Domain Specific Languages
and rewriting-based optimisations for
performance-portable parallel programming

Michel Steuwer

THE UNIVERSITY of EDINBURGH

Domain Specific Languages

- Definition by Paul Hudak:
  "A programming language tailored specifically for an application domain"
- DSLs are *not* general purpose programming language
- Capture the semantics of a particular application domain
- Raise level of abstraction (often *declarative* not *imperative*)

Examples of Domain Specific Languages

SQL

```
SELECT Book.title AS Title,
       count(*) AS Authors
 FROM  Book
 JOIN  Book_author
   ON  Book.isbn = Book_author.isbn
 GROUP BY Book.title;
```

make

```
all: hello

hello: main.o factorial.o hello.o
    g++ main.o factorial.o hello.o -o hello

main.o: main.cpp
    g++ -c main.cpp

factorial.o: factorial.cpp
    g++ -c factorial.cpp

hello.o: hello.cpp
    g++ -c hello.cpp

clean:
    rm *o hello
```

HTML

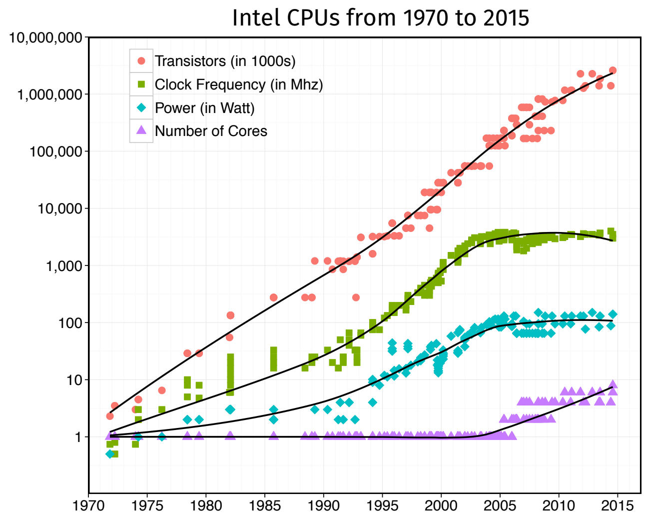
VHDL

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;    -- for the unsigned type

entity COUNTER is
  generic (
    WIDTH : in natural := 32);
  port (
    RST   : in std_logic;
    CLK   : in std_logic;
    LOAD  : in std_logic;
    DATA  : in std_logic_vector(WIDTH-1 downto 0);
    Q     : out std_logic_vector(WIDTH-1 downto 0));
end entity COUNTER;

architecture RTL of COUNTER is
  signal CNT : unsigned(WIDTH-1 downto 0);
begin
  process(RST, CLK) is
  begin
    if RST = '1' then
      CNT <= (others => '0');
    elsif rising_edge(CLK) then
      if LOAD = '1' then
        CNT <= unsigned(DATA); -- type is converted to unsigned
      else
        CNT <= CNT + 1;
      end if;
    end if;
  end process;

  Q <= std_logic_vector(CNT); -- type is converted back to std_logic_vector
end architecture RTL;
```

shell scripts

```
#!/bin/sh
if [ $(id -u) != "0" ]; then
    echo "You must be the superuser to
          run this script" >&2
    exit 1
fi
```


Parallelism everywhere: The Many-Core Era



Inspired by Herb Sutter *"The Free Lunch is Over: A Fundamental Turn Towards Concurrency in Software"*

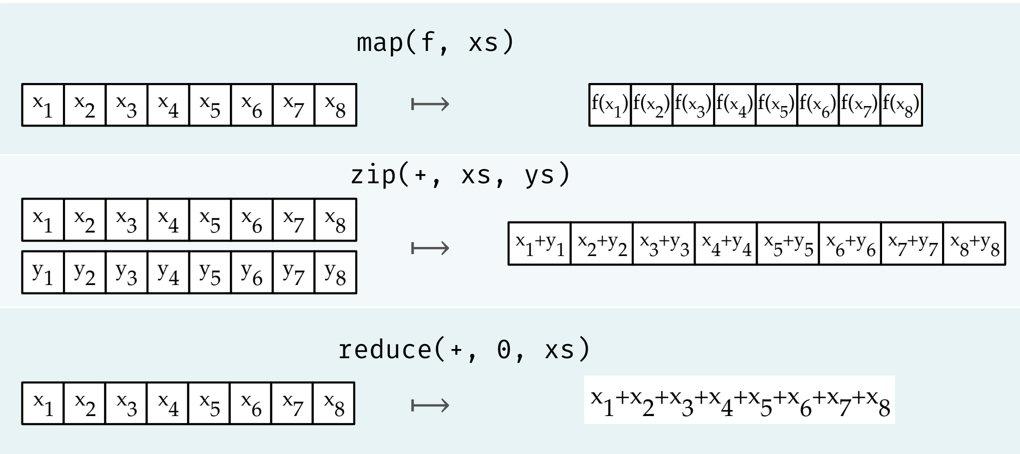
Challenges of Parallel Programming

- *Threads* are the dominant parallel programming model for multi-core architectures
- Concurrently executing threads can modify shared data, leading to:
  - race conditions
  - need for mutual execution and synchronisation
  - deadlocks
  - non-determinism
- Writing correct parallel programs is extremely challenging

Structured *Parallel* Programming
aka: "Threads Considered Harmful"

- Dijkstra's: "GO TO" Considered Harmful let to structured programming
  - Raise the level of abstraction by capturing common *patterns:*
    - E.g. use '`if A then B else C`' instead of multiple `goto` statements
- Murray Cole at Edinburgh invented *Algorithmic Skeletons:*
  - special higher-order functions which describe the "computational skeleton" of a parallel algorithm
  - E.g. use `Dc indivisible split join f` instead of a custom divide-and-conquer implementation with threads
- Algorithmic Skeletons are structured *parallel* programming and raise the level of abstraction over threads
  - No race conditions and no need for explicit synchronisation
  - No deadlocks
  - Deterministic

Examples of Algorithmic Skeletons



- Algorithmic Skeletons have a parallel semantics
- Every (parallel) execution order to compute the result is valid
- Complexity of parallelism is hidden by the skeleton

DSLs for Parallel Programming with Algorithmic Skeletons

- There exist numerous implementations of algorithmic skeletons libraries
  - The Edinburgh Skeleton Library (eSkel): C, MPI
  - FastFlow and Muesli: C++, multi-core CPU, MPI, GPU
  - SkePU, **SkelCL**: C++, GPU
  - Accelerate: Haskell, GPU
  - …
- Libraries from industry implementing similar concepts:
  - Intel's Threading Building Blocks (TBB)
  - Nvidia's Thrust Library

SkelCL by Example

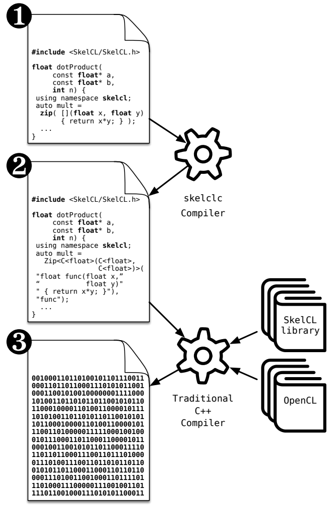
$$\text{dotProduct } A\ B = \text{reduce } (+)\ 0 \circ \text{zip } (\times)\ A\ B$$

```
#include <SkelCL/SkelCL.h>
#include <SkelCL/Zip.h>
#include <SkelCL/Reduce.h>
#include <SkelCL/Vector.h>

float dotProduct(const float* a, const float* b, int n) {
  using namespace skelcl;
  skelcl::init( 1_device.type(deviceType::ANY) );

  auto mult =    zip([](float x, float y) { return x*y; });
  auto sum  = reduce([](float x, float y) { return x+y; }, 0);

  Vector<float> A(a, a+n); Vector<float> B(b, b+n);

  Vector<float> C = sum( mult(A, B) );

  return C.front();
}
```


From SkelCL to OpenCL

❶

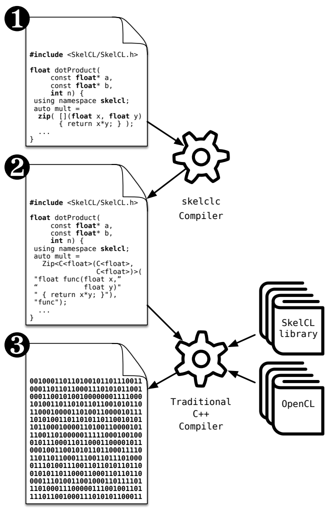
```
#include <SkelCL/SkelCL.h>
#include <SkelCL/Zip.h>
#include <SkelCL/Reduce.h>
#include <SkelCL/Vector.h>


float dotProduct(const float* a, const float* b, int n) {
  using namespace skelcl;
  skelcl::init( 1_device.type(deviceType::ANY) );

  auto mult =    zip([](float x, float y) { return x*y; });
  auto sum  = reduce([](float x, float y) { return x+y; }, 0);

  Vector<float> A(a, a+n); Vector<float> B(b, b+n);

  Vector<float> C = sum( mult(A, B) );

  return C.front();
}
```



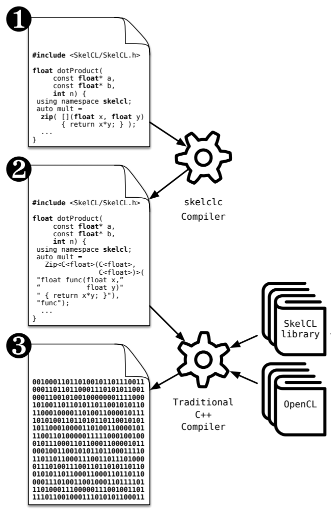
From SkelCL to OpenCL

❷

```
#include <SkelCL/SkelCL.h>
#include <SkelCL/Zip.h>
#include <SkelCL/Reduce.h>
#include <SkelCL/Vector.h>


float dotProduct(const float* a, const float* b, int n) {
  using namespace skelcl;
  skelcl::init( 1_device.type(deviceType::ANY) );

  auto mult = Zip<Container<float>(Container<float>,
                                   Container<float>)>(
   Source("float func(float x, float y) {return x*y;}"));
  auto sum  = Reduce<Vector<float>(Vector<float>)>(
   Source("float func(float x, float y) {return x+y;}"), "0");

  Vector<float> A(a, a+n); Vector<float> B(b, b+n);

  Vector<float> C = sum( mult(A, B) );

  return C.front();
}
```



From SkelCL to OpenCL

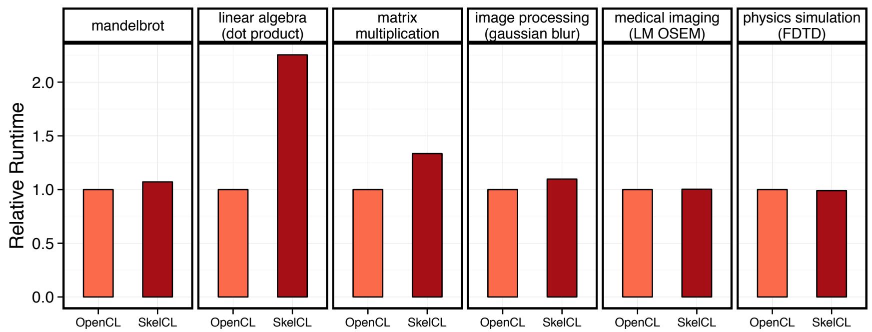


Implementations of Algorithmic Skeletons in OpenCL

```
// reduce kernel
```

```
// zip kernel

typedef float T0; typedef float T1;
typedef float T2;

kernel void ZIP(const global T0* left,
                const global T1* right,
                      global T2* out,
                const int size) {
  size_t id = get_global_id(0);
  if (id < size)
    out[id] = func(left[id], right[id]);
}
```

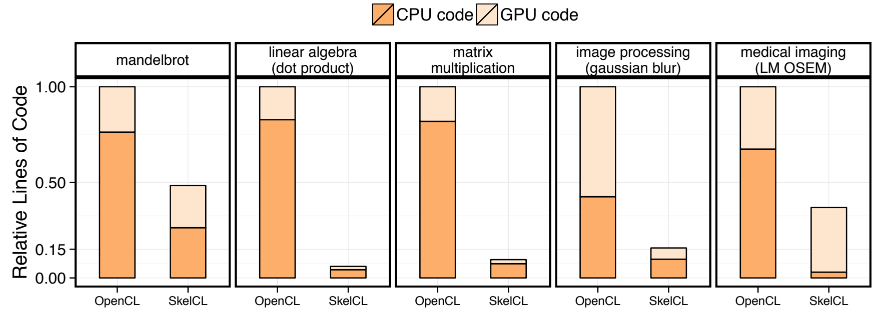
SkelCL Evaluation — Performance



SkelCL performance close to native OpenCL code!

(Exception: dot product … we will address this later)

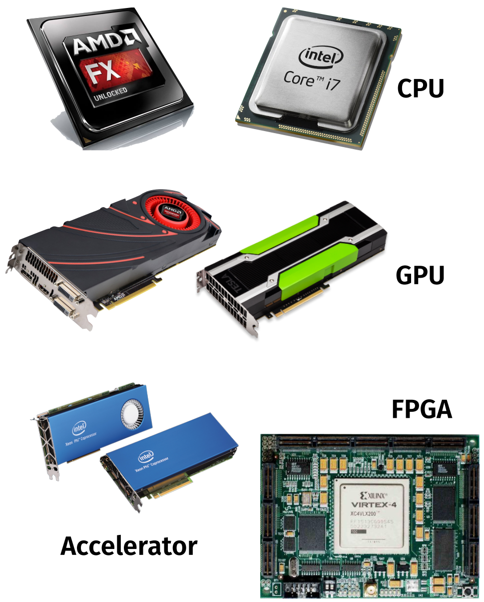
SkelCL Evaluation — Productivity



SkelCL programs are significantly shorter!
Common advantage of Domain Specific Languages!

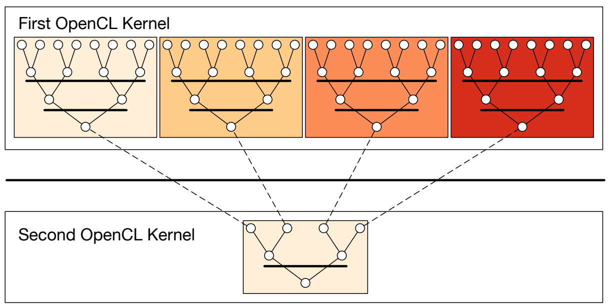
The Performance Portability Problem

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



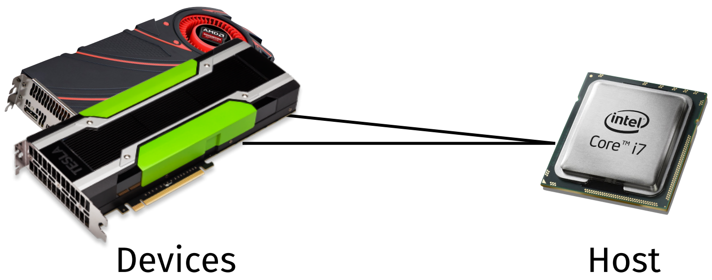
Case Study: Parallel Reduction in OpenCL

- Summing up all values of an array (== *reduce* skeleton)
- Comparison of 7 implementations by Nvidia
- Investigating complexity and efficiency of optimisations



OpenCL

- Parallel programming language for GPUs, multi-core CPUs
- Application is executed on the *host* and offloads computations to *devices*
- Computations on the device are expressed as *kernels:*
  - functions executed in parallel
- Usual problems of deadlocks, race conditions, ...



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*

Unoptimised Implementation Parallel Reduction

```
kernel void reduce0(global float* g_idata, global float* g_odata,
                    unsigned int n, local float* l_data) {
  unsigned int tid = get_local_id(0);
  unsigned int i   = get_global_id(0);
  l_data[tid] = (i < n) ? g_idata[i] : 0;
  barrier(CLK_LOCAL_MEM_FENCE);
  // do reduction in local memory
  for (unsigned int s=1; s < get_local_size(0); s*= 2) {
    if ((tid % (2*s)) == 0) {
      l_data[tid] += l_data[tid + s];
    }
    barrier(CLK_LOCAL_MEM_FENCE);
  }
  // write result for this work-group to global memory
  if (tid == 0) g_odata[get_group_id(0)] = l_data[0];
}
```


Avoid Divergent Branching

```
kernel void reduce1(global float* g_idata, global float* g_odata,
                    unsigned int n, local float* l_data) {
  unsigned int tid = get_local_id(0);
  unsigned int i   = get_global_id(0);
  l_data[tid] = (i < n) ? g_idata[i] : 0;
  barrier(CLK_LOCAL_MEM_FENCE);

  for (unsigned int s=1; s < get_local_size(0); s*= 2) {
    // continuous work-items remain active
    int index = 2 * s * tid;
    if (index < get_local_size(0)) {
      l_data[index] += l_data[index + s];
    }
    barrier(CLK_LOCAL_MEM_FENCE);
  }
  if (tid == 0) g_odata[get_group_id(0)] = l_data[0];
}
```

Avoid Interleaved Addressing

```
kernel void reduce2(global float* g_idata, global float* g_odata,
                    unsigned int n, local float* l_data) {
  unsigned int tid = get_local_id(0);
  unsigned int i   = get_global_id(0);
  l_data[tid] = (i < n) ? g_idata[i] : 0;
  barrier(CLK_LOCAL_MEM_FENCE);

  // process elements in different order
  // requires commutativity
  for (unsigned int s=get_local_size(0)/2; s>0; s>>=1) {
    if (tid < s) {
      l_data[tid] += l_data[tid + s];
    }
    barrier(CLK_LOCAL_MEM_FENCE);
  }
  if (tid == 0) g_odata[get_group_id(0)] = l_data[0];
}
```


Increase Computational Intensity per Work-Item

```
kernel void reduce3(global float* g_idata, global float* g_odata,
                    unsigned int n, local float* l_data) {
  unsigned int tid = get_local_id(0);
  unsigned int i = get_group_id(0) * (get_local_size(0)*2)
                                   + get_local_id(0);
  l_data[tid] = (i < n) ? g_idata[i] : 0;
  // performs first addition during loading
  if (i + get_local_size(0) < n)
    l_data[tid] += g_idata[i+get_local_size(0)];
  barrier(CLK_LOCAL_MEM_FENCE);

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


Avoid Synchronisation inside a Warp

```
kernel void reduce4(global float* g_idata, global float* g_odata,
                    unsigned int n, local volatile float* l_data) {
  unsigned int tid = get_local_id(0);
  unsigned int i = get_group_id(0) * (get_local_size(0)*2)
                                   + get_local_id(0);
  l_data[tid] = (i < n) ? g_idata[i] : 0;
  if (i + get_local_size(0) < n)
    l_data[tid] += g_idata[i+get_local_size(0)];
  barrier(CLK_LOCAL_MEM_FENCE);

  # pragma unroll 1
  for (unsigned int s=get_local_size(0)/2; s>32; s>>=1) {
    if (tid < s) { l_data[tid] += l_data[tid + s]; }
    barrier(CLK_LOCAL_MEM_FENCE); }

  // this is not portable OpenCL code!
  if (tid < 32) {
    if (WG_SIZE >= 64) { l_data[tid] += l_data[tid+32]; }
    if (WG_SIZE >= 32) { l_data[tid] += l_data[tid+16]; }
    if (WG_SIZE >= 16) { l_data[tid] += l_data[tid+ 8]; }
    if (WG_SIZE >=  8) { l_data[tid] += l_data[tid+ 4]; }
    if (WG_SIZE >=  4) { l_data[tid] += l_data[tid+ 2]; }
    if (WG_SIZE >=  2) { l_data[tid] += l_data[tid+ 1]; } }
  if (tid == 0) g_odata[get_group_id(0)] = l_data[0]; }
```

Complete Loop Unrolling

```
kernel void reduce5(global float* g_idata, global float* g_odata,
                    unsigned int n, local volatile float* l_data) {
  unsigned int tid = get_local_id(0);
  unsigned int i = get_group_id(0) * (get_local_size(0)*2)
                                   + get_local_id(0);
  l_data[tid] = (i < n) ? g_idata[i] : 0;
  if (i + get_local_size(0) < n)
    l_data[tid] += g_idata[i+get_local_size(0)];
  barrier(CLK_LOCAL_MEM_FENCE);

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

Fully Optimised Implementation

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

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


Case Study Conclusions

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

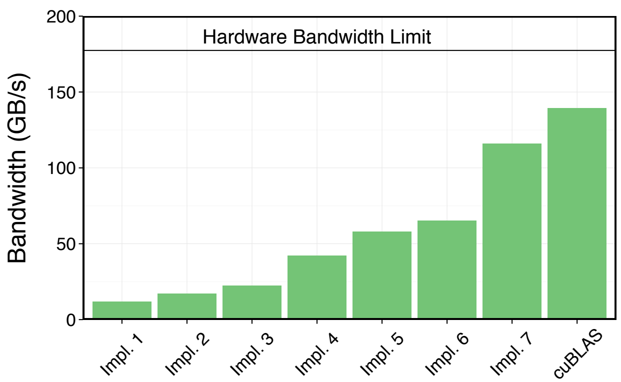
```
kernel
void reduce0(global float* g_idata,
             global float* g_odata,
             unsigned int n,
             local float* l_data) {
  unsigned int tid = get_local_id(0);
  unsigned int i   = get_global_id(0);
  l_data[tid] = (i < n) ? g_idata[i] : 0;
  barrier(CLK_LOCAL_MEM_FENCE);

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

Unoptimized Implementation

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

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

Fully Optimized Implementation

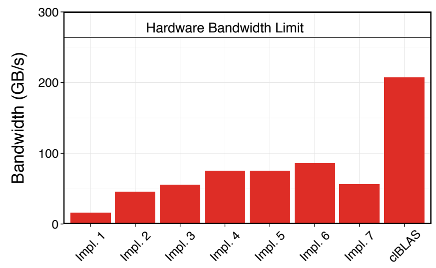
Performance Results Nvidia



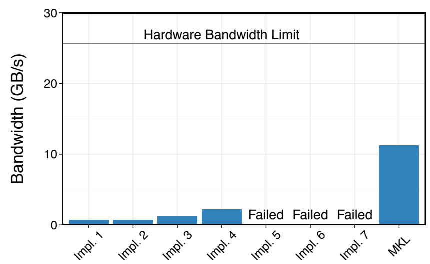
(a) Nvidia's GTX 480 GPU.

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

Performance Results AMD and Intel



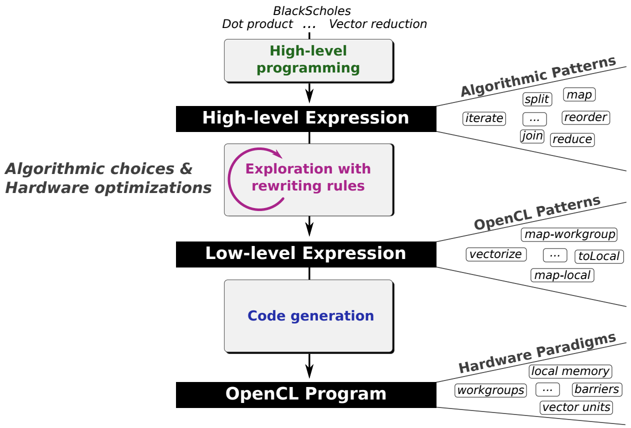
(b) AMD's HD 7970 GPU.



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

- ... unfortunately, optimisations in OpenCL are not portable!
- **Challenge**: how to achieving portable performance?

Generating Performance Portable Code using Rewrite Rules



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

Michel Steuwer, Christian Fensch, Sam Lindley, Christophe Dubach:
"Generating performance portable code using rewrite rules:
from high-level functional expressions to high-performance OpenCL code."
ICFP 2015

Example Parallel Reduction ③

① $\text{vecSum} = \mathit{reduce}\ (+)\ 0$

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
```

```
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];
}
```


Example Parallel Reduction ③

① $\text{vecSum} = \mathit{reduce}\ (+)\ 0$

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
```

```
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

$$map_{A,B,I} : (A \to B) \to [A]_I \to [B]_I$$

$$zip_{A,B,I} : [A]_I \to [B]_I \to [A \times B]_I$$

$$reduce_{A,I} : ((A \times A) \to A) \to A \to [A]_I \to [A]_1$$

$$split_{A,I} : (n : \text{size}) \to [A]_{n \times I} \to [[A]_n]_I$$

$$join_{A,I,J} : [[A]_I]_J \to [A]_{I \times J}$$

$$iterate_{A,I,J} : (n : size) \to ((m : size) \to [A]_{I \times m} \to [A]_m) \to [A]_{I^n \times J} \to [A]_J$$

$$reorder_{A,I} : [A]_I \to [A]_I$$

① High-Level Programs

$$scal = \lambda\ a.map\ (*a)$$

$$asum = reduce\ (+)\ 0 \circ map\ abs$$

$$dot = \lambda\ xs\ ys.(reduce\ (+)\ 0 \circ map\ (*))\ (zip\ xs\ ys)$$

$$gemv = \lambda\ mat\ xs\ ys\ \alpha\ \beta.map\ (+)\ (\\ zip\ (map\ (scal\ \alpha \circ dot\ xs)\ mat)\ (scal\ \beta\ ys)\ )$$

Example Parallel Reduction ③

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
```

```
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];
}
```


Example Parallel Reduction ③

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
```

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

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


② Algorithmic Rewrite Rules

- Provably correct rewrite rules
- Express algorithmic implementation choices

Split-join rule:

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

Map fusion rule:

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

Reduce rules:

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

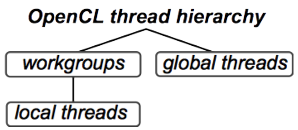
$$reducePart\ f\ z \rightarrow reducePart\ f\ z \circ reorder$$

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

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

② OpenCL Primitives

Primitive	OpenCL concept
$mapGlobal$	Work-items
$mapWorkgroup$, $mapLocal$	Work-groups
$mapSeq$, $reduceSeq$	Sequential implementations
$toLocal$, $toGlobal$	Memory areas
$mapVec$, $splitVec$, $joinVec$	Vectorization



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:

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

Example Parallel Reduction ③

① $\mathrm{vecSum} = reduce\ (+)\ 0$

rewrite rules

code generation

②

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

```
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];
}
```


Example Parallel Reduction ③

① $\mathrm{vecSum} = reduce\ (+)\ 0$

rewrite rules

code generation

②

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

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

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

③ Pattern based OpenCL Code Generation

- Generate OpenCL code for each OpenCL primitive

$mapGlobal\ f\ xs$ →

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

$reduceSeq\ f\ z\ xs$ →

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

⋮ ⋮

Rewrite rules define a space of possible implementations

reduce (+) 0

reduce (+) 0 ∘ *reducePart* (+) 0

Rewrite rules define a space of possible implementations

reduce (+) 0

reduce (+) 0 ∘ *reducePart* (+) 0

reduce (+) 0 ∘ *reducePart* (+) 0 ∘ *reorder*

Rewrite rules define a space of possible implementations

reduce (+) 0

reduce (+) 0 ∘ *reducePart* (+) 0

reduce (+) 0 ∘ *reducePart* (+) 0 ∘ *reorder*

reduce (+) 0 ∘ *iterate* n (*reducePart* (+) 0)

Rewrite rules define a space of possible implementations

$\mathit{reduce}\ (+)\ 0$

$\mathit{reduce}\ (+)\ 0 \circ \mathit{reducePart}\ (+)\ 0$

$\mathit{reduce}\ (+)\ 0 \circ \mathit{reducePart}\ (+)\ 0 \circ \mathit{reorder}$

$\mathit{reduce}\ (+)\ 0 \circ \mathit{iterate}\ \mathrm{n}\ (\mathit{reducePart}\ (+)\ 0)$

$\mathit{reduce}\ (+)\ 0 \circ \mathit{join} \circ \mathit{map}\ (\mathit{reducePart}\ (+)\ 0) \circ \mathit{split}\ \mathrm{n}$

Rewrite rules define a space of possible implementations

$\mathit{reduce}\ (+)\ 0$

$\mathit{reduce}\ (+)\ 0 \circ \mathit{reducePart}\ (+)\ 0$

$\mathit{reduce}\ (+)\ 0 \circ \mathit{reducePart}\ (+)\ 0 \circ \mathit{reorder}$

$\mathit{reduce}\ (+)\ 0 \circ \mathit{iterate}\ \mathrm{n}\ (\mathit{reducePart}\ (+)\ 0)$

$\mathit{reduce}\ (+)\ 0 \circ \mathit{join} \circ \mathit{map}\ (\mathit{reducePart}\ (+)\ 0) \circ \mathit{split}\ \mathrm{n}$

… … …

- Fully automated search for good implementations possible

Search Strategy

- For each node in the tree:
  - Apply one rule and randomly sample subtree
  - Repeat for node with best performing subtree

$\mathit{reduce}\ (+)\ 0 \circ \mathit{reducePart}\ (+)\ 0$

1 apply rule

$\mathit{reduce}\ (+)\ 0 \circ \mathit{reducePart}\ (+)\ 0 \circ \mathit{reorder}$

$\mathit{reduce}\ (+)\ 0 \circ \mathit{iterate}\ \mathrm{n}\ (\mathit{reducePart}\ (+)\ 0)$

$\mathit{reduce}\ (+)\ 0 \circ \mathit{join} \circ \mathit{map}\ (\mathit{reducePart}\ (+)\ 0) \circ \mathit{split}\ \mathrm{n}$

… … …

2 generate code
execute
measure performance

Search Strategy

- For each node in the tree:
  - Apply one rule and randomly sample subtree
  - Repeat for node with best performing subtree

$\mathit{reduce}\ (+)\ 0 \circ \mathit{reducePart}\ (+)\ 0$

1 apply rule

$\mathit{reduce}\ (+)\ 0 \circ \mathit{reducePart}\ (+)\ 0 \circ \mathit{reorder}$

$\mathit{reduce}\ (+)\ 0 \circ \mathit{iterate}\ \mathrm{n}\ (\mathit{reducePart}\ (+)\ 0)$

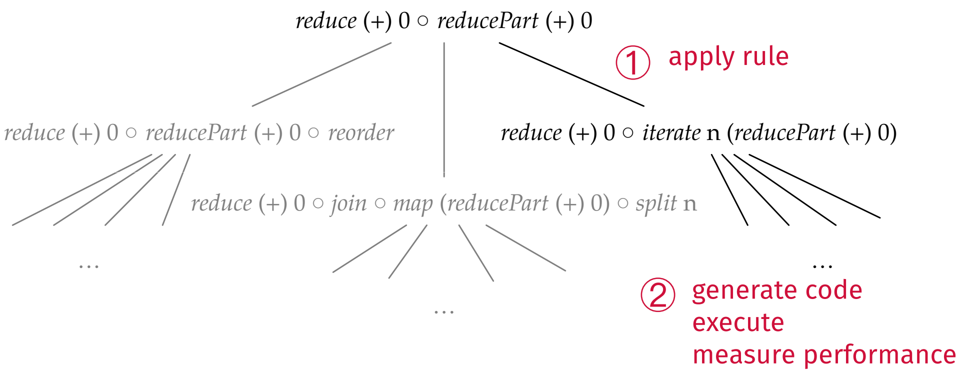
$\mathit{reduce}\ (+)\ 0 \circ \mathit{join} \circ \mathit{map}\ (\mathit{reducePart}\ (+)\ 0) \circ \mathit{split}\ \mathrm{n}$

… … …

2 generate code
execute
measure performance

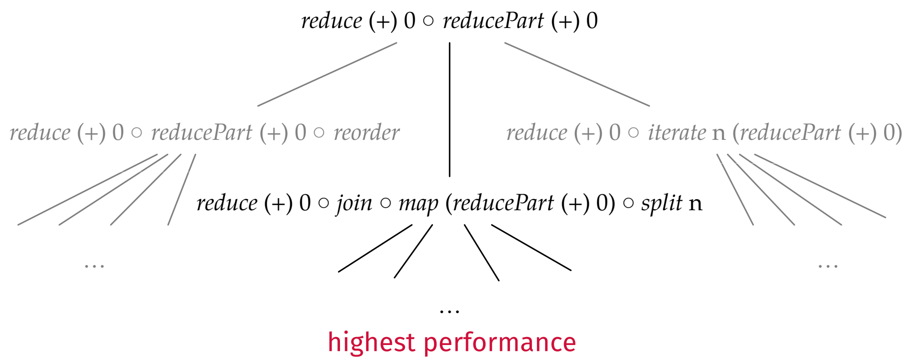
Search Strategy

- For each node in the tree:
  - Apply one rule and randomly sample subtree
  - Repeat for node with best performing subtree



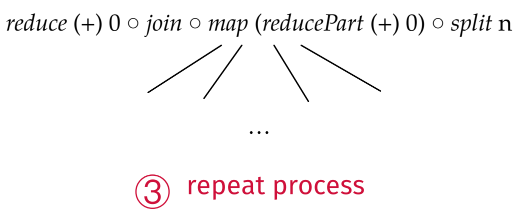
Search Strategy

- For each node in the tree:
  - Apply one rule and randomly sample subtree
  - Repeat for node with best performing subtree



Search Strategy

- For each node in the tree:
  - Apply one rule and randomly sample subtree
  - Repeat for node with best performing subtree



Search Results
Automatically Found Expressions

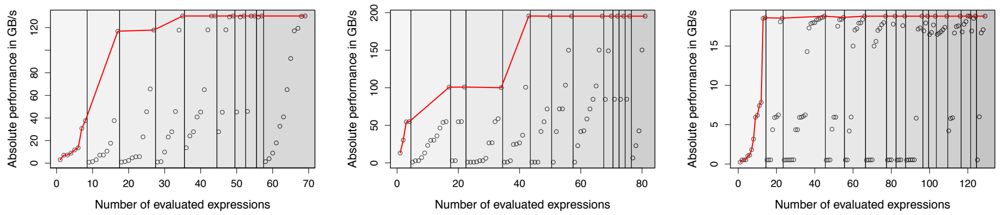
$$asum = reduce\ (+)\ 0 \circ map\ abs$$

$$\Downarrow$$

Nvidia GPU	$\lambda x.(reduceSeq \circ join \circ join \circ mapWorkgroup\ ($ $toGlobal\ (mapLocal\ (reduceSeq\ (\lambda(a,b).\ a + (abs\ b))\ 0)) \circ reorderStride\ 2048$ $) \circ split\ 128 \circ split\ 2048)\ x$
AMD GPU	$\lambda x.(reduceSeq \circ join \circ joinVec \circ join \circ mapWorkgroup\ ($ $mapLocal\ (reduceSeq\ (mapVec\ 2\ (\lambda(a,b).\ a + (abs\ b)))\ 0 \circ reorderStride\ 2048$ $) \circ split\ 128 \circ splitVec\ 2 \circ split\ 4096)\ x$
Intel CPU	$\lambda x.(reduceSeq \circ join \circ mapWorkgroup\ (join \circ joinVec \circ mapLocal\ ($ $reduceSeq\ (mapVec\ 4\ (\lambda(a,b).\ a + (abs\ b)))\ 0$ $) \circ splitVec\ 4 \circ split\ 32768) \circ split\ 32768)\ x$

- Search on: **Nvidia** GTX 480 GPU, **AMD** Radeon HD 7970 GPU, **Intel** Xeon E5530 CPU

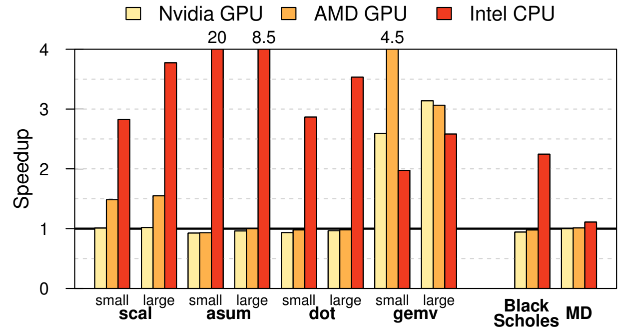
Search Results
Search Efficiency



(a) Nvidia GPU **(b)** AMD GPU **(c)** Intel CPU

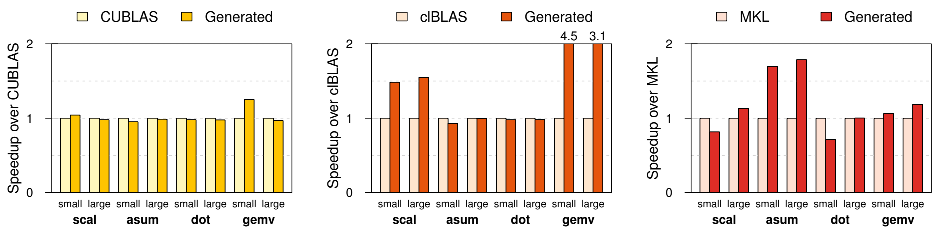
- Overall search on each platform took < 1 hour
- Average execution time per tested expression < 1/2 second

Performance Results
vs. Portable Implementation



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

Performance Results
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

Summary

- DSLs simplify programming but also enable optimisation opportunities
- Algorithmic skeletons allow for structured parallel programming

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

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