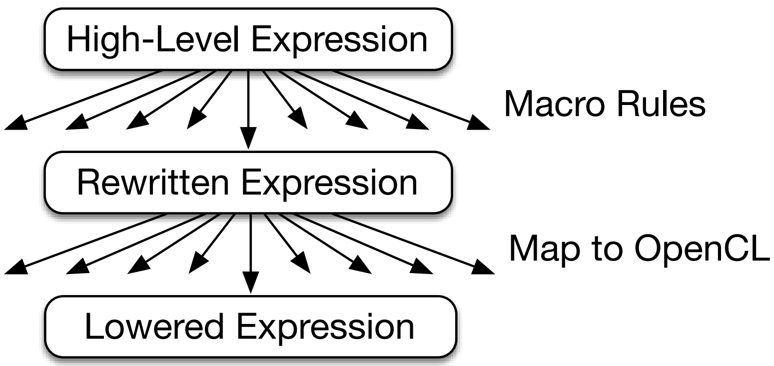
Exploration Strategy



1.3

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

Exploration Strategy

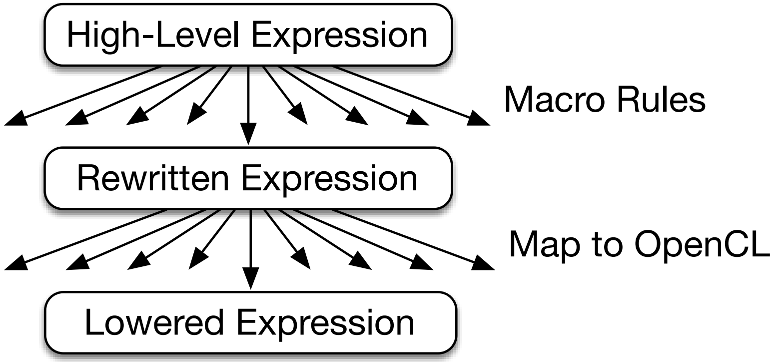


1.3

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

THE UNIVERSITY *of* EDINBURGH
informatics

Exploration Strategy



1.3

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

1.3.1

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

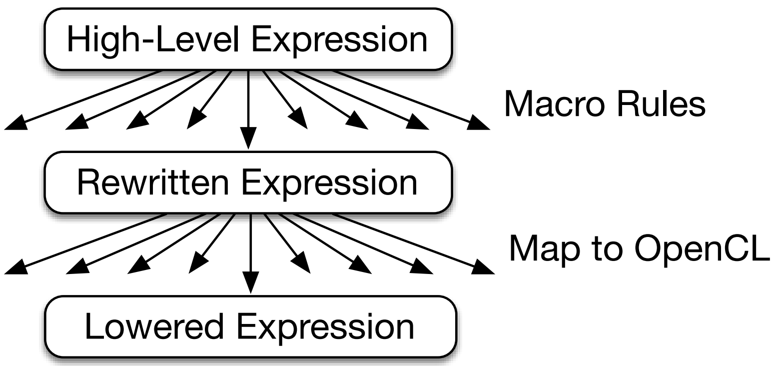
1.3.2

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

1.3.3

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

Exploration Strategy

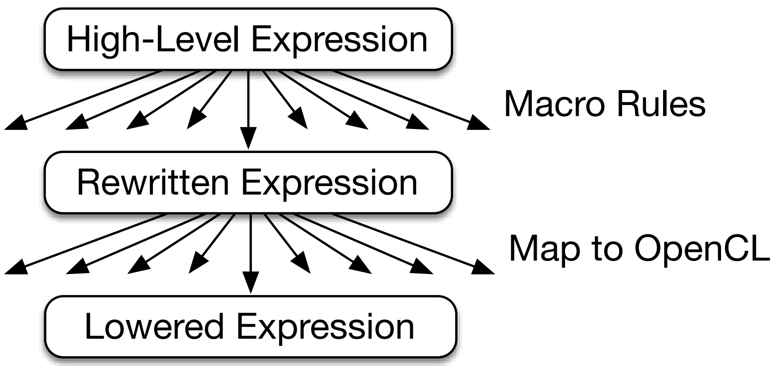


1.3.2

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

THE UNIVERSITY of EDINBURGH
informatics

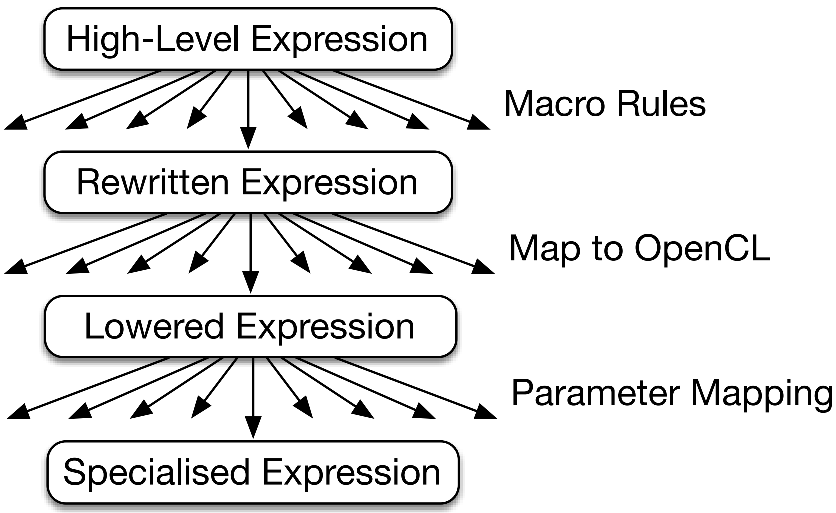
Exploration Strategy



1.3.2

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

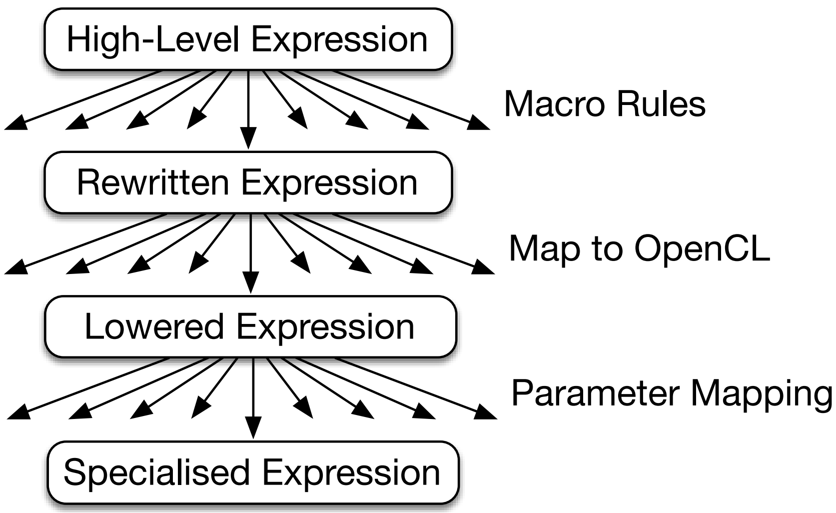
Exploration Strategy

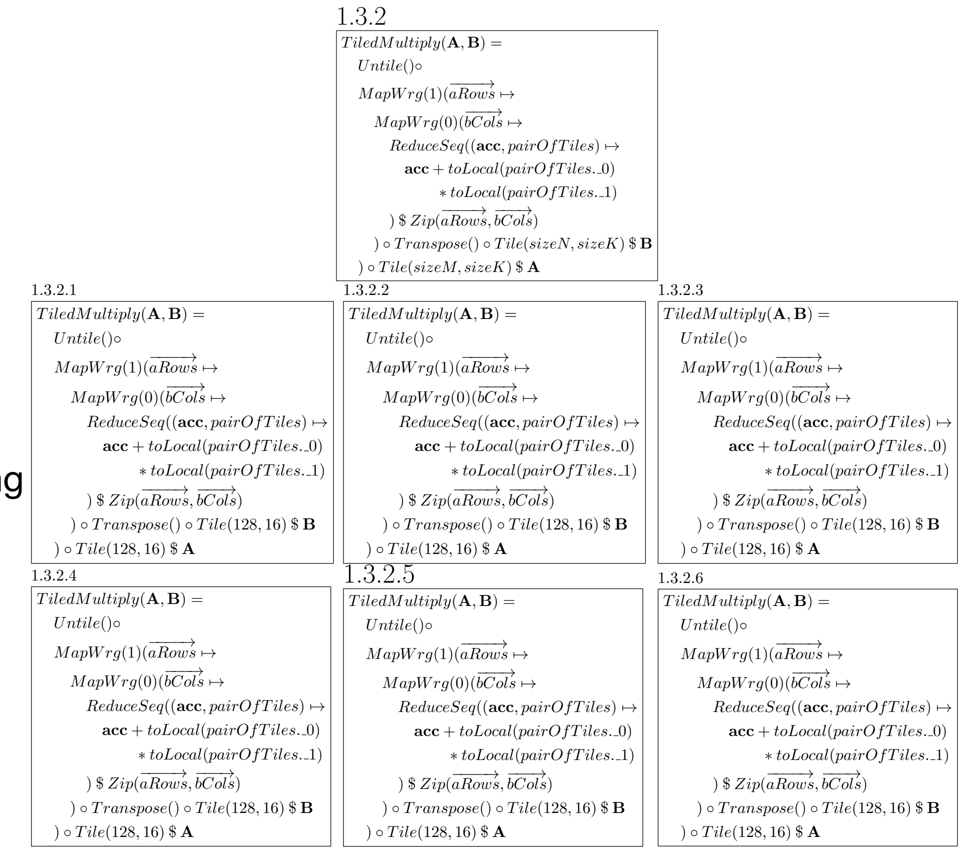


1.3.2

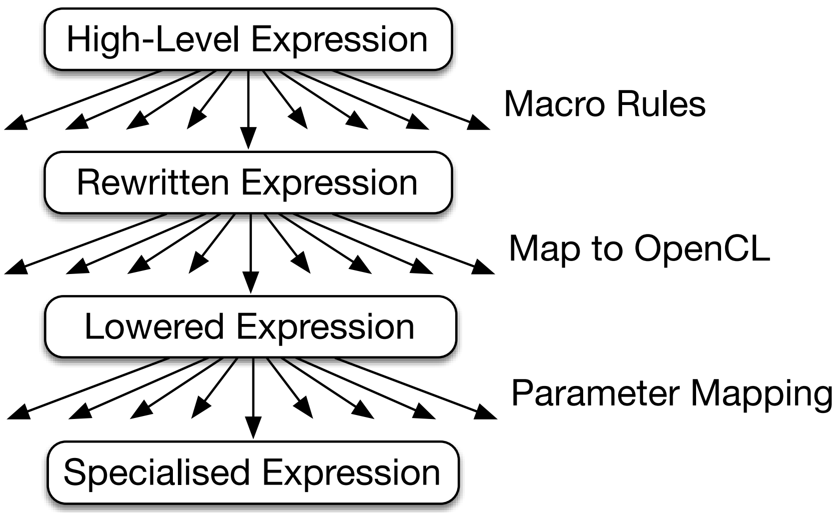
$$
\begin{aligned}
&TiledMultiply(\mathbf{A}, \mathbf{B}) = \\
&\quad Untile() \circ \\
&\quad MapWrg(1)(\overrightarrow{aRows} \mapsto \\
&\qquad MapWrg(0)(\overrightarrow{bCols} \mapsto \\
&\qquad\quad ReduceSeq((\mathbf{acc}, pairOfTiles) \mapsto \\
&\qquad\qquad \mathbf{acc} + toLocal(pairOfTiles.\_0)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

Exploration Strategy





Exploration Strategy

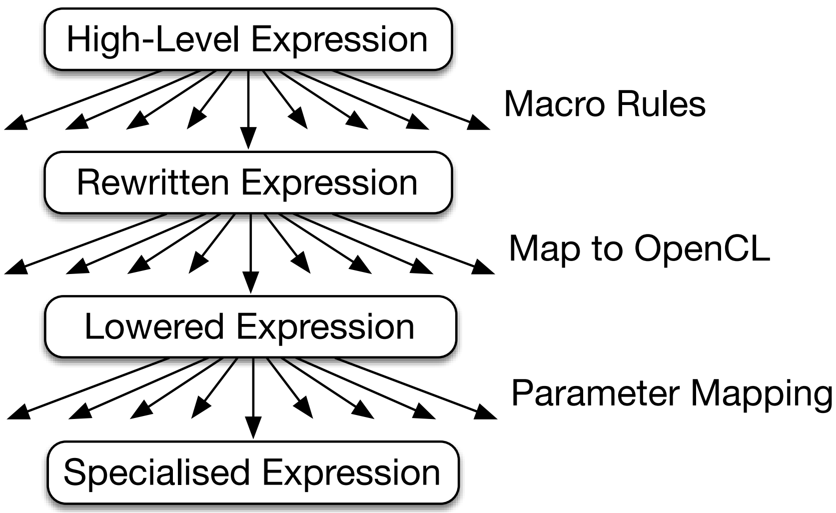


1.3.2.5

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

THE UNIVERSITY *of* EDINBURGH
informatics

Exploration Strategy

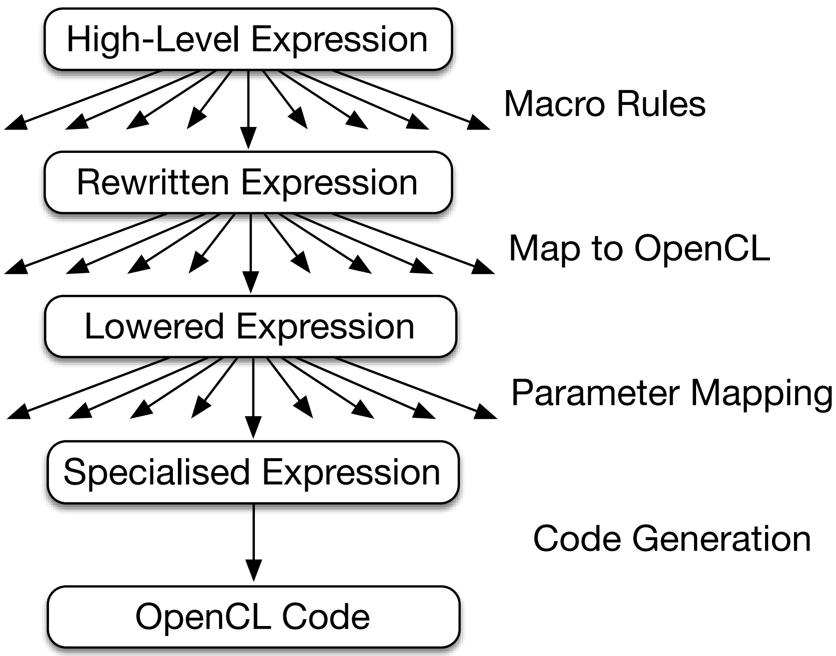


1.3.2.5

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

THE UNIVERSITY *of* EDINBURGH informatics

Exploration Strategy

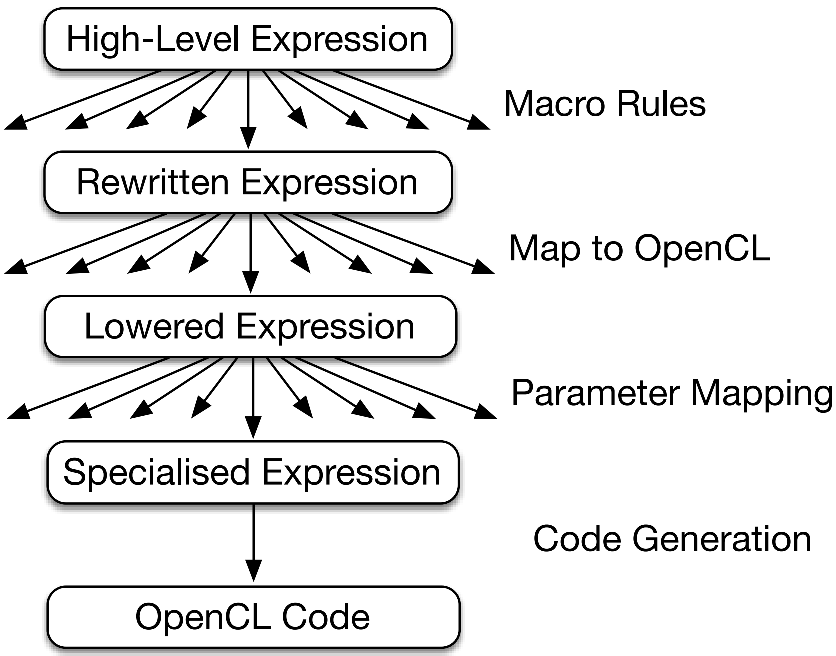


1.3.2.5

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

THE UNIVERSITY *of* EDINBURGH
informatics

Exploration Strategy




```
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

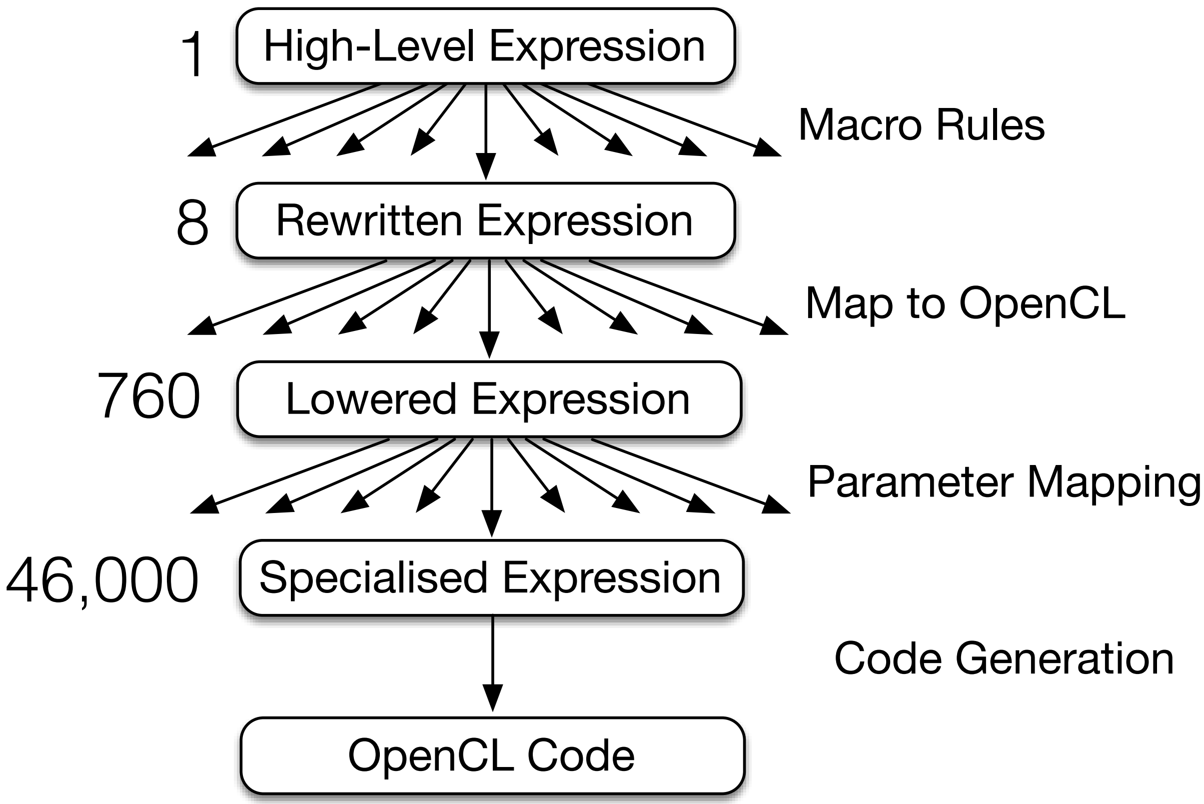
For Map to OpenCL:

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

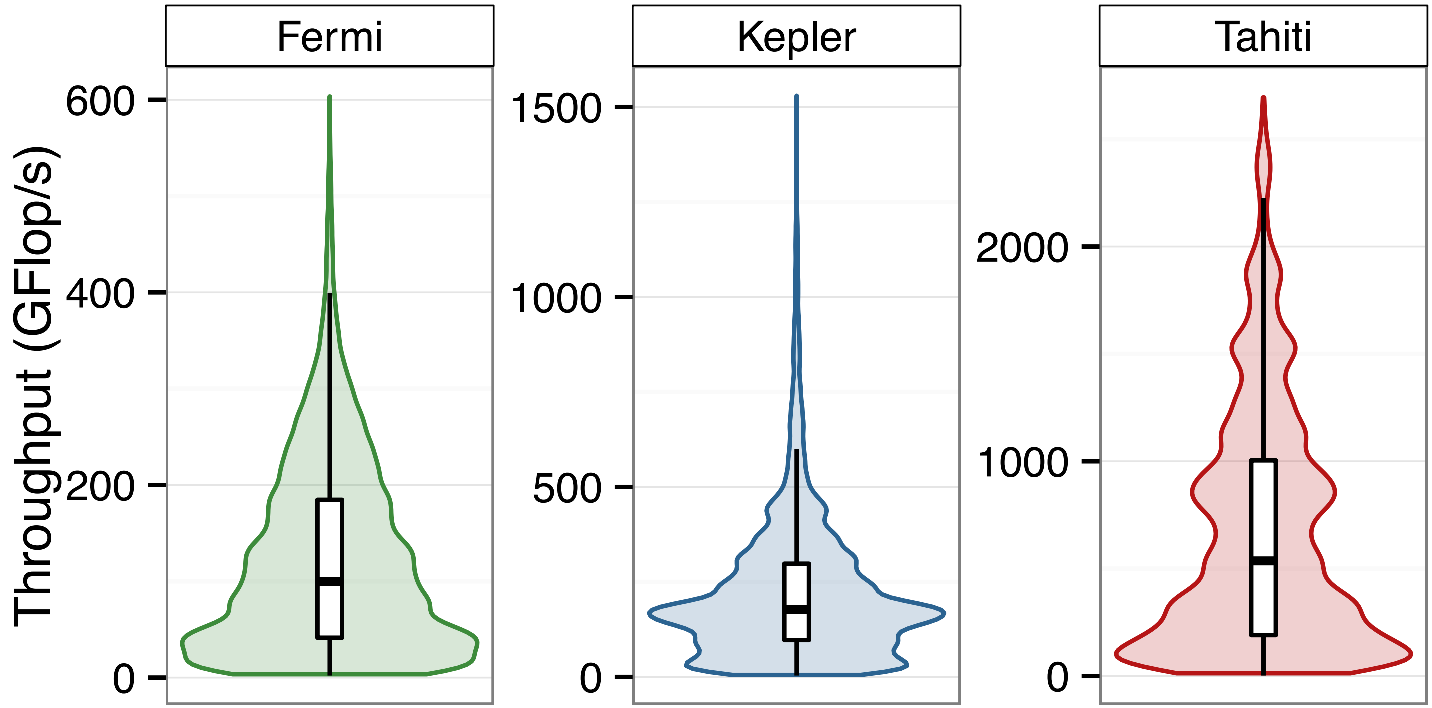
For Parameter Mapping:

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

Exploration in Numbers for Matrix Multiplication

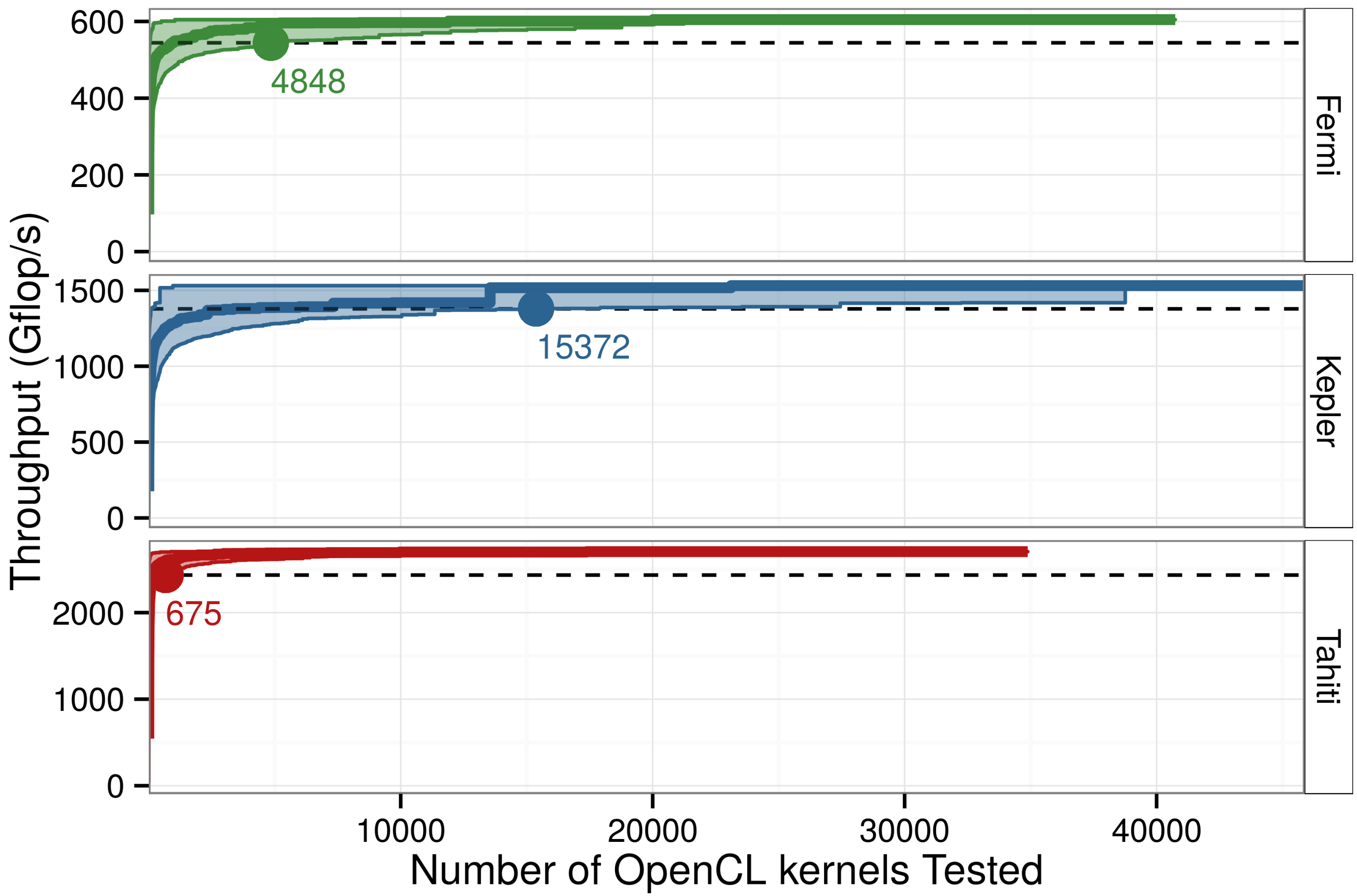


Exploration Space for Matrix Multiplication



Only few OpenCL kernel with very good performance

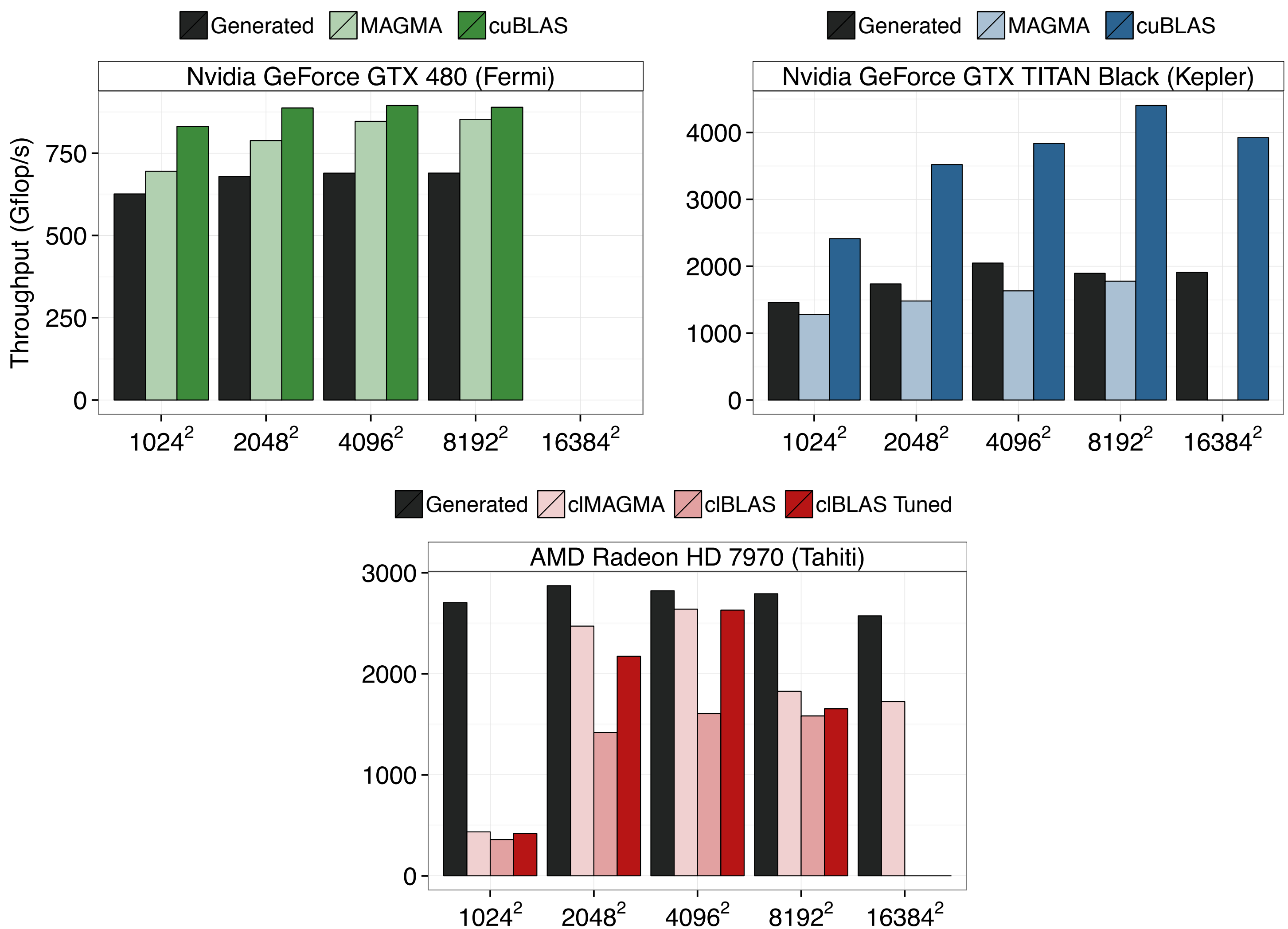
Performance Evolution for Randomised Search



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

THE UNIVERSITY of EDINBURGH informatics

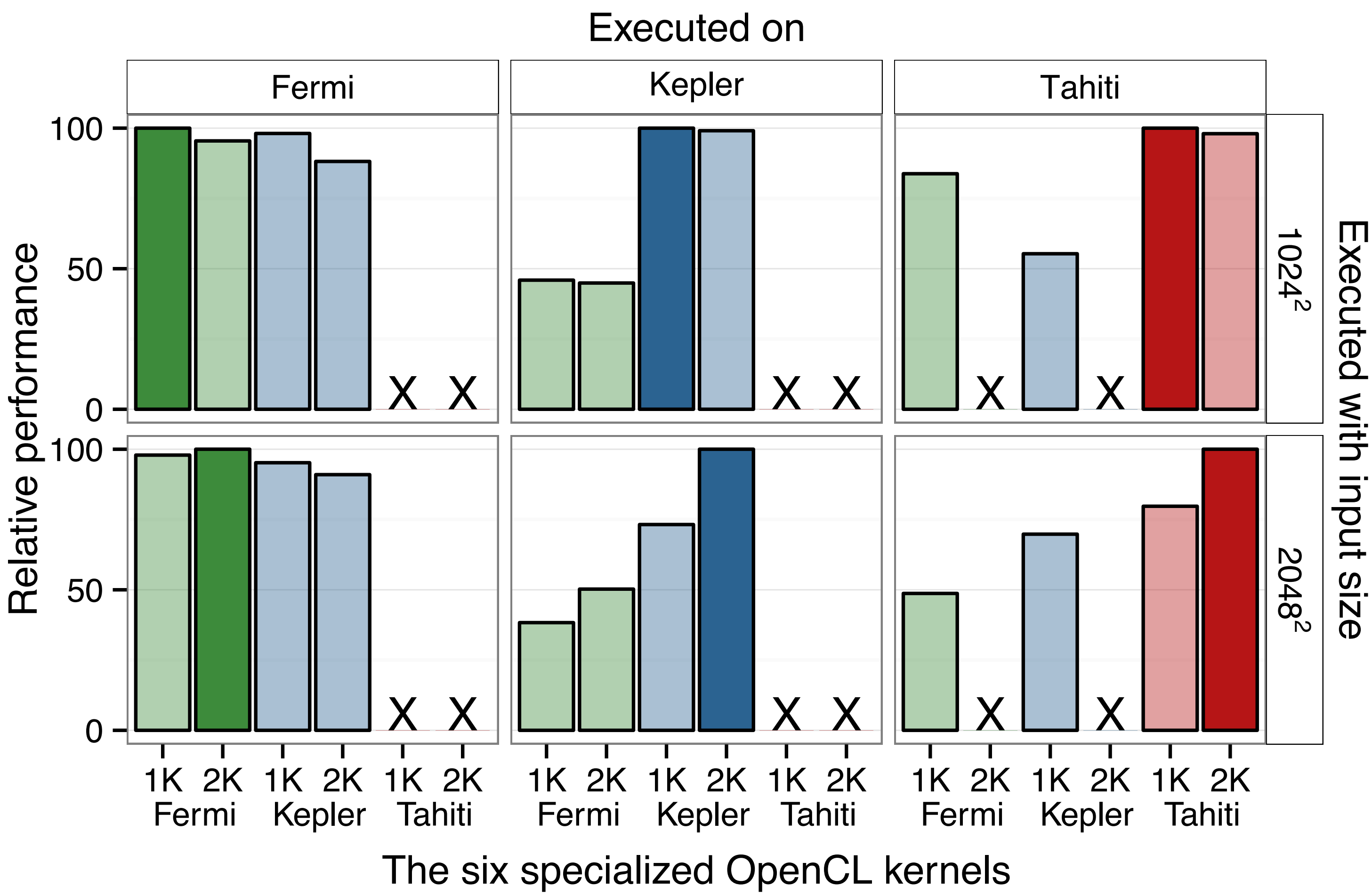
Performance Results Matrix Multiplication



Performance close or better than hand-tuned MAGMA library

THE UNIVERSITY of EDINBURGH informatics

Performance Portability Matrix Multiplication



Generated kernels are specialised for device and input size

THE UNIVERSITY *of* EDINBURGH
informatics

Summary

- OpenCL code is not *performance portable*
- Our approach uses
  - *portable* and functional **high-level primitives**,
  - **OpenCL-specific low-level primitives**, and
  - **rewrite-rules** to generate high *performance* code.
- Rewrite-rules define a space of possible implementations
- Performance on par with specialised, highly-tuned code

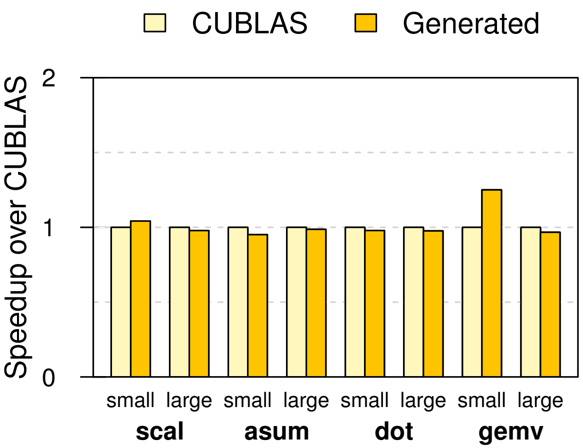
More details in the **ICFP 2015** and **GPGPU 2016** papers available at:
http://homepages.inf.ed.ac.uk/msteuwer/

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

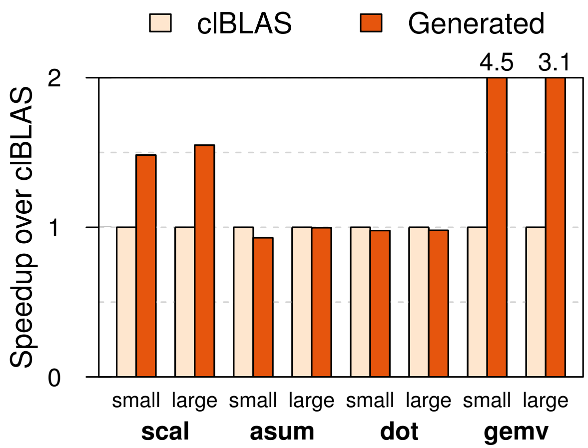
supported by:
Google Oracle Labs

THE UNIVERSITY of EDINBURGH informatics

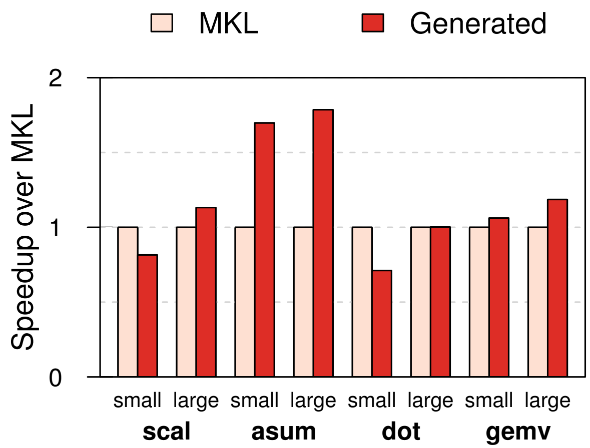
Performance Results more Benchmarks
vs. Hardware-Specific Implementations



(a) Nvidia GPU



(b) AMD GPU

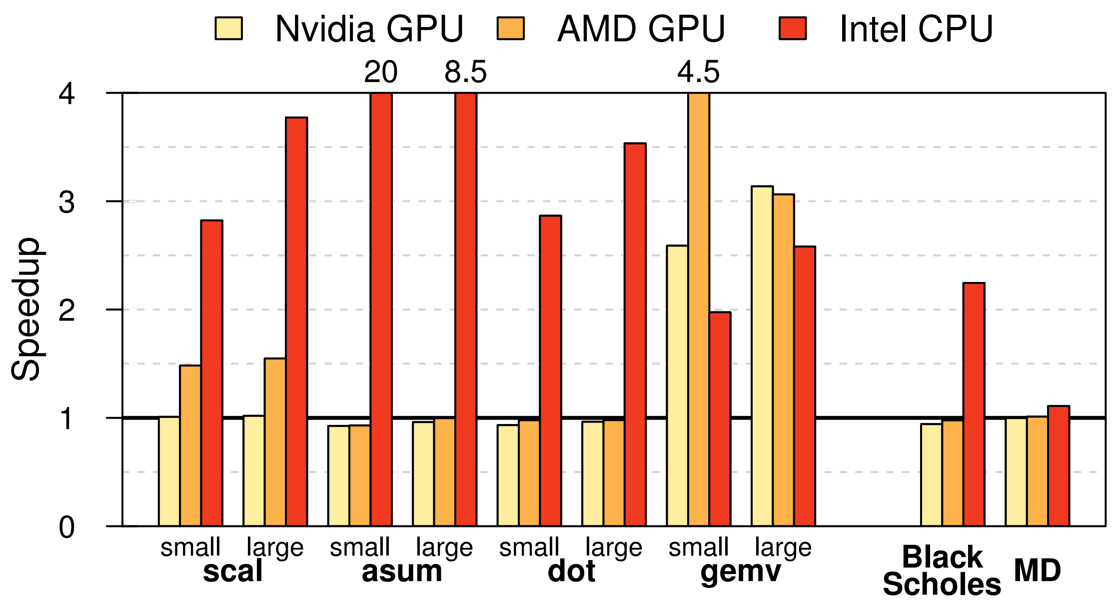


(c) Intel CPU

- Automatically generated code vs. expert written code
- Competitive performance vs. highly optimised implementations
- Up to **4.5x** speedup for *gemv* on AMD

THE UNIVERSITY *of* EDINBURGH
informatics

Performance Results more Benchmarks
vs. Portable Implementation



- Up to **20x** speedup on fairly simple benchmarks vs. portable clBLAS implementation

THE UNIVERSITY *of* EDINBURGH
informatics