Lift: a Data-Parallel Language for High-Performance Parallel Pattern Code Generation

THE UNIVERSITY of EDINBURGH informatics

Christophe Dubach

SCALEW, Cambridge

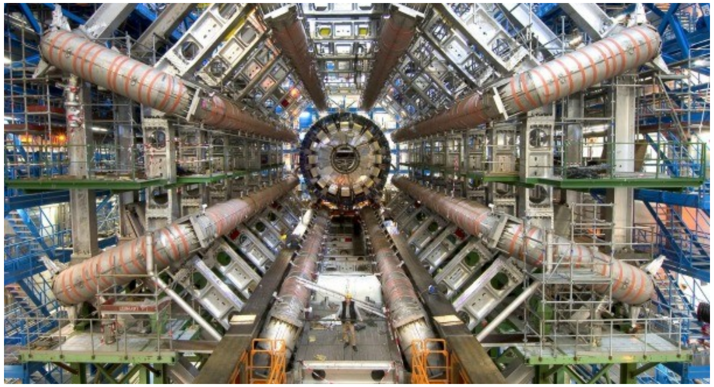
14th July 2016

Michel Steuwer
Postdoc

Thibaut Lutz
former Postdoc
(now at Nvidia)

Toomas Remmelg
PhD student

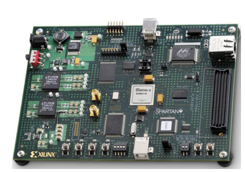
...



Big Data → Big Computers

Big Computers → Accelerators

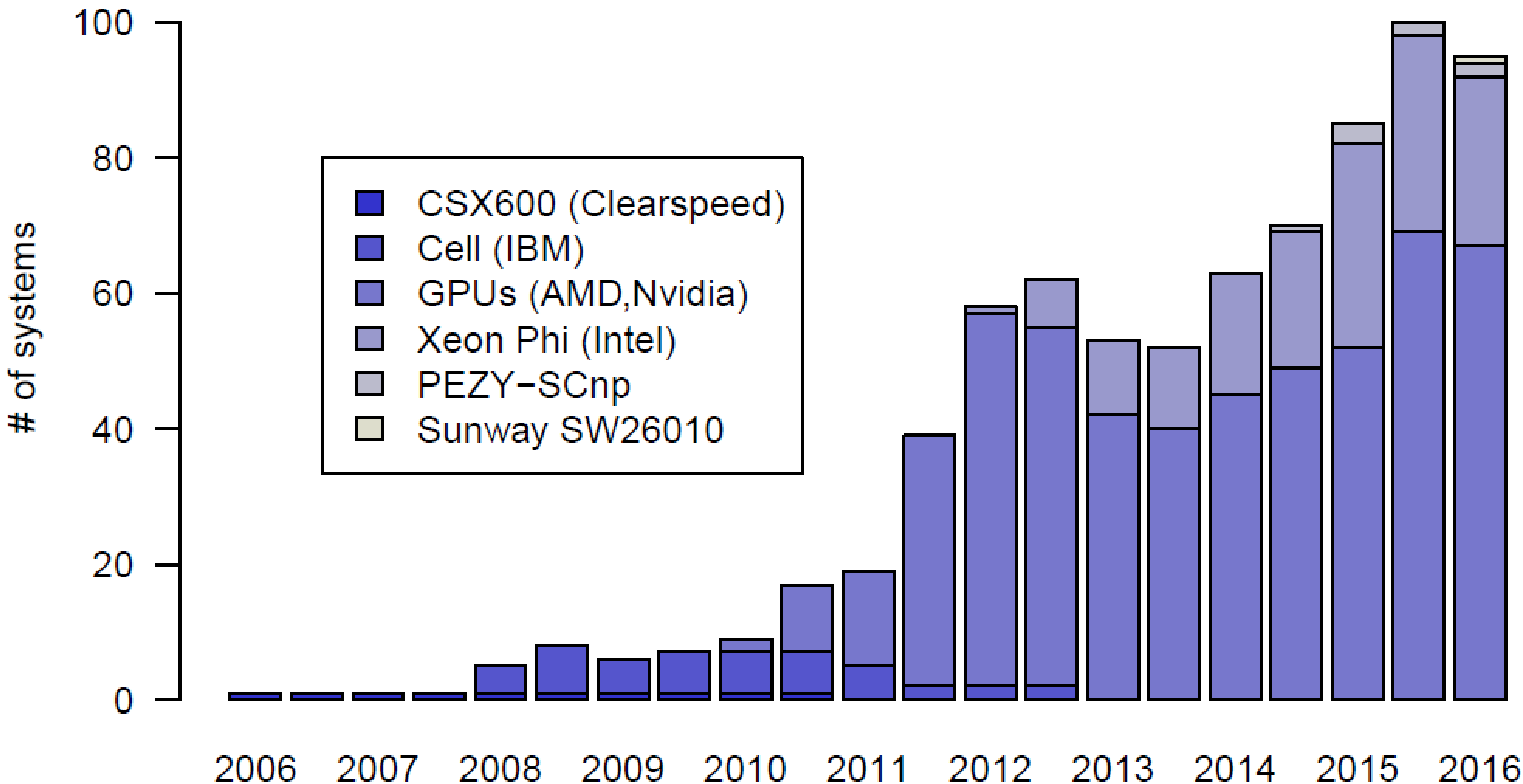
GPU



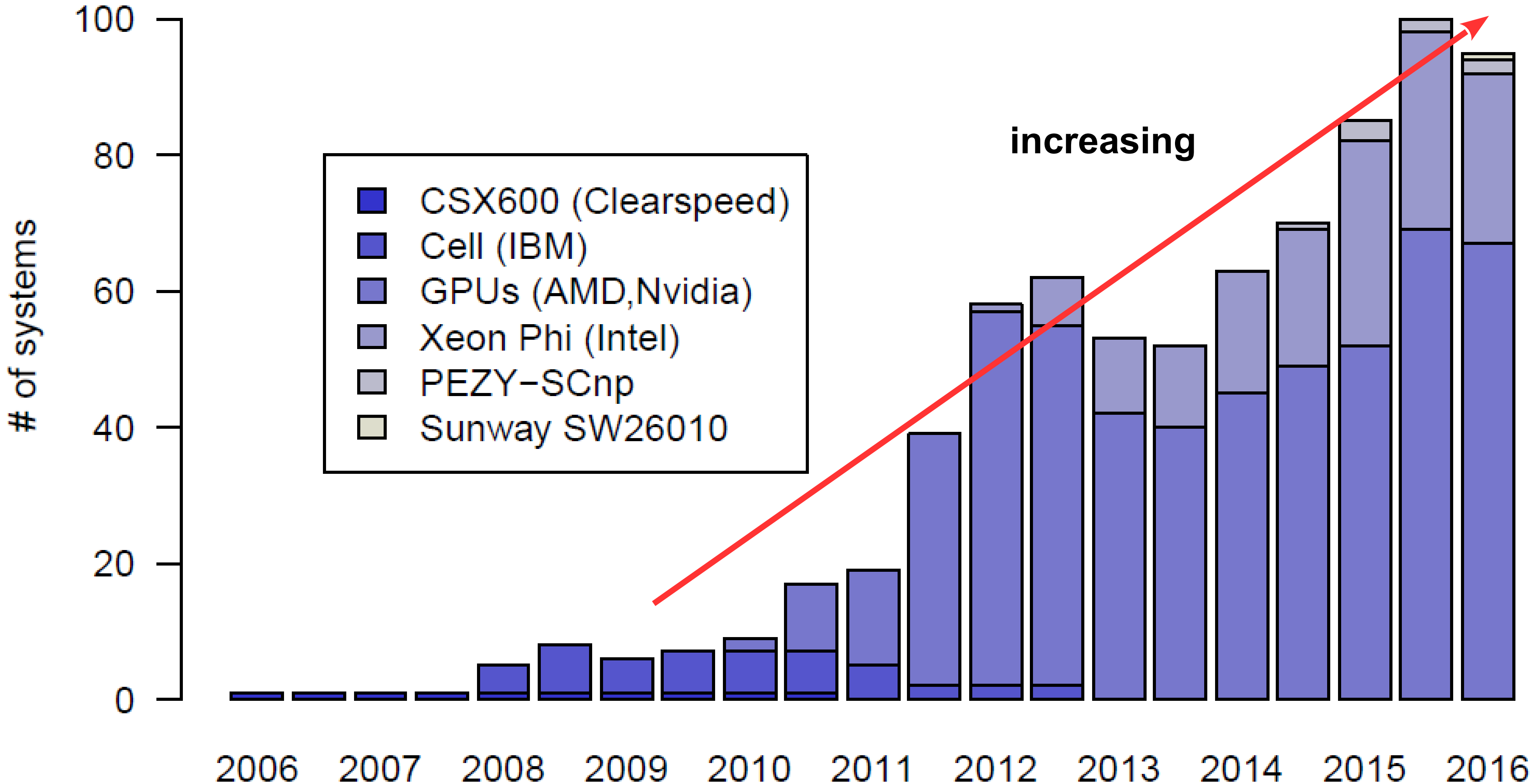
FPGA

CPU/GPU

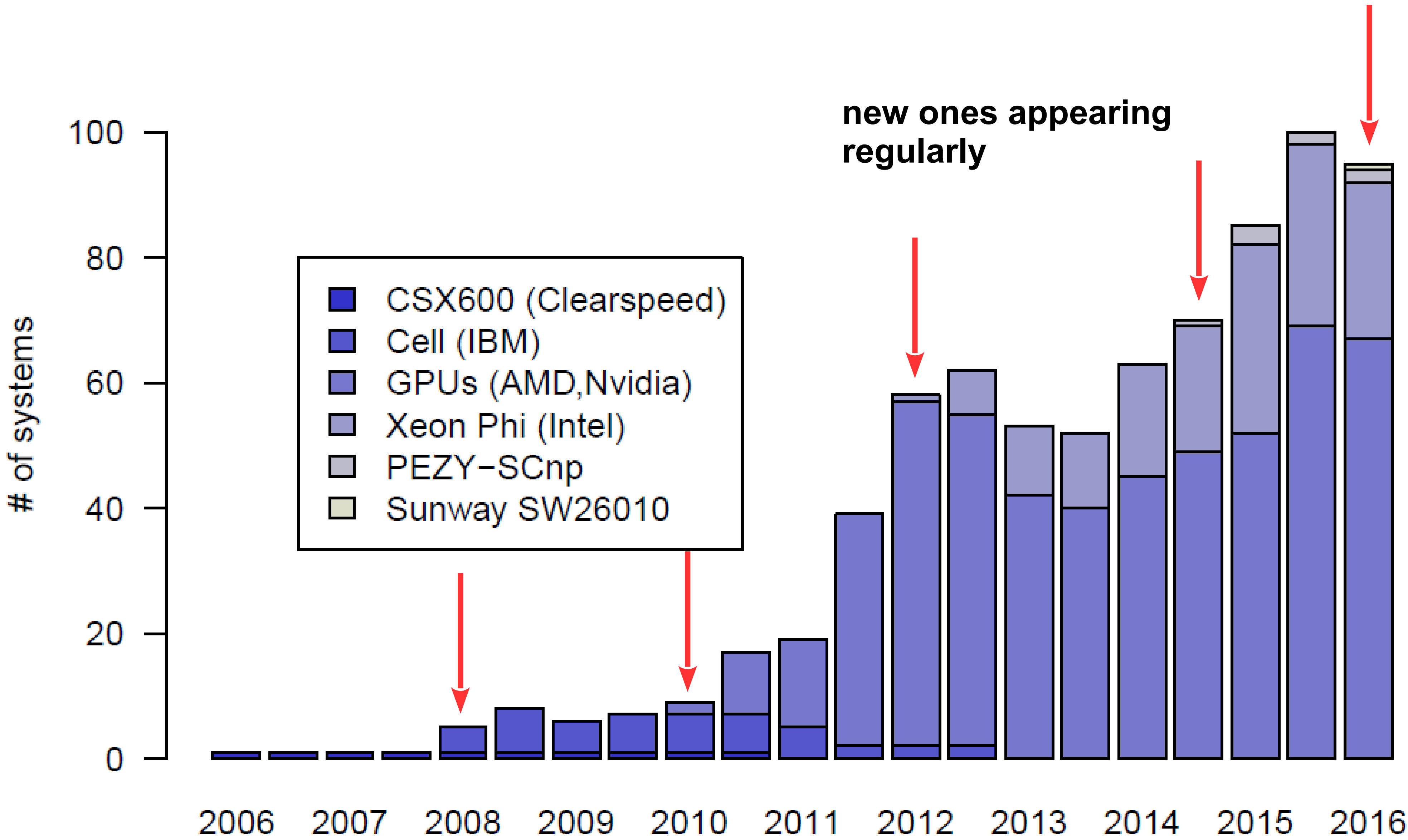
Top 500 with parallel accelerators



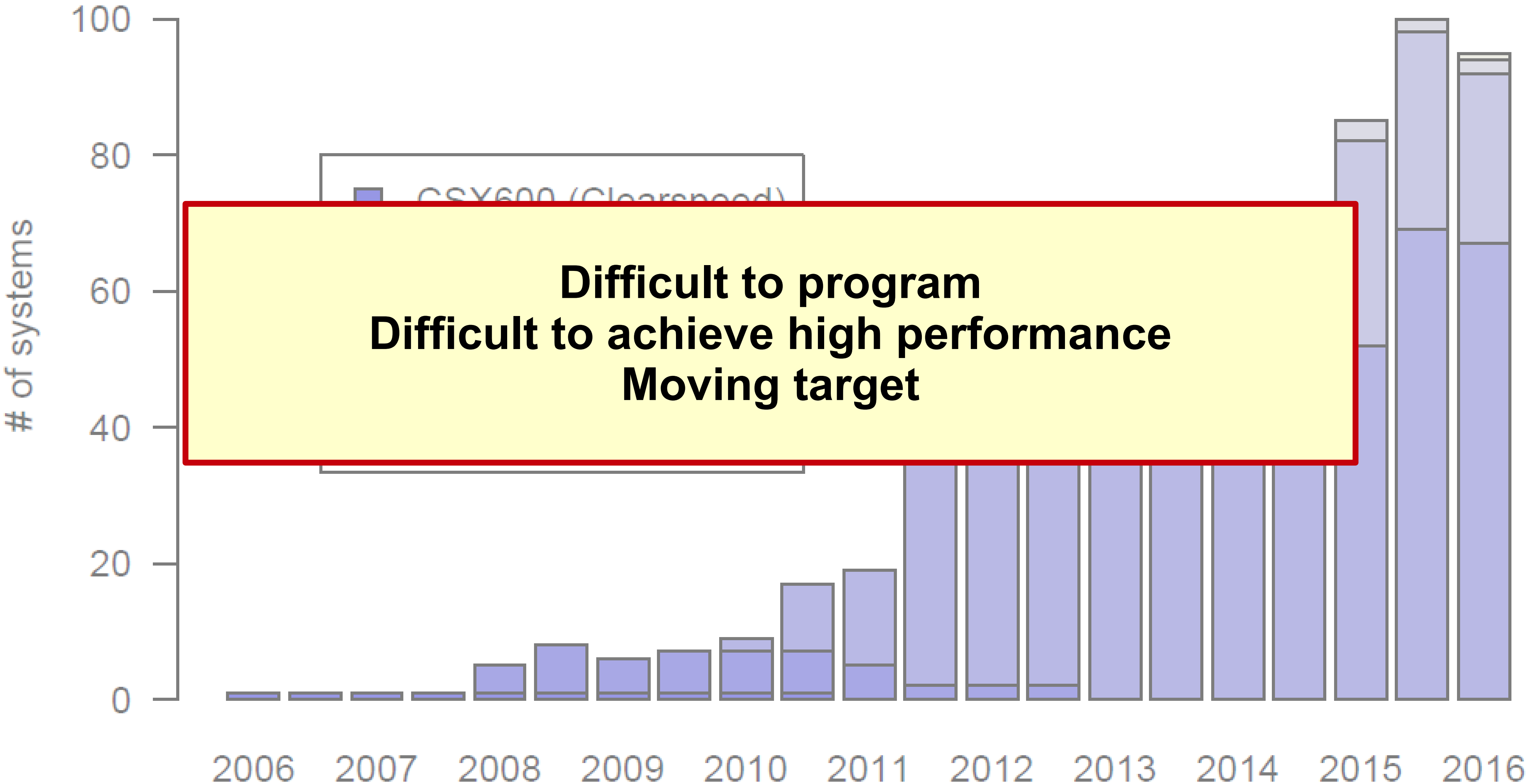
Top 500 with parallel accelerators



Top 500 with parallel accelerators



Top 500 with parallel accelerators

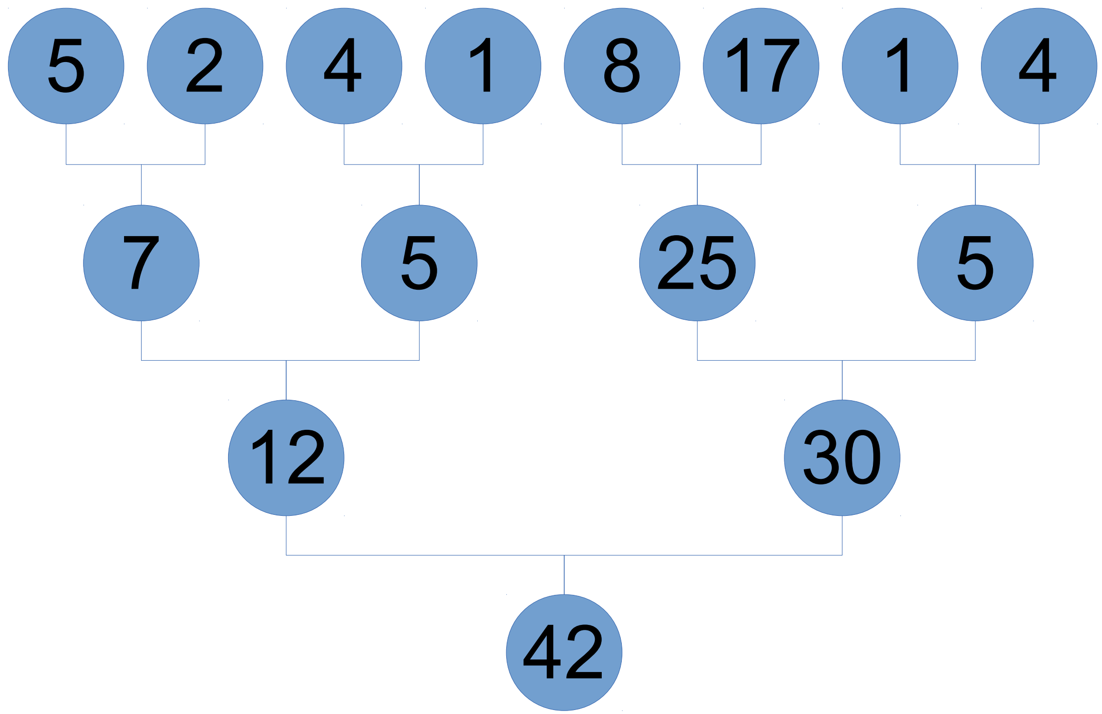


Difficult to program
Difficult to achieve high performance
Moving target

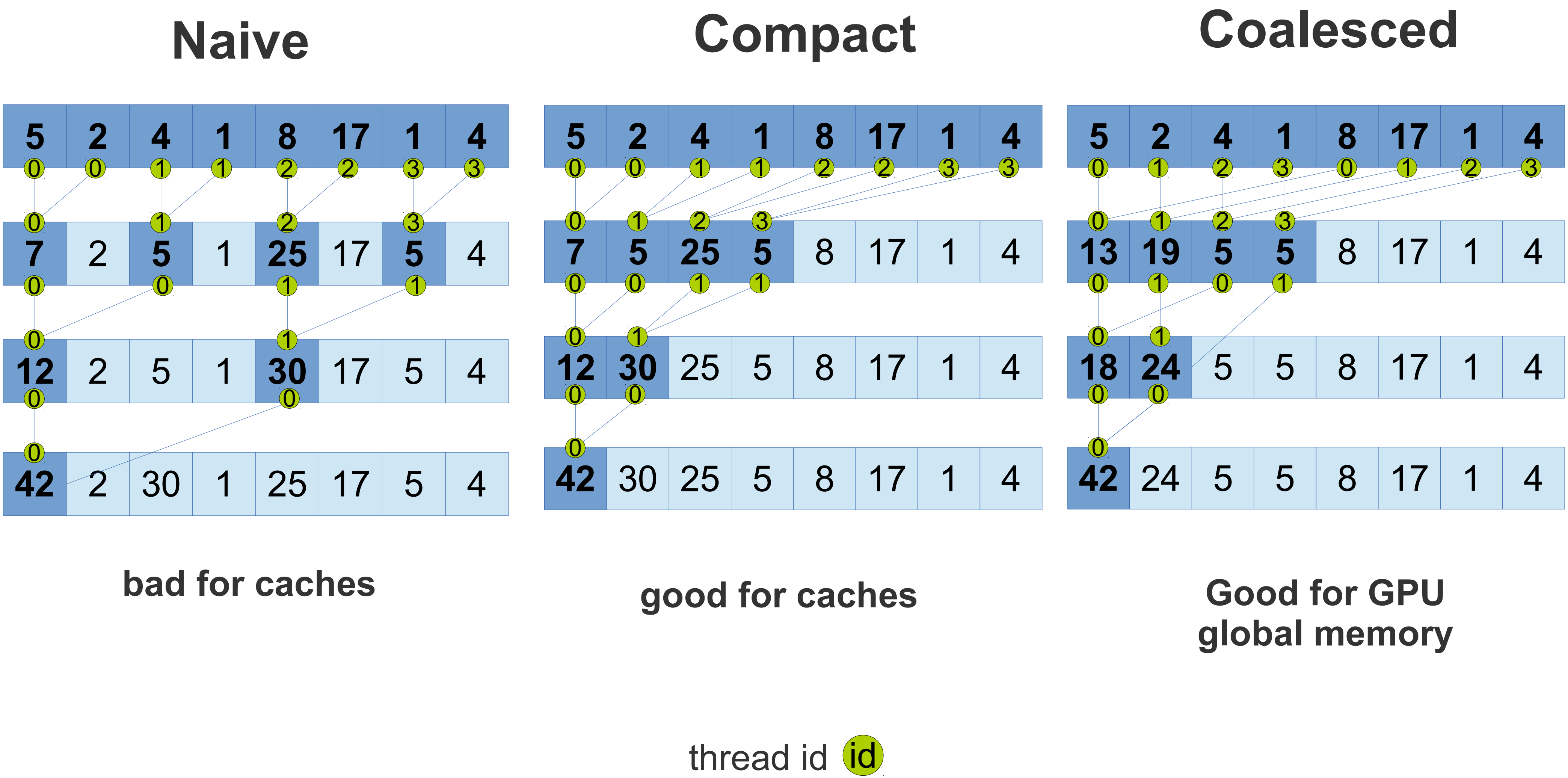
Optimising for accelerators is hard

Example:
Parallel Array Sum on GPU

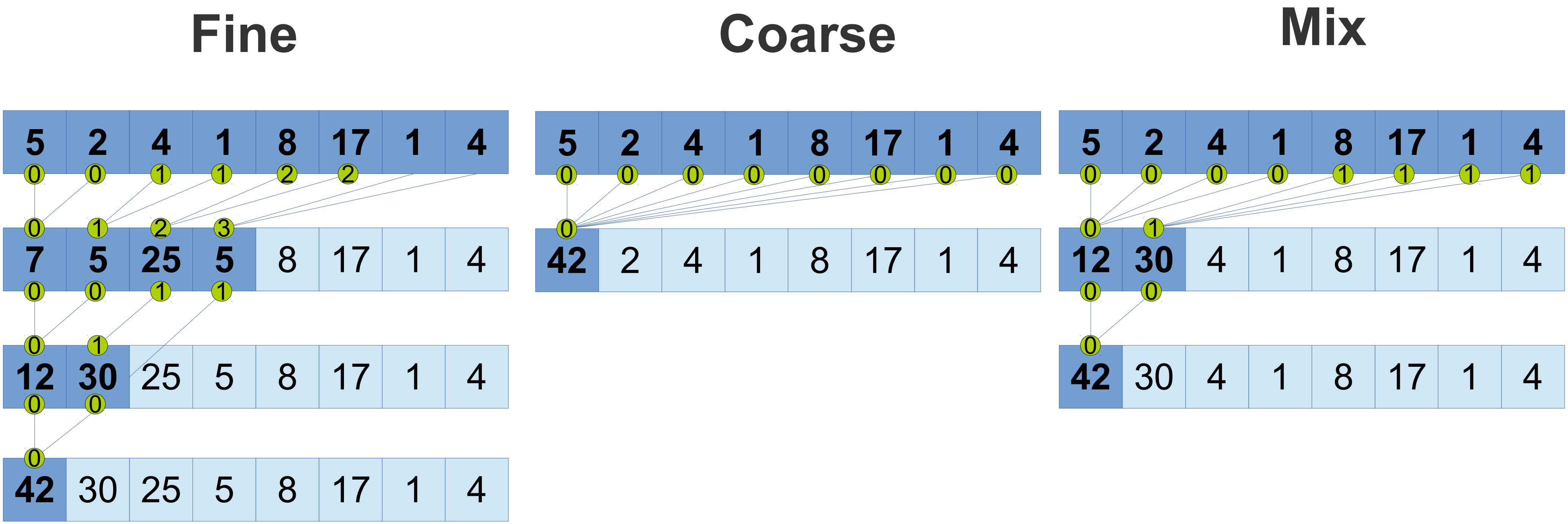
Tree-based parallel array sum



Memory accesses



Thread mapping



Basic Implementation

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

  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];
}
```

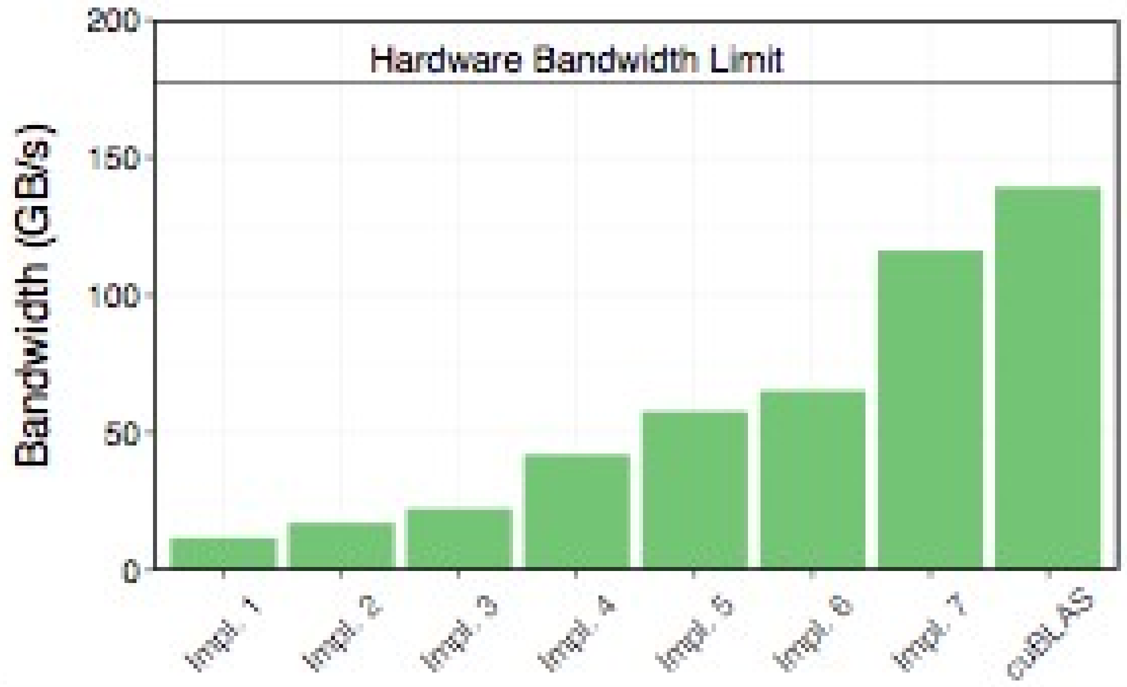
- Optimising OpenCL kernels is hard
  - Need to understand target hardware
- Moving target
  - Hardware keeps changing

Fully Optimized Implementation (Nvidia)

```
kernel
void reduce(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];
}
```

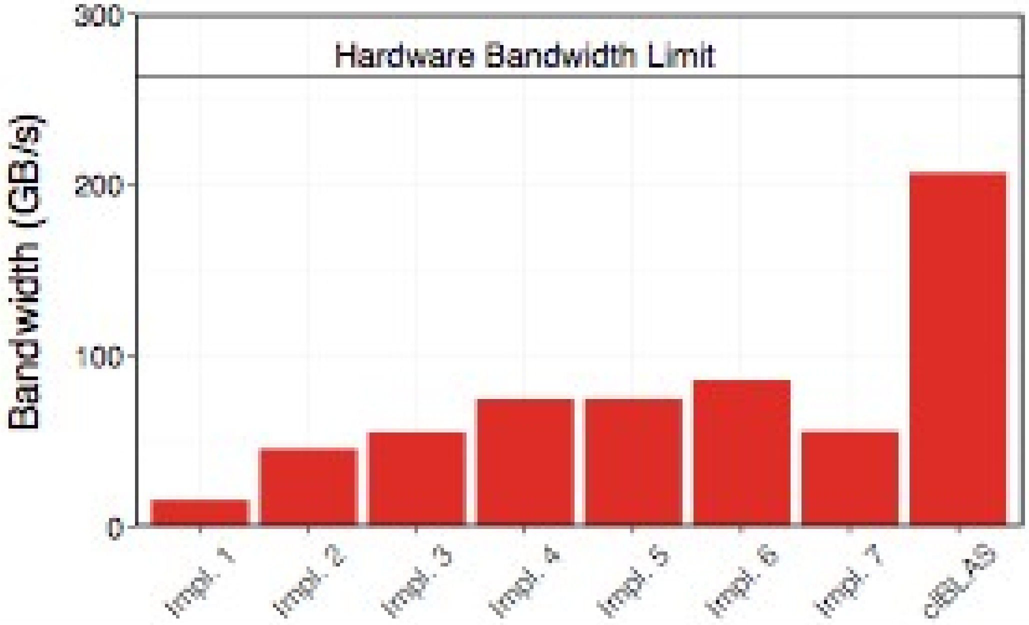
10x improvement for optimised code



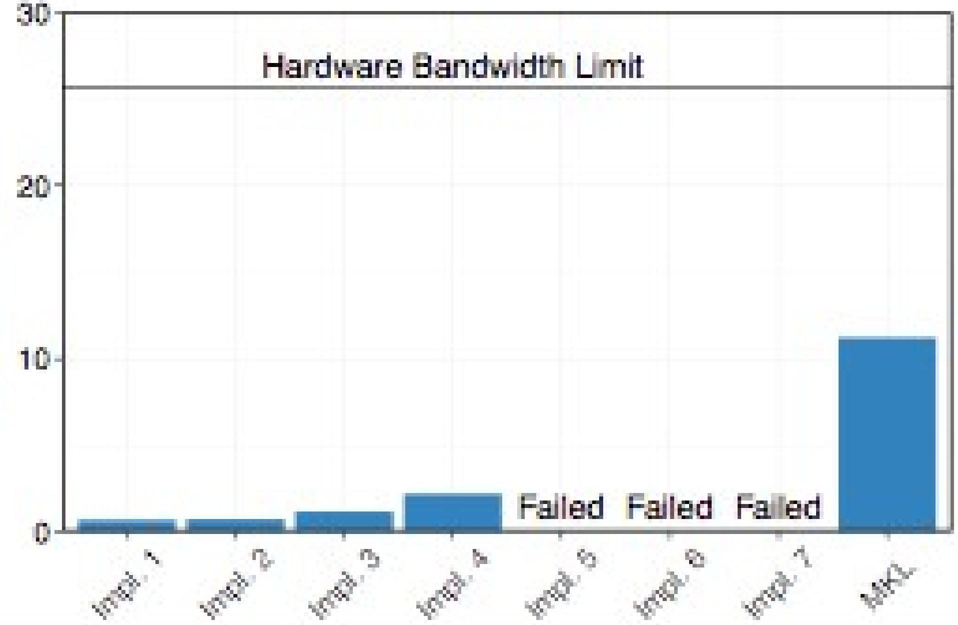
(a) Nvidia's GTX 480 GPU.

Nvidia GPU

Unfortunately, performance is not portable



(b) AMD's HD 7970 GPU.



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

AMD GPU

Intel CPU

How to achieve performance portability?

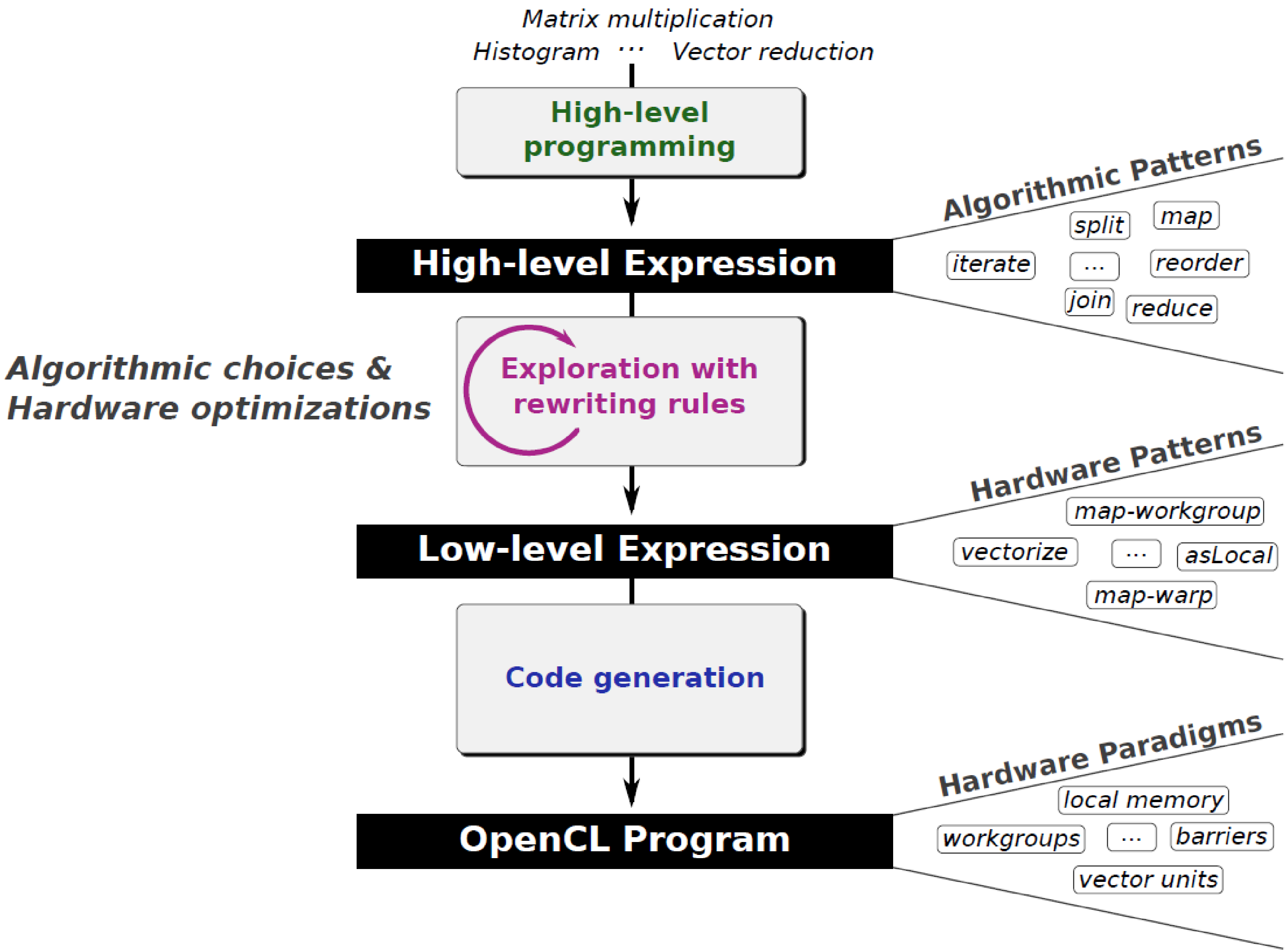
State-of-the-art:

hand-written implementation (maybe parametric) for each device!

The Lift approach:

- a language to express parallel portion of programs
- optimisations and decisions expressed as rewrite rules

Generating Performance Portable Code using Rewrite Rules



High-level expression

```
def add3(int x) = x + 3
def vectorAdd = map(add3)
```

rewrite rules

Low-level expression

```
def vectorAdd = join ( map-workgroup(
                    join o map-local(
                        vect-4(add3)
                    ) o asVector-4
                ) o split-1024)
```

code generation

OpenCL kernel

```
int4 add3(int4 x) { return x + 3; }

Kernel void map_add(global int* in,out, int len) {

    // division into workgroup by chuncks of 1024
    for (int i=get_group_id; i < len/1024; i+=get_num_groups) {
        global int* grp_in = in+(i*1024);
        global int* grp_out = in+(i*1024);

        // division into threads by chunks of 4
        for (int j=get_local_id; j < 1024/4; j+=get_local_size) {
            global int* lcl_in = grp_in+(j*4);
            global int* lcl_out = grp_out+(j*4);

            // vectorization with vector width of 4
            global int4* in_vec4 = (int4*) lcl_in;
            global int4* out_vec4 = (int4*) lcl_out;
            *out_vec4 = add3(*in_vec4);
        }
    }
}
```

Functional World

High-level expression

```
def add3(int x) = x + 3
def vectorAdd = map(add3)
```

rewrite rules

Low-level expression

```
def vectorAdd = join ( map-workgroup(
                    join o map-local(
                        vect-4(add3)
                    ) o asVector-4
                ) o split-1024)
```

code generation

OpenCL kernel

```
int4 add3(int4 x) { return x + 3; }

Kernel void map_add(global int* in,out, int len) {

    // division into workgroup by chuncks of 1024
    for (int i=get_group_id; i < len/1024; i+=get_num_groups) {
        global int* grp_in = in+(i*1024);
        global int* grp_out = in+(i*1024);

        // division into threads by chunks of 4
        for (int j=get_local_id; j < 1024/4; j+=get_local_size) {
            global int* lcl_in = grp_in+(j*4);
            global int* lcl_out = grp_out+(j*4);

            // vectorization with vector width of 4
            global int4* in_vec4 = (int4*) lcl_in;
            global int4* out_vec4 = (int4*) lcl_out;
            *out_vec4 = add3(*in_vec4);
        }
    }
}
```

Functional World

High-level expression

```
def add3(int x) = x + 3
def vectorAdd = map(add3)
```

rewrite rules

Low-level expression

```
def vectorAdd = join ( map-workgroup(
                   join o map-local(
                       vect-4(add3)
                   ) o asVector-4
               ) o split-1024)
```

code generation

Imperative World

OpenCL kernel

```
int4 add3(int4 x) { return x + 3; }

Kernel void map_add(global int* in,out, int len) {

    // division into workgroup by chuncks of 1024
    for (int i=get_group_id; i < len/1024; i+=get_num_groups) {
        global int* grp_in = in+(i*1024);
        global int* grp_out = in+(i*1024);

        // division into threads by chunks of 4
        for (int j=get_local_id; j < 1024/4; j+=get_local_size) {
            global int* lcl_in = grp_in+(j*4);
            global int* lcl_out = grp_out+(j*4);

            // vectorization with vector width of 4
            global int4* in_vec4 = (int4*) lcl_in;
            global int4* out_vec4 = (int4*) lcl_out;
            *out_vec4 = add3(*in_vec4);
        }
    }
}
```

Functional Programming

- Focus on the **what** rather than the **how**

- Imperative program

```
float sum(float* input, int length)
{
  float accumulator = 0;
  for(int i = 0; i < length; i++)
    accumulator += input[i];
  return accumulator;
}
```

- Functional Program

```
reduce (+,0, input)
```

Algorithmic Patterns
(or skeletons)

Functional Algorithmic Primitives

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

High-level Programs

scal(a, vec) = map(*a, vec)

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

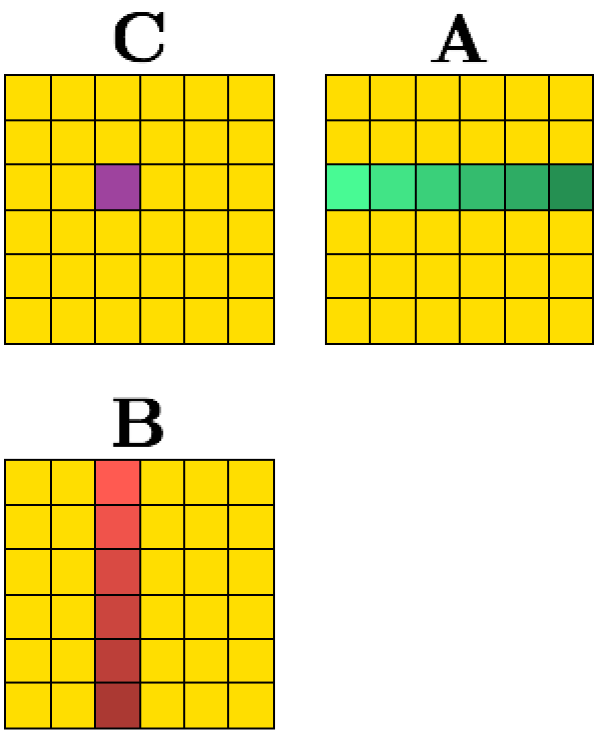
dotProduct(x, y) = reduce(+, 0, map(*, zip(x, y)))

gemv(mat, x, y, a, b) =

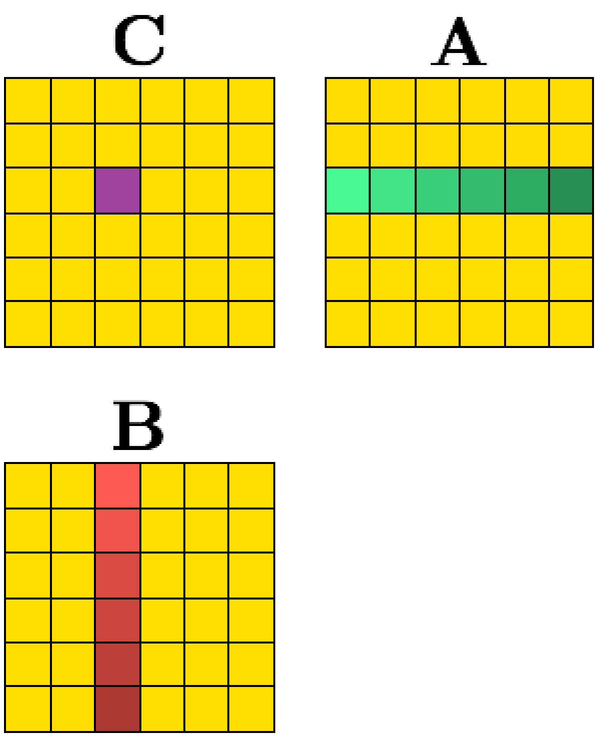
```
map(+, zip(
    map(scal(a) o dotProduct(x), mat),
    scal(b, y) ) )
```

Case study:

Matrix-multiplication



Matrix-multiplication expressed functionally



High-level functional expression

```
A x B =
    map(rowA →
        map(colB →
            Reduce(+) o Map(x) o
            Zip(rowA, colB)
        , transpose(B))
    , A)
```

How to explore the implementation space?

Algorithmic Rewrite Rules
(algebra of parallelism)

- Provably correct rewrite rules
- Express algorithmic implementation choices

Algorithmic Rewrite Rules
(algebra of parallelism)

- Provably correct rewrite rules
- Express algorithmic implementation choices

Split-join rule:

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

Algorithmic Rewrite Rules
(algebra of parallelism)

- 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:

$$map\ f \circ map\ g \rightarrow map\ (f \circ g)$$

Algorithmic Rewrite Rules
(algebra of parallelism)

- 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:

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

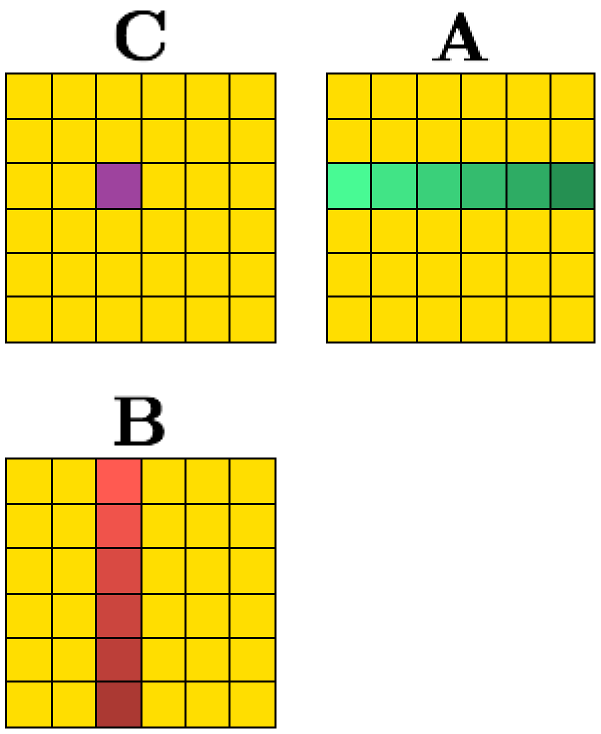
$$reducePart\ f\ z \rightarrow reducePart\ f\ z \circ reorder$$

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

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

...

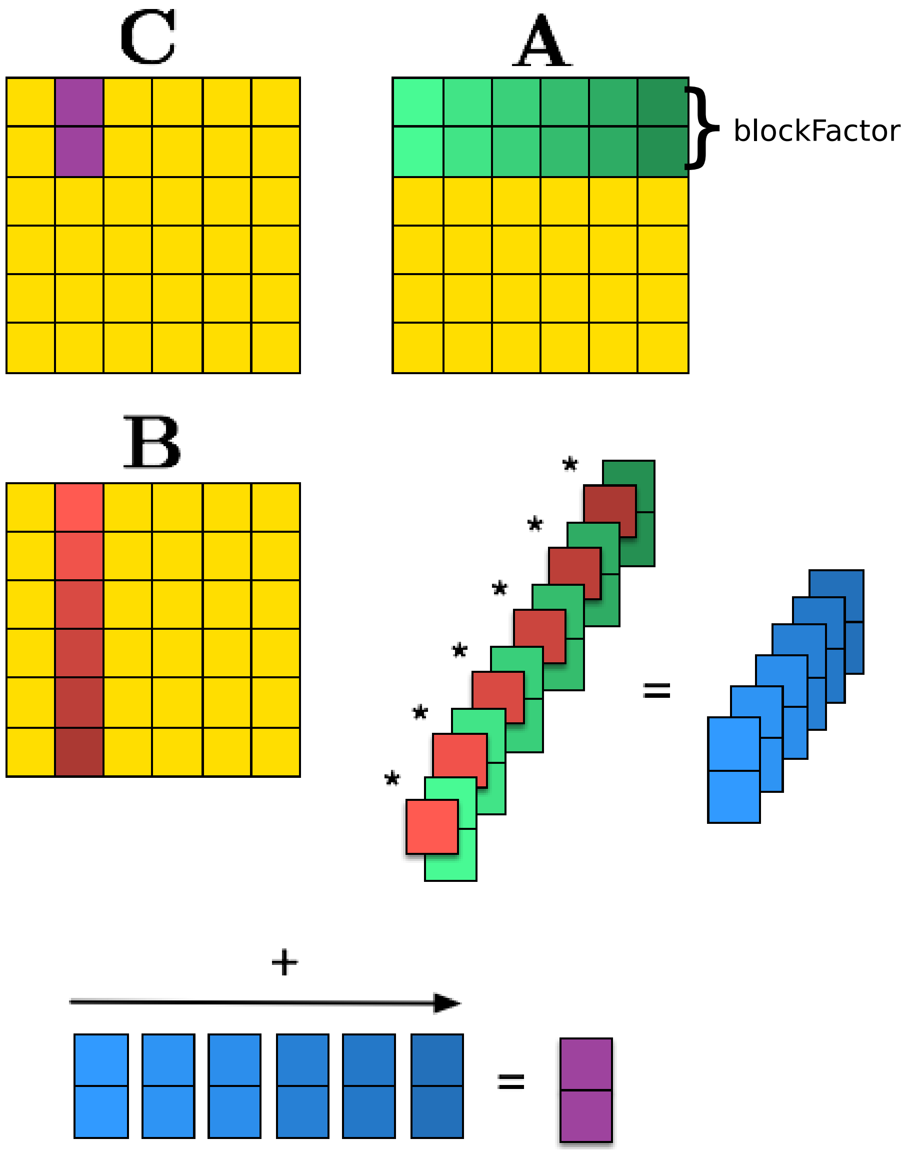
Matrix-multiplication example



High-level functional expression

```
A x B =
    map(rowA →
        map(colB →
            Reduce(+) o Map(x) o
            Zip(rowA, colB)
        , transpose(B))
    , A)
```

OpenCL implementation with Register Blocking



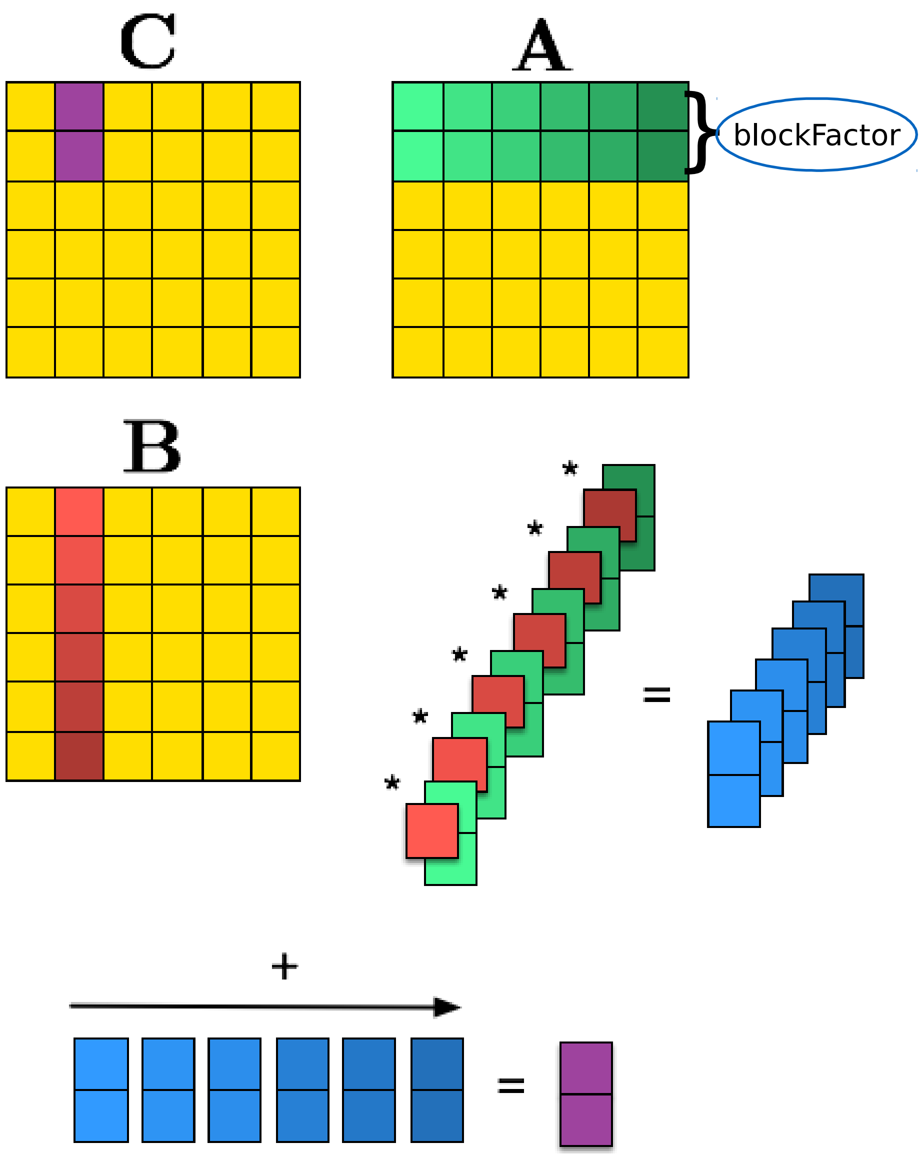

```
kernel void KERNEL(
  const global float* restrict A,
  const global float* restrict B,
  global float* C, int K, int M, int N)
{
  float acc[blockFactor];

  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];
    }
  }
}
```

OpenCL implementation with Register Blocking



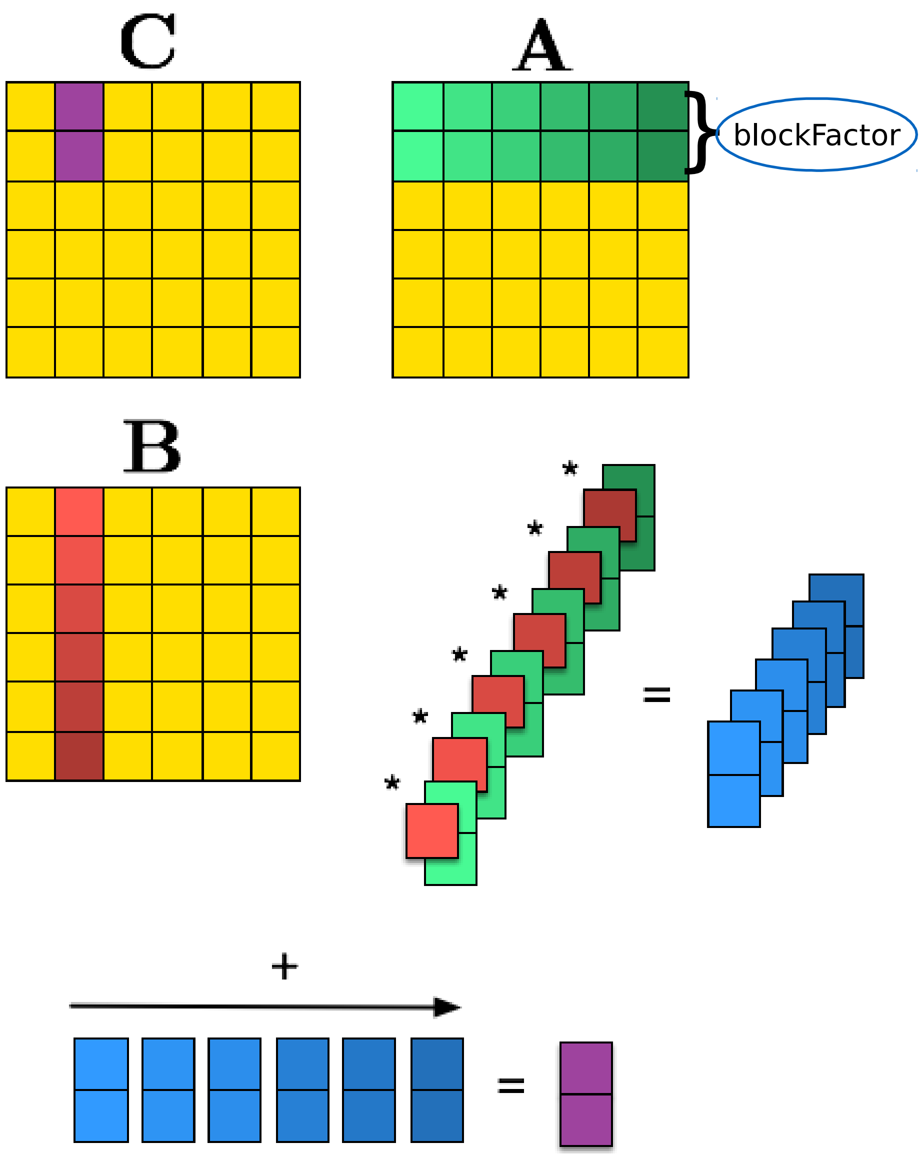

```
kernel void KERNEL(
  const global float* restrict A,
  const global float* restrict B,
  global float* C, int K, int M, int N)
{
  float acc[blockFactor];

  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];
    }
  }
}
```

OpenCL implementation with 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];
    }
  }
}
```

Register Blocking as a series of rewrites

Starting point

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

Register Blocking as a series of rewrites

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

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

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

Register Blocking as a series of rewrites

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

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

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

Register Blocking as a series of rewrites

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

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

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

Register Blocking as a series of rewrites

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

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

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

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

Register Blocking as a series of rewrites

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

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

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

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

Register Blocking as a series of rewrites

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

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

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

Register Blocking as a series of rewrites

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

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

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

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

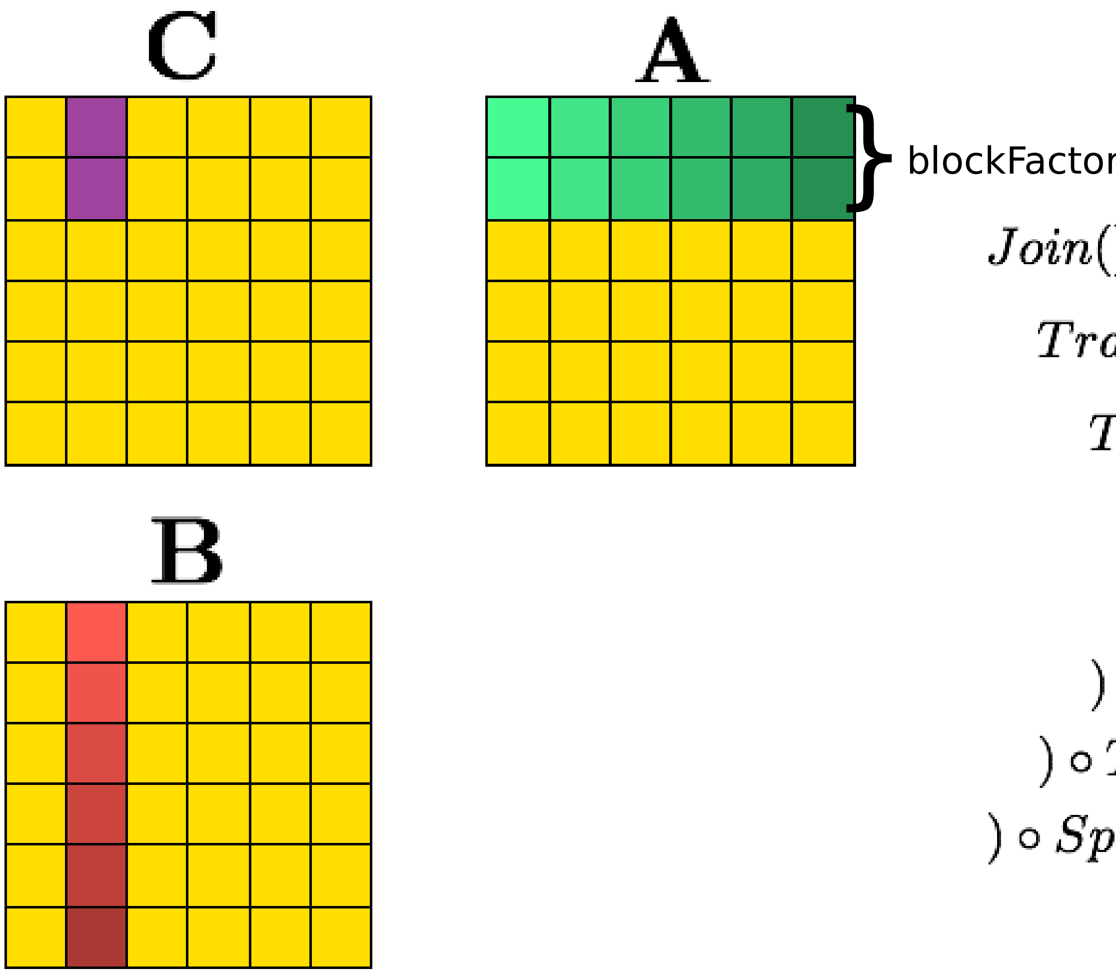
Register Blocking as a series of rewrites

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

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

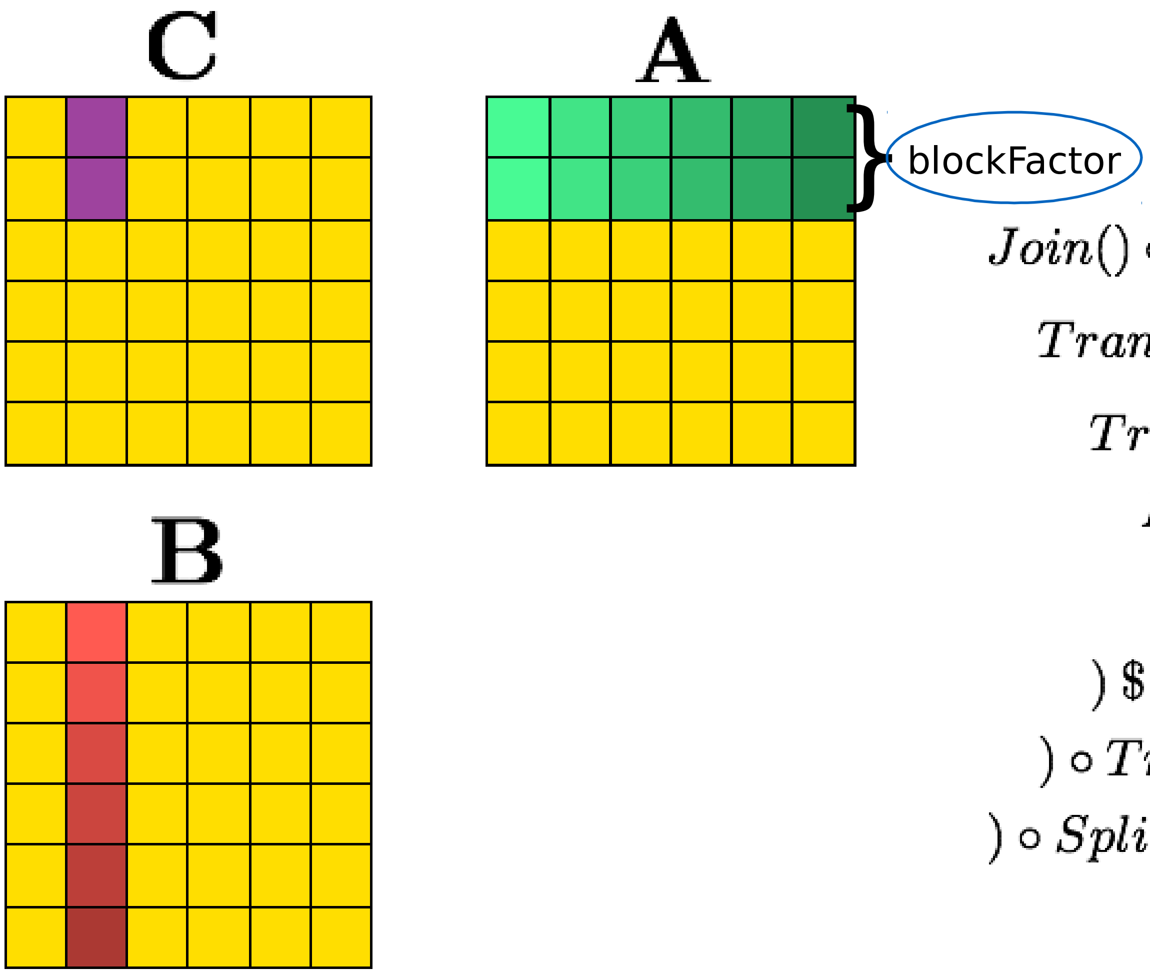
$$Map(f) \circ Map(g) \Rightarrow Map(f \circ g)$$

Register Blocking expressed functionally



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

Register Blocking expressed functionally



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

1

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

1

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

2

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

1

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

2

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

3

```
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];
    }
  }
}
```

Job almost done! now need to “map” parallelism

1

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

2

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

3

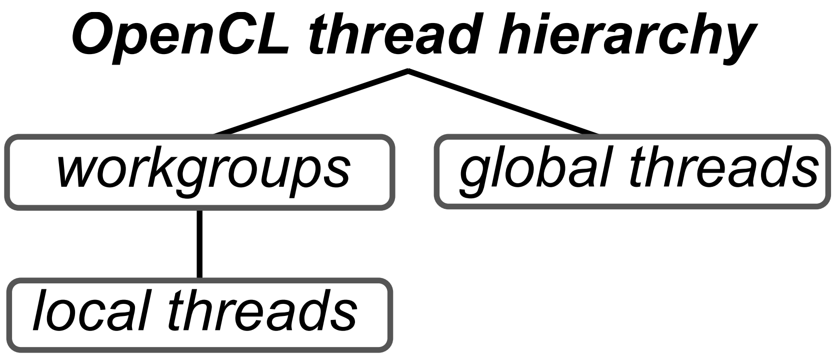
```
kernel void KERNEL(
  const global float* restrict A,
  const global float* restrict B,
  global float* C, int K, int M, int N)
{
  float acc[blockFactor];

  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];
    }
  }
}
```

Mapping Parallelism



map-global

map-workgroup

map-local

map-sequential

Mapping Parallelism

```
map (x => x+3, input)
```

```
mapGlobal (x => x+3, input)
```

...

```
mapSequential (x => x+3, input)
```

OpenCL Code generator

```
for (uint i=get_global_id;
     i<n;
     i+= get_global_size) {
   output[i] = input[i]+3;
}
```

```
for (uint i=0; i<n; i+= 1) {
   output[i] = input[i]+3;
}
```

→ Pattern based code generator

map-global (f,input)

```
for (uint i=get_global_id;
     i<n;
     i+= get_global_size) {
   output[i] = f(input[i]);
}
```

...

map-sequential (f,input)

```
for (uint i=0; i<n; i++) {
   output[i] = f(input[i]);
}
```

reduce-sequential (f,z,input)

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

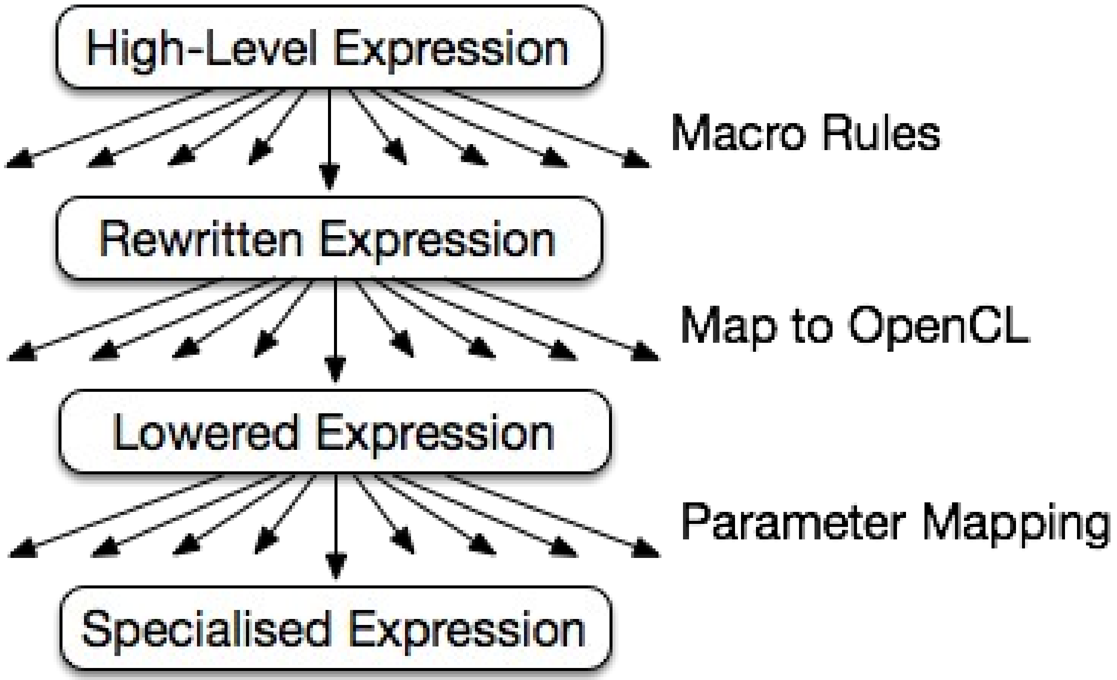
Memory
- **toLocal**
- **toGlobal**

Vectorisation
- **vect^n**
- **asScalar**
- **asVector**

Data partitioning
- **split**
- **join**

Rewrite rules define a search space

Exploration process



Heuristics

Macro Rules:

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

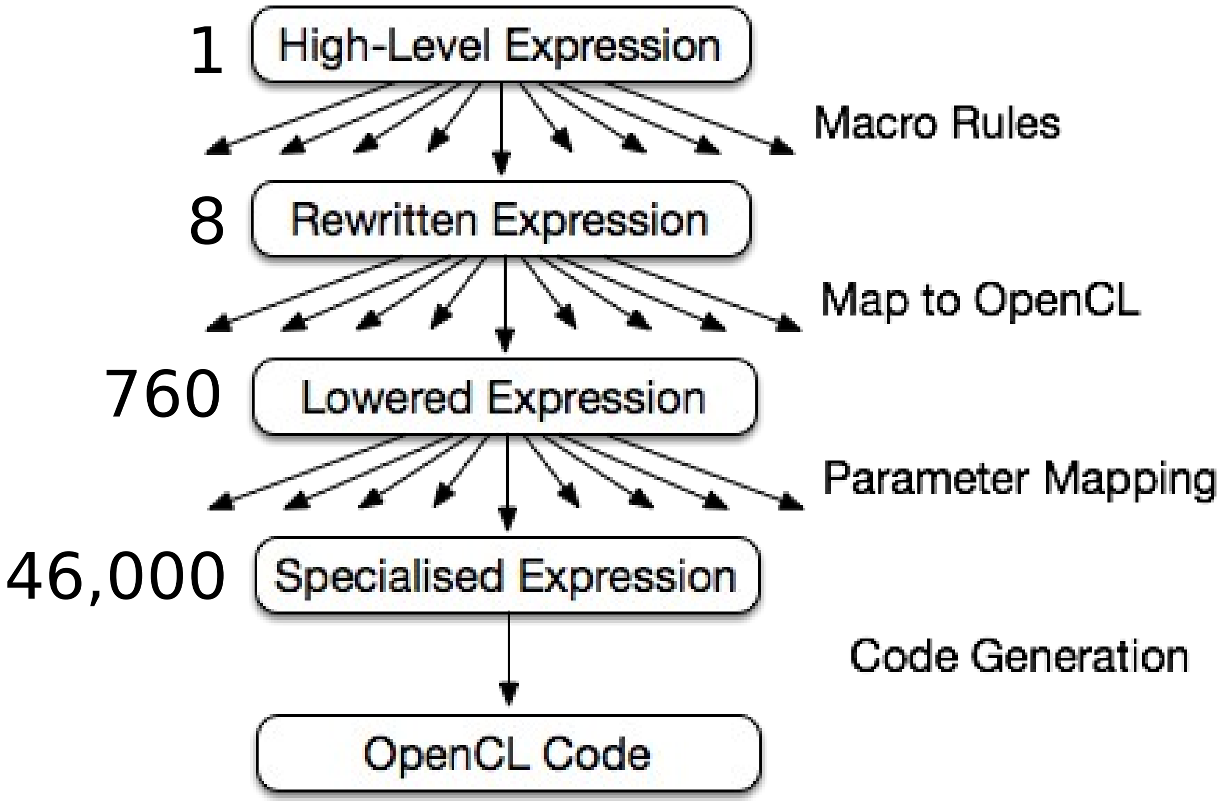
Mapping to OpenCL:

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

Parameter Tuning:

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

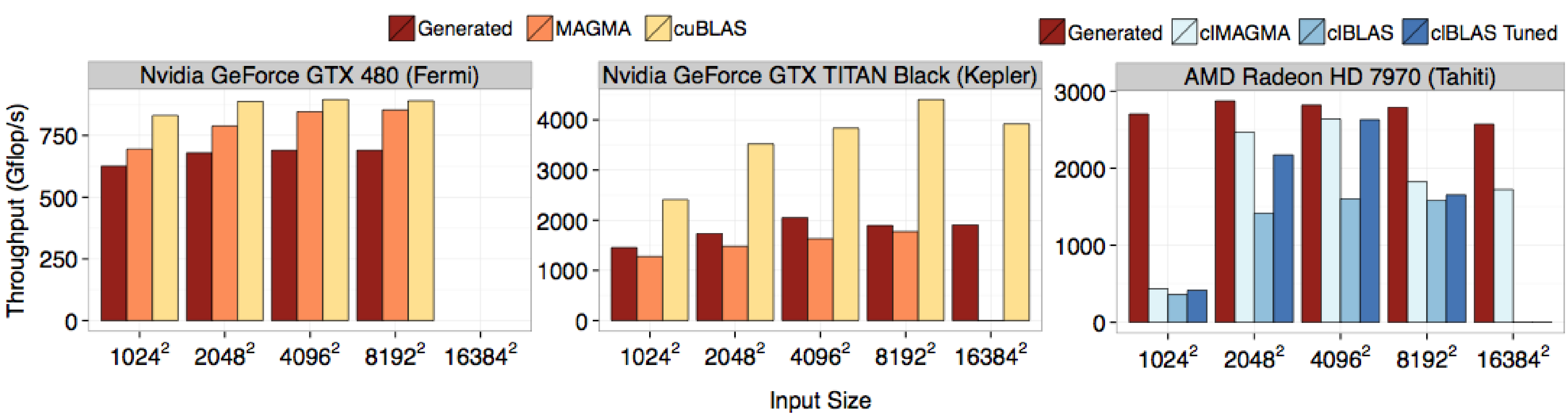
Exploration in numbers for matrix multiplication



Performance Portability Achieved

Compiler input:

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



Summary

- Language for expressing parallelism
  - functional in nature, could be targeted by DSL
- Rewrite rules define a search space
  - formalisation of algorithmic and optimisation choices
- High performance achieved:
  - on par with highly-tuned code
- Works for other applications: e.g. Nbody simulation, K-means clustering, ...
- Future work: Stencil, Convolution (Neural Network)

if you want to know more: www.lift-project.org

partially funded by: Oracle Labs Google