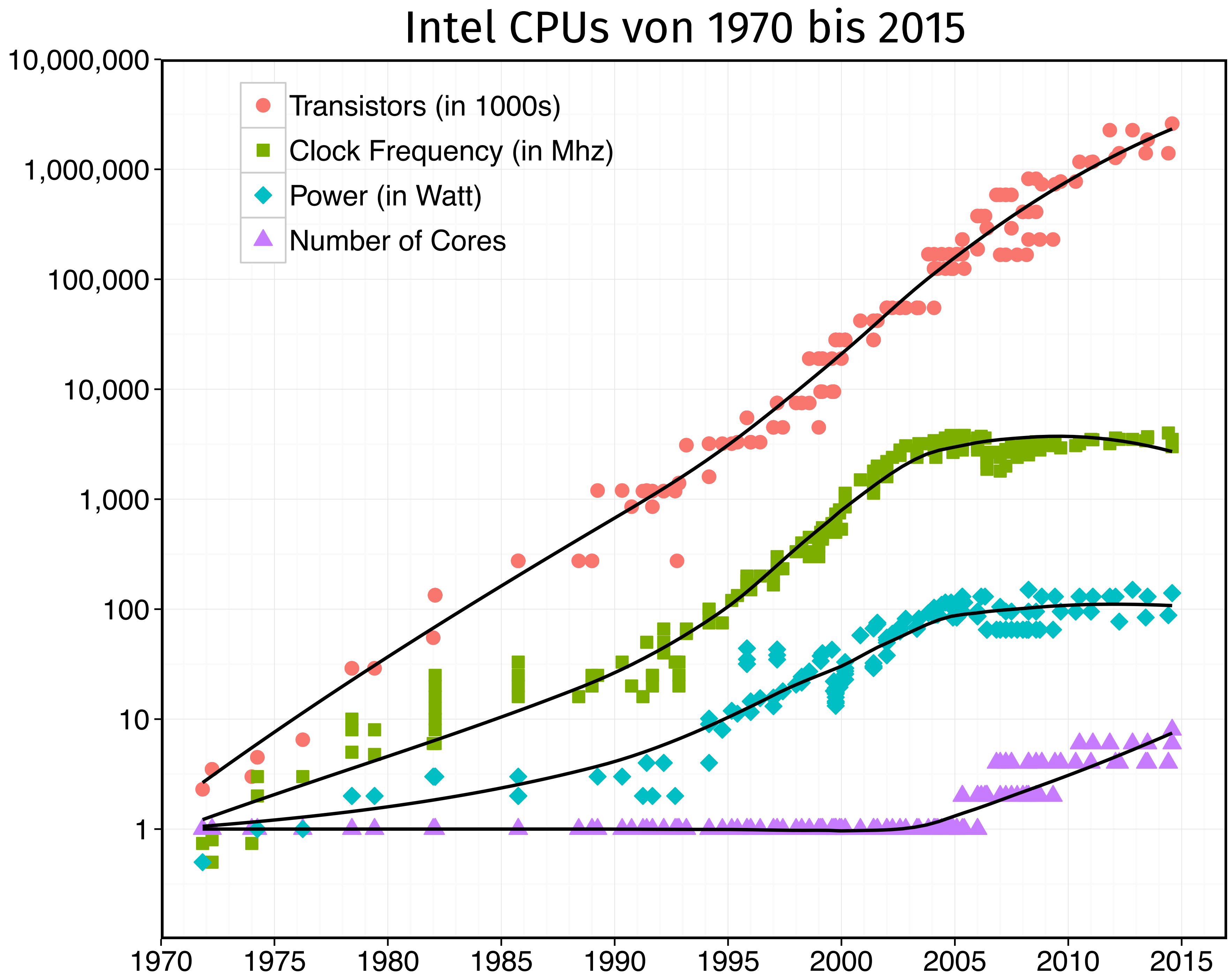
IMPROVING PROGRAMMABILITY AND PERFORMANCE PORTABILITY ON MANY-CORE PROCESSORS

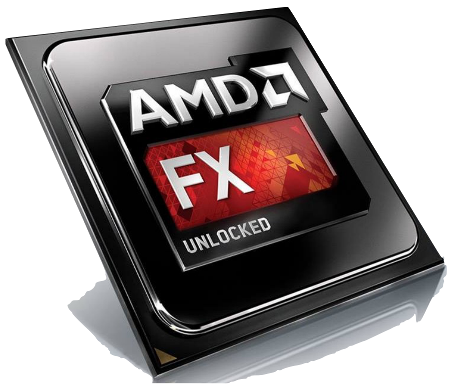
MICHEL STEUWER

Die Manycore Ära



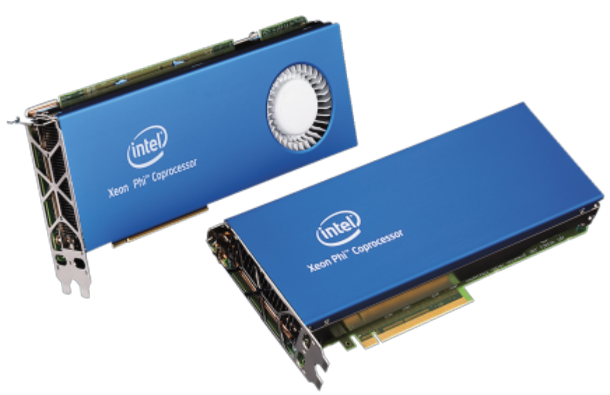
Inspiriert von Herb Sutter *“The Free Lunch is Over: A Fundamental Turn Towards Concurrency in Software”*

Die Manycore Ära



multicore CPUs

GPUs



Beschleuniger

FPGAs

Agenda

Meine Dissertation adressiert zwei zentrale Herausforderungen:

I. Die *Herausforderung der Programmierbarkeit*

II. Die *Herausforderung der Performance-Portabilität*

TEIL I

Die Herausforderung der Programmierbarkeit

Programmierung mit OpenCL

- Beispiel: Parallele Summation eines Arrays in OpenCL

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

Programmierung mit OpenCL

- Beispiel: Parallele Summation eines Arrays in OpenCL

Kernel Funktion wird parallel von vielen **work-items** ausgeführt

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

Work-items werden durch eine **globale id** identifizier

Programmierung mit OpenCL

- Beispiel: Parallele Summation eines Arrays in OpenCL

Work-items werden zu **work-groups** zusammengefasst

Lokale id innerhalb einer work-group

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

Programmierung mit OpenCL

- Beispiel: Parallele Summation eines Arrays in OpenCL

Großer, aber langsamer **globaler Speicher**

Kleiner, aber schneller **lokaler Speicher**

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

Barrieren für Speicherkonsistenz

Programmierung mit OpenCL

- Beispiel: Parallele Summation eines Arrays in OpenCL

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

Funktional korrekte Implementierungen in OpenCL sind schwierig!

Das SkelCL Programmiermodel

Das SkelCL Programmiermodel

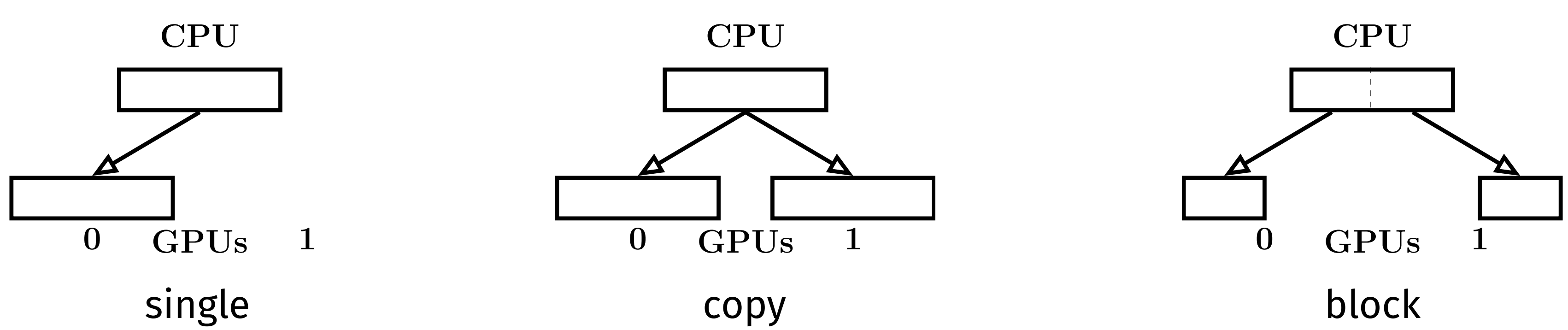
Drei Abstraktionen zu OpenCL hinzugefügt:

- **Parallele Datencontainer**
  für eine einheitliche Speicherverwaltung zwischen CPU und (mehreren) GPUs
  - **implizite Speichertransfers zwischen CPU und GPU**
  - ***lazy copying* minimiert den Datentransfer**

- **Wiederkehrende Muster paralleler Programmierung** (Algorithmische Skelette)
  für eine vereinfachte Beschreibung paralleler Berechnungen

$$zip\ (\oplus)\ [x_1, ..., x_n]\ [y_1, ..., y_n] = [x_1 \oplus y_1, ..., x_n \oplus y_n]$$
$$reduce\ (\oplus)\ \oplus_{id}\ [x_1, ..., x_n] = \oplus_{id} \oplus x_1 \oplus ... \oplus x_n$$

- **Daten Verteilungen**
  für einen transparenten Datentransfer in Systemen mit mehreren GPUs.



Die SkelCL Softwarebibliothek am Beispiel

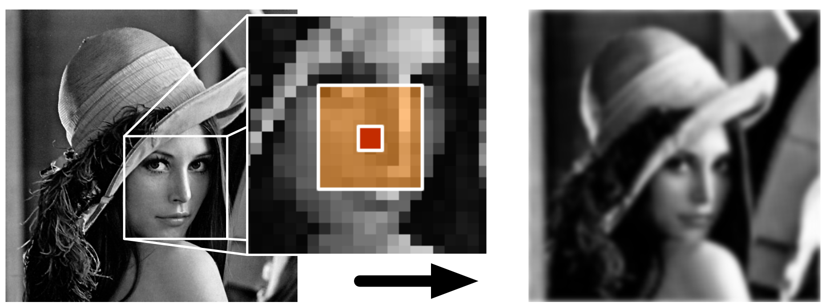
$$\text{dotProduct } A\ B = \text{reduce } (+)\ 0\ (\text{zip } (\times)\ A\ B)$$

```
#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();
}
```

Neue Algorithmische Skelette

Stencil Berechnungen

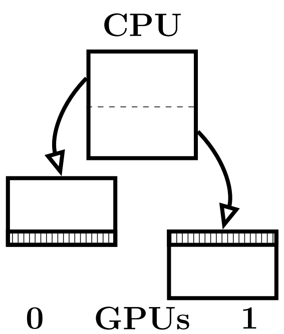
Beispiel: Gaußscher Weichzeichner



$$\text{gauss } M = \text{stencil } f\ 1\ \bar{0}\ M$$

wo f die Funktion is welche den Gaußschen Weichzeichner beschreibt

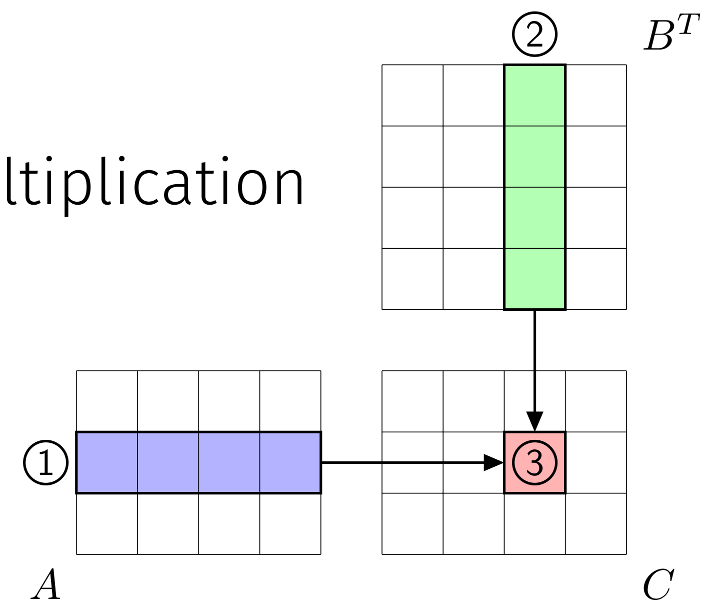
Unterstützung für mehre GPUs:



overlap Verteilung

Allpairs Berechnungen

Example:
Matrix Multiplication



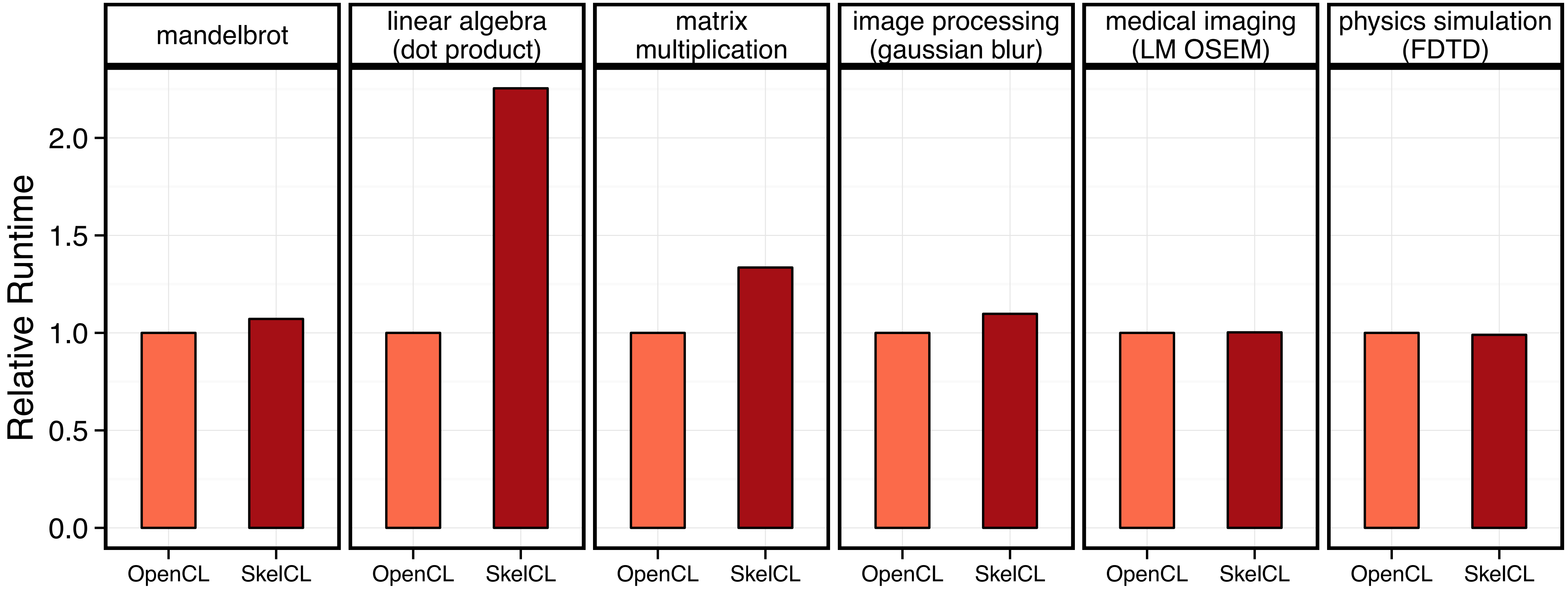
$$A \times B = \text{allpairs dotProduct } A\ B^T$$

Optimierung für *zipReduce* Muster:

$$\text{dotProduct } a\ b = \text{zipReduce } (+)\ 0\ (\times)\ a\ b$$

Unterstützung für mehrere GPUs mit **block** und **copy** Verteilung

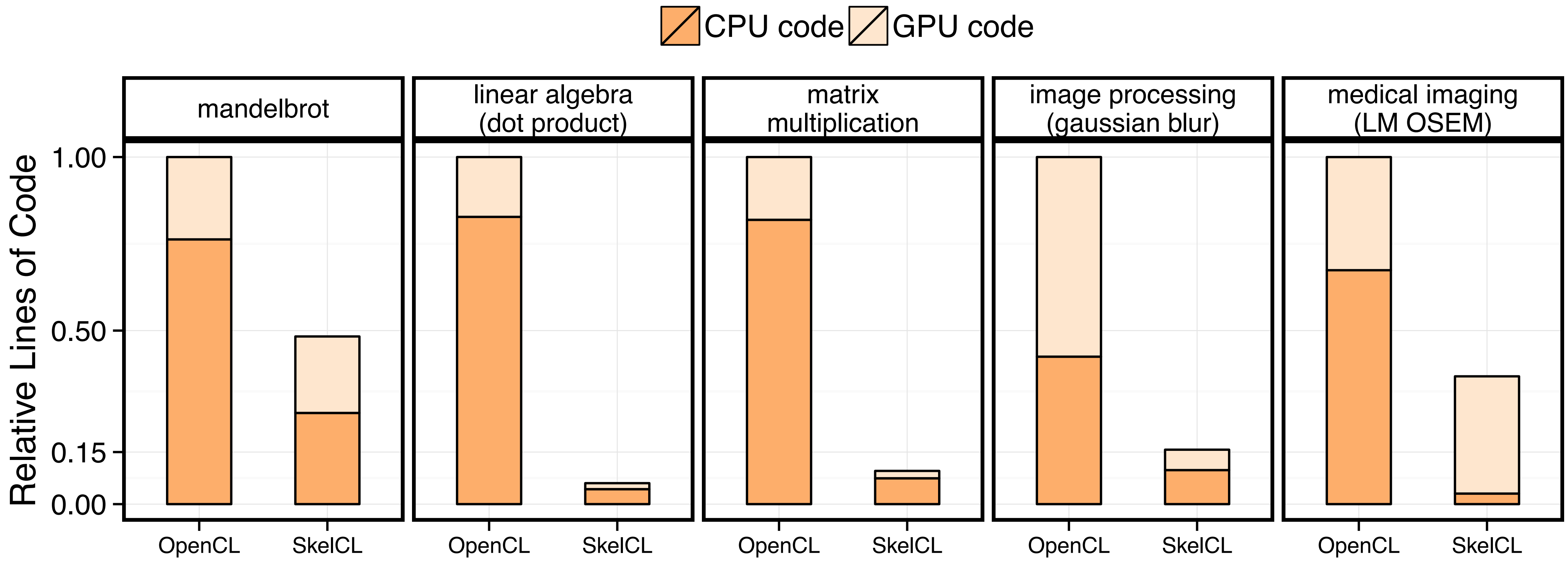
SkelCL Evaluation — Geschwindigkeit



SkelCL nahe an der Geschwindigkeit von OpenCL!

(Ausnahme: dot product … mehr dazu in Teil II)

SkelCL Evaluation — Produktivität



SkelCL Programme sind signifikant kürzer!

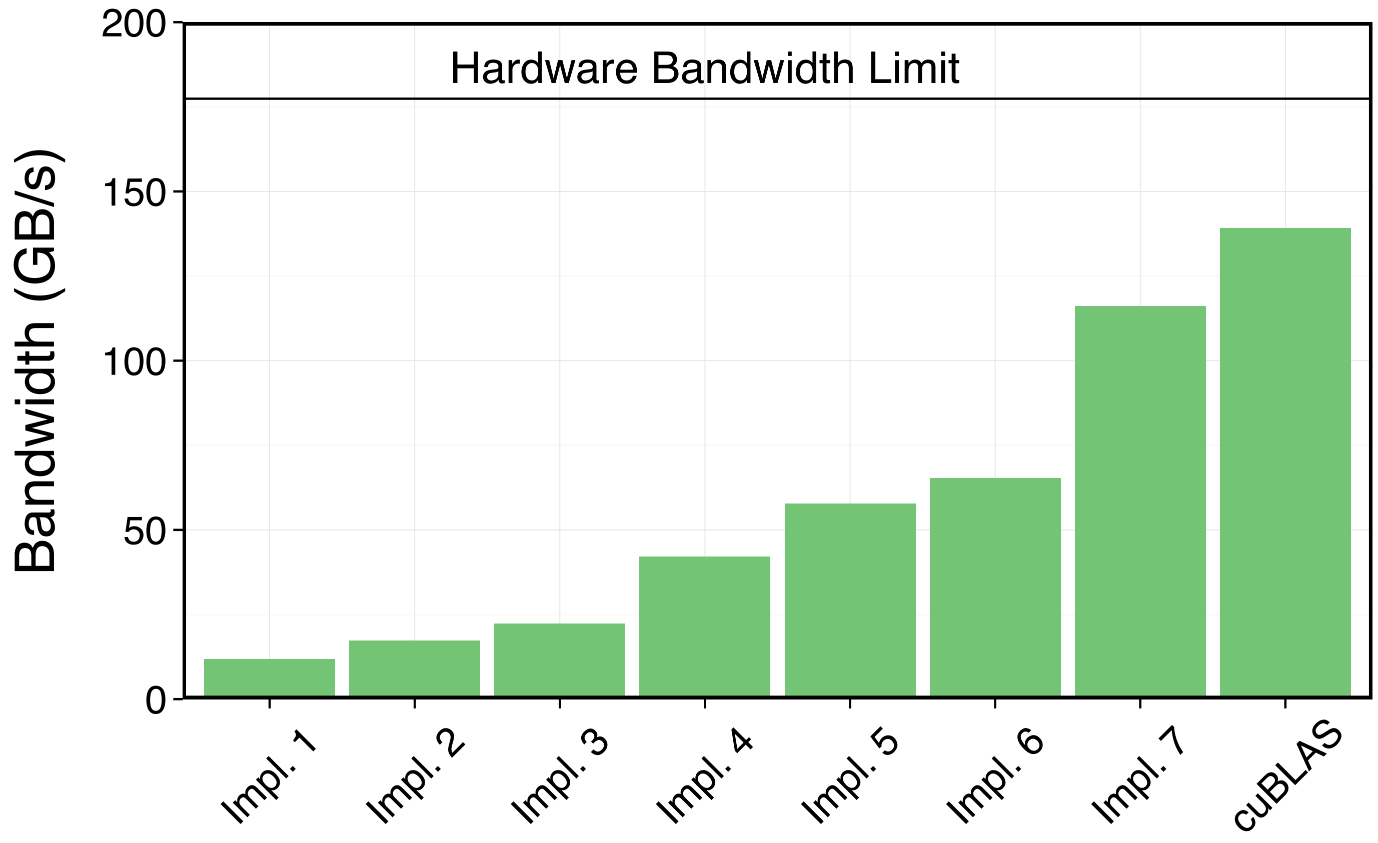
TEIL II

Die Herausforderung der Performance-Portabilität

Ein neuer Ansatz zur Performance portablen Codegenerierung

OpenCL und Performance-Portabilität

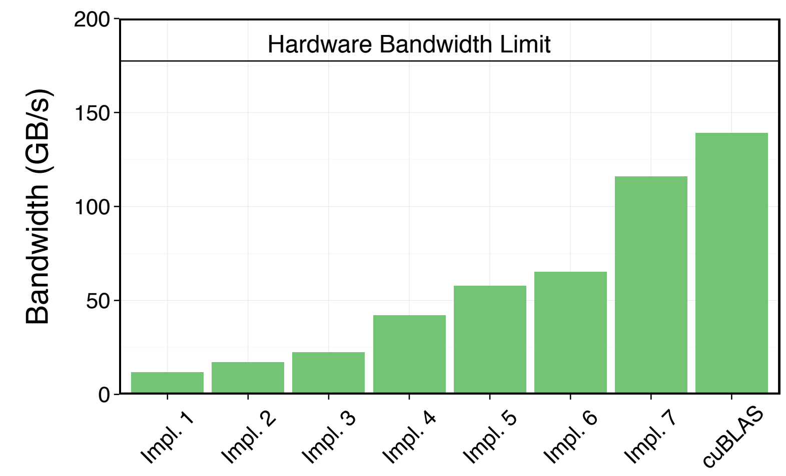
- Beispiel: Parallele Summation eines Arrays in OpenCL
- Vergleich von 7 OpenCL Implementierungen von Nvidia



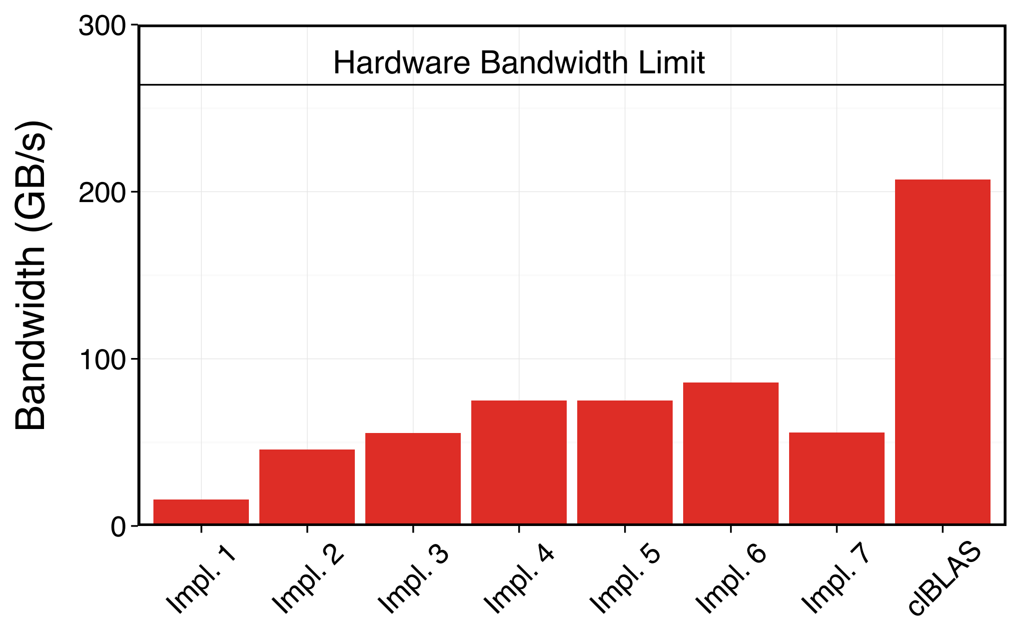
(a) Nvidia's GTX 480 GPU.

OpenCL und Performance-Portabilität

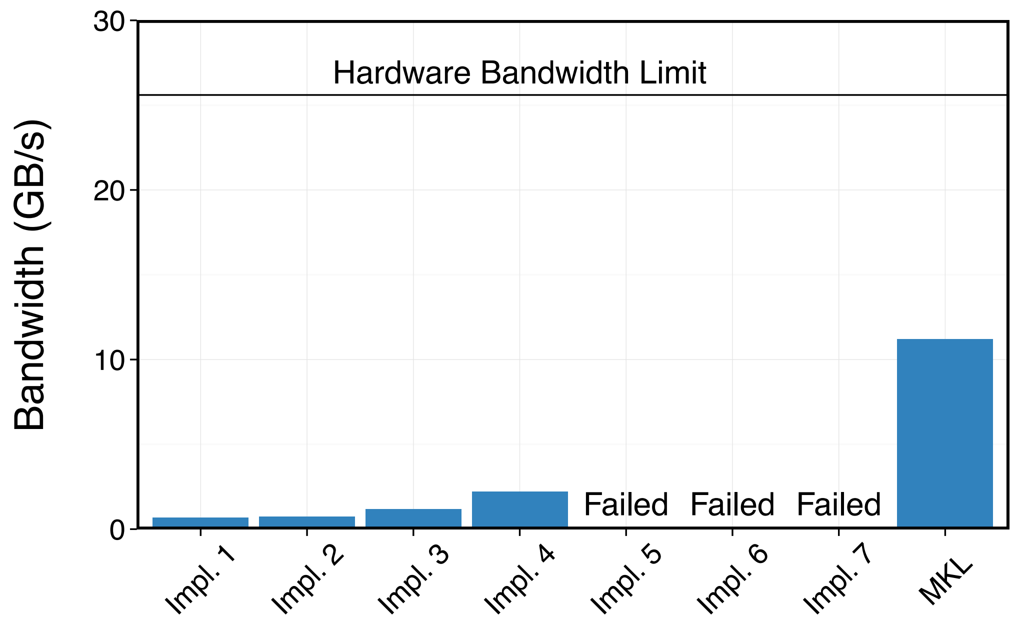
- Beispiel: Parallele Summation eines Arrays in OpenCL
- Vergleich von 7 OpenCL Implementierungen von Nvidia



(a) Nvidia's GTX 480 GPU.



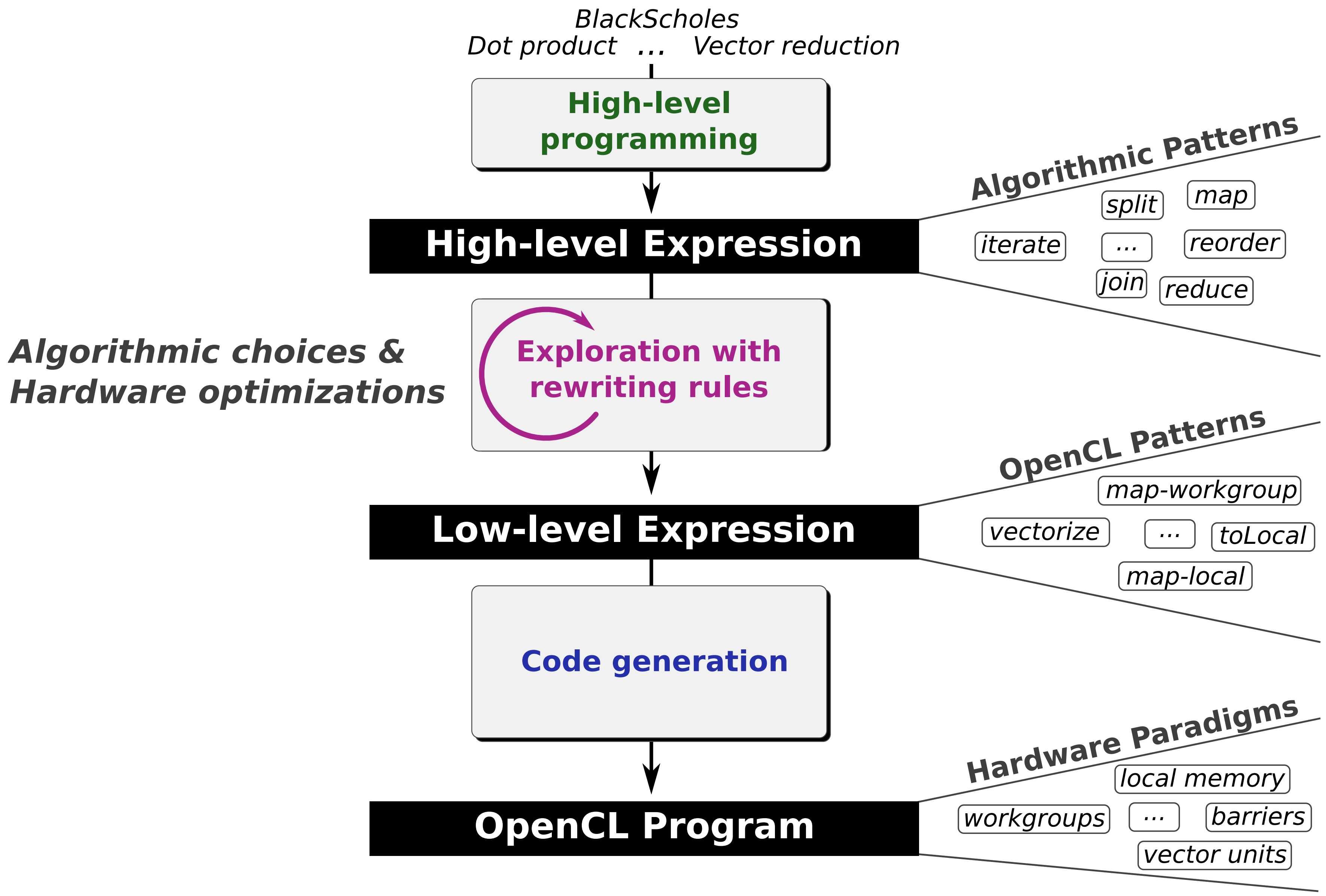
(b) AMD's HD 7970 GPU.



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

Performance in OpenCL ist nicht portabel!

Performance portable Codegenerierung mit Transformationsregeln



Beispiel: Parallele Summation

① $vecSum = reduce\ (+)\ 0$

↓ rewrite rules

②

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

↗ code generation

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];
}
```

Beispiel: Parallele Summation

① $$vecSum = reduce\ (+)\ 0$$

rewrite rules

code generation

②

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

③

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

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

① Algorithmische Primitive

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

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

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

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

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

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

① High-Level Programme

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

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

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

$$\begin{aligned} gemv = \lambda\ mat\ xs\ ys\ \alpha\ \beta.map\ (+)\ (&\\ zip\ (map\ (scal\ \alpha \circ dot\ xs)\ mat)\ (scal\ \beta\ ys)\ )&\end{aligned}$$

Beispiel: Parallele Summation

① $vecSum = reduce\ (+)\ 0$

rewrite rules

code generation

②

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

③

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

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

Beispiel: Parallele Summation

① $vecSum = reduce\ (+)\ 0$

rewrite rules ↓

code generation ↗

②

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

③

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

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

② Algorithmische Transformationsregeln

- Transformationsregeln sind **semantikerhaltend**
- Drücken Auswahl bei der algorithmische Implementierungen aus

Split-Join Zerlegung:

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

Map Zusammenschluss:

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

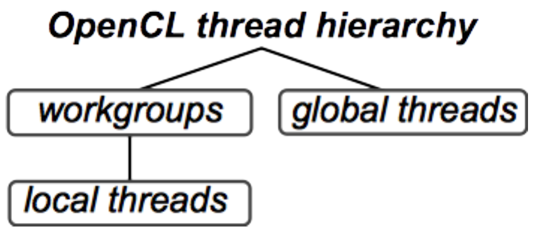
Reduktionsregeln:

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

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

② OpenCL Primitive

Primitive	**OpenCL Konzept**
mapGlobal	Work-items
mapWorkgroup / mapLocal	Work-groups
mapSeq / reduceSeq	Sequentielle Implementierungen
toLocal / toGlobal	Speicherbereiche
mapVec / splitVec / joinVec	Vektorisierung



② OpenCL Transformationsregeln

- Drücken hardware-spezifische Optimierungen aus

Map:

$$map\ f \rightarrow mapWorkgroup\ f \mid mapLocal\ f \mid mapGlobal\ f \mid mapSeq\ f$$

Lokaler/ Globaler Speicher:

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

Vektorisierung:

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

Map-Reduktion Zusammenschluss:

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

Beispiel: Parallele Summation

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

rewrite rules

code generation

②

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

③

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

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

Beispiel: Parallele Summation

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];
}
```

③ Muster basierte OpenCL Codegenerierung

- Generiere OpenCL Code für jedes OpenCL Primitiv

$mapGlobal\ f\ xs$ →

```
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]);
}
```

$\vdots$ $\vdots$

Transformationsregeln definieren einen Suchraum gültiger Implementierungen

$$reduce\ (+)\ 0$$

$$\Big|$$

$$reduce\ (+)\ 0 \circ reducePart\ (+)\ 0$$

Transformationsregeln definieren einen Suchraum gültiger Implementierungen

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

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

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

Transformationsregeln definieren einen Suchraum gültiger Implementierungen

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

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

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

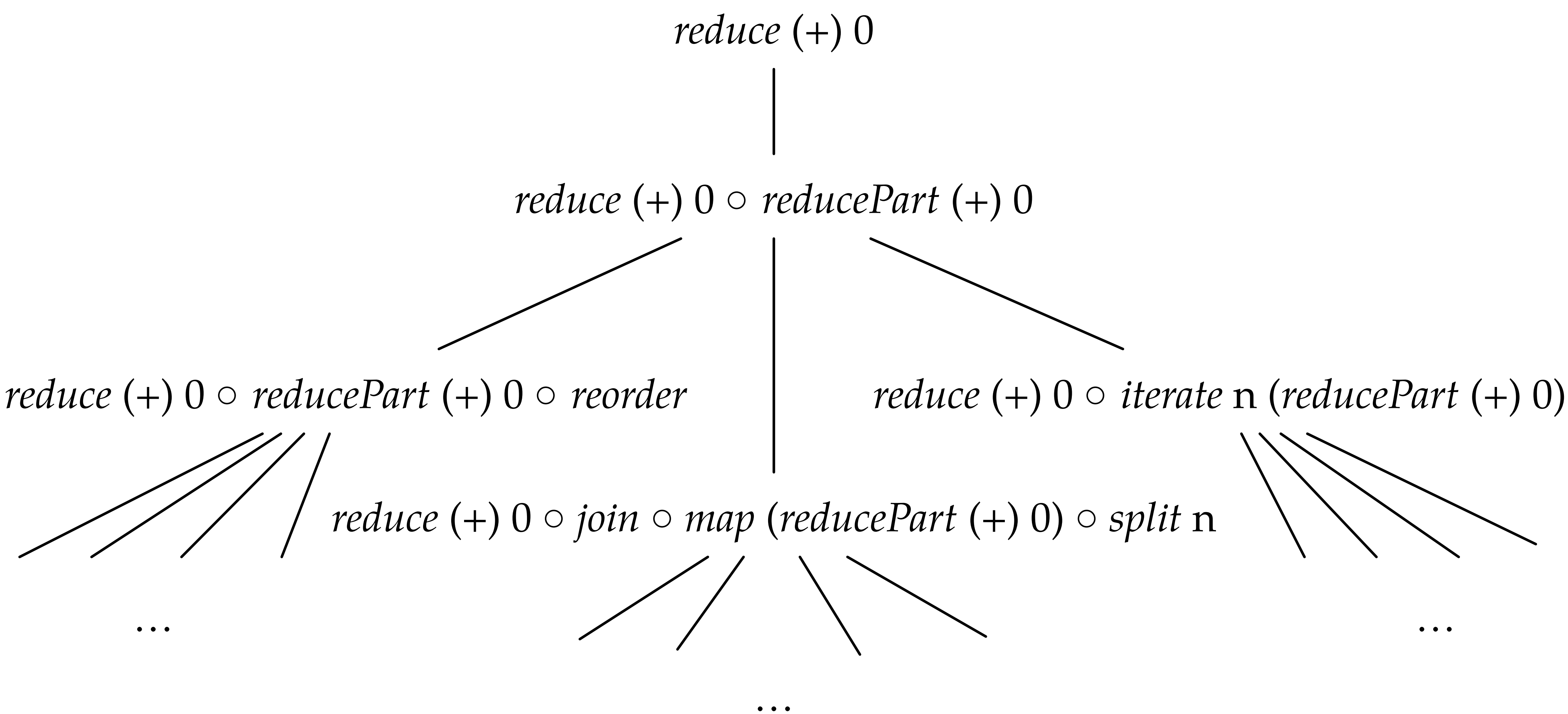
$$\mathit{reduce}\ (+)\ 0 \circ \mathit{iterate}\ \mathrm{n}\ (\mathit{reducePart}\ (+)\ 0)$$

Transformationsregeln definieren einen Suchraum gültiger Implementierungen

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

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

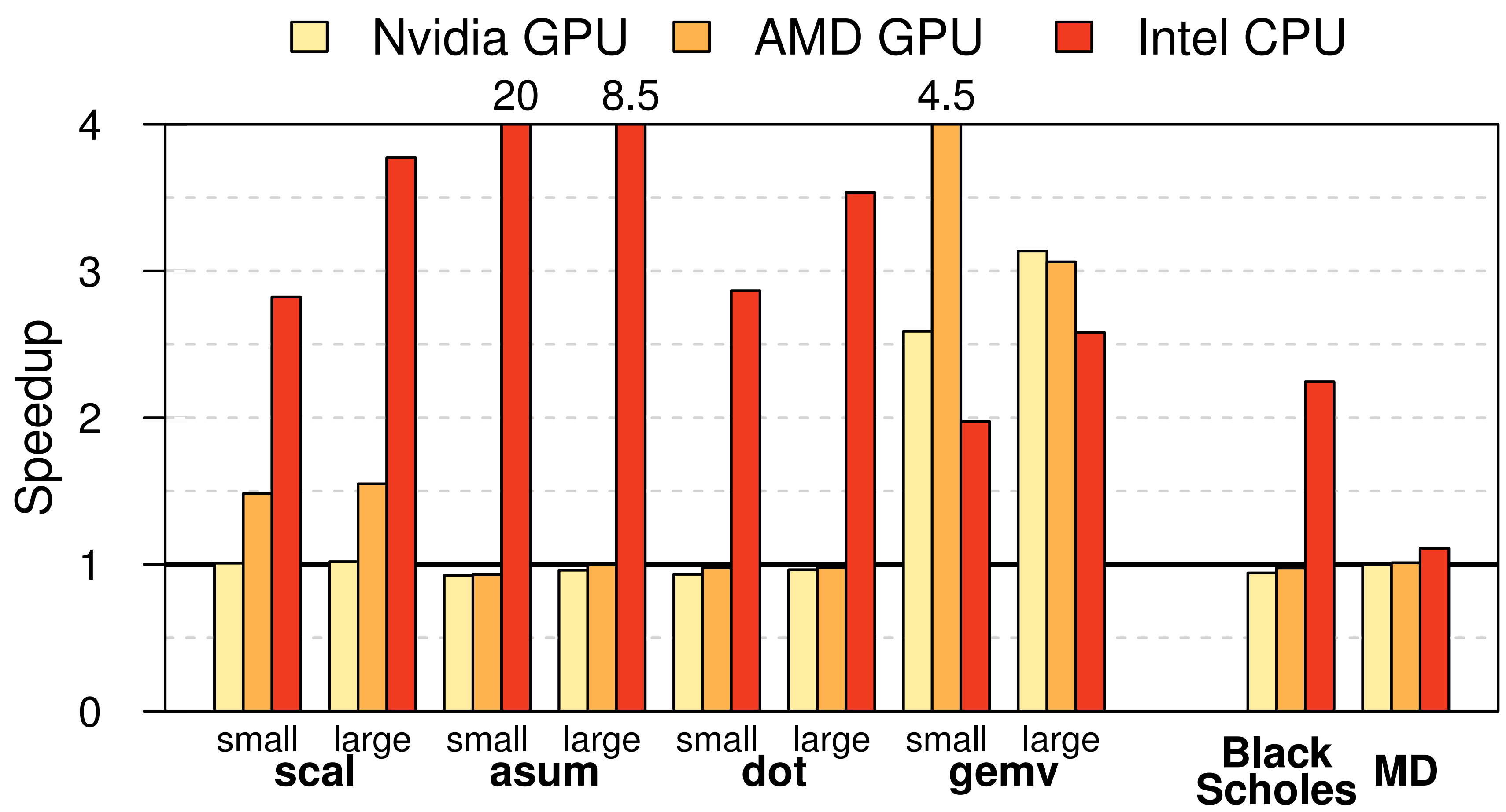
Transformationsregeln definieren einen Suchraum gültiger Implementierungen



- Vollautomatische Suche nach guten Implementierungen möglich!
  (Eine einfache Suchstrategie ist in der Dissertation beschrieben)

Evaluation — Geschwindigkeit

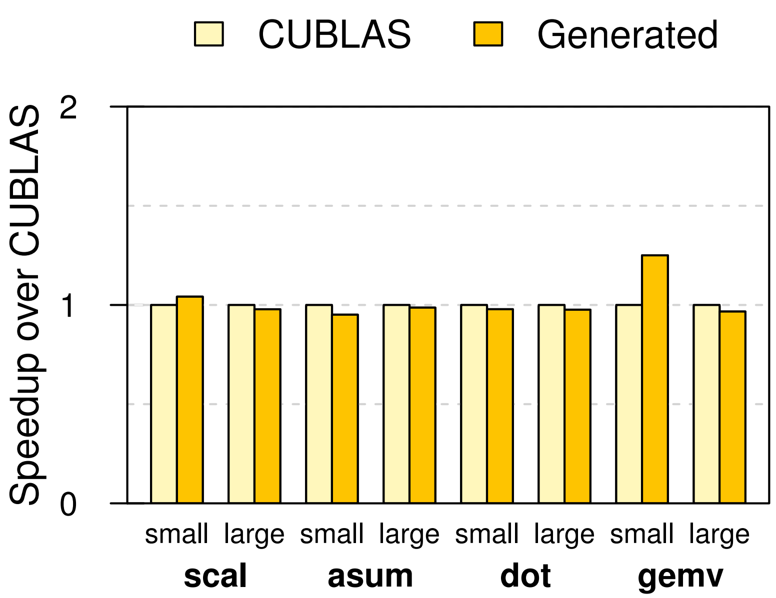
gegenüber einer funktional portablen Implementierung



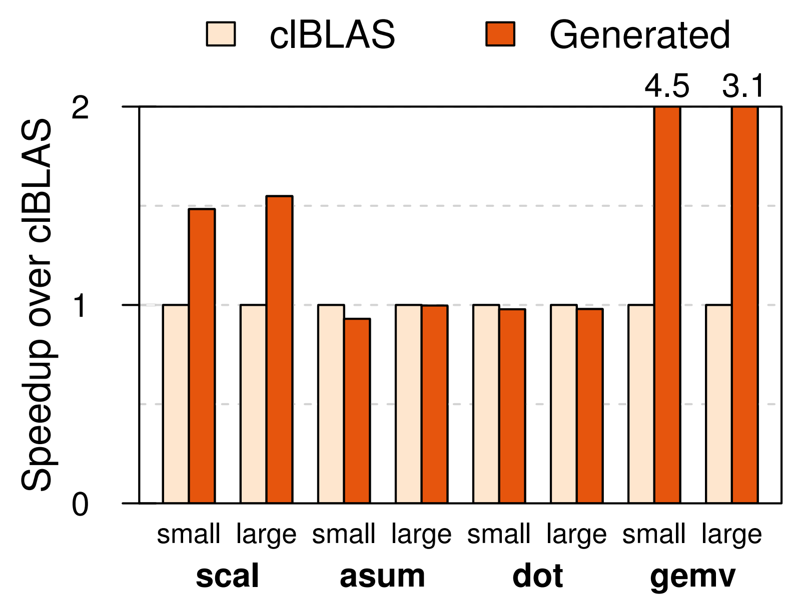
Bis zu **20x** Speedup gegenüber der funktional portablen clBLAS Implementierung

Evaluation — Geschwindigkeit

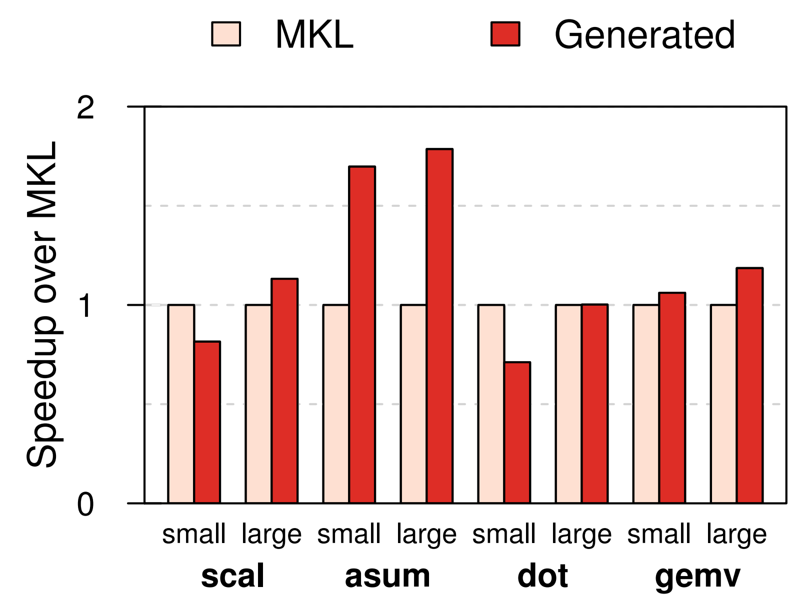
gegenüber Hardware spezifischen Implementierungen



(a) Nvidia GPU



(b) AMD GPU



(c) Intel CPU

- Automatisch generierter Code vs. handoptimierten Code
- Konkurrenzfähige Ergebnisse vs. hochoptimierte Implementierungen
- Bis zu **4.5x S**peedup für gemv auf der AMD GPU

Zusammenfassung

- Um die *Herausforderung der Programmierbarkeit* zu adressieren:
  - Ein neuer Ansatz zur Programmierung von Systemen mit mehreren GPUs
  - Zwei neue formell definierte und implementierte algorithmische Skelette
- Um die *Herausforderung der Performance-Portabilität* zu adressieren:
  - Ein formelles System zur Transformation muster-basierter Programme
  - Ein Codegenerator der Performance-Portabilität erreicht

Ergebnisse der Suche

Automatisch Gefundene Ausdrücke

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

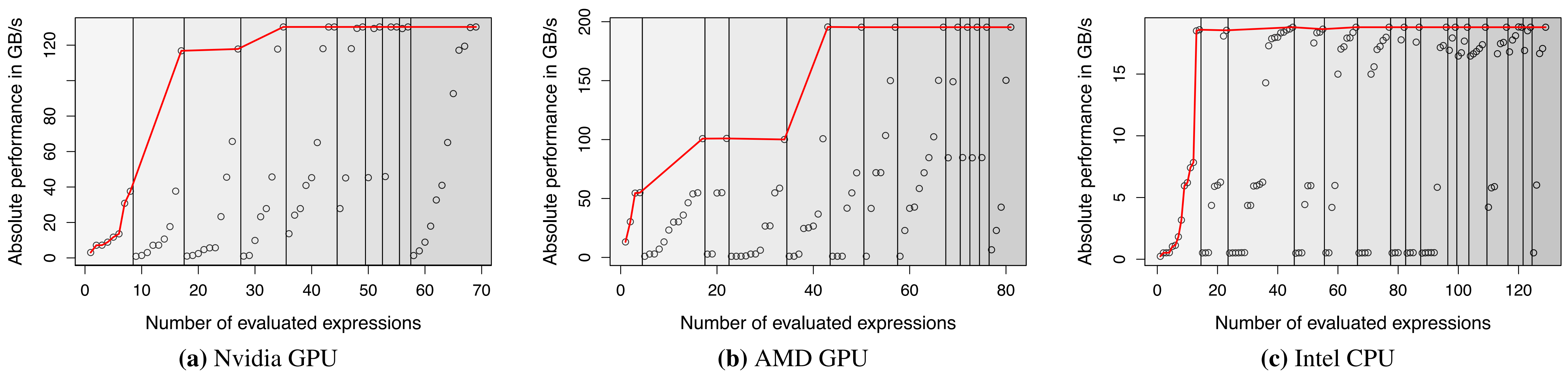
↓

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

Gesucht für: **Nvidia** GTX 480 GPU, **AMD** Radeon HD 7970 GPU, **Intel** Xeon E5530 CPU

Ergebnisse der Suche

Effizienz der Suche



- Die Suche hat auf jeder Platform weniger als 1 Stunde gedauert
- Durchschnittliche Zeit zur Ausführung eines Kandidaten weniger als 1/2 Sekunde

Fazit des Beispiels

- Optimieren in OpenCL ist kompliziert
  - Verständnis für die Zielarchitektur benötigt
- Veränderungen im Program nicht offensichtlich

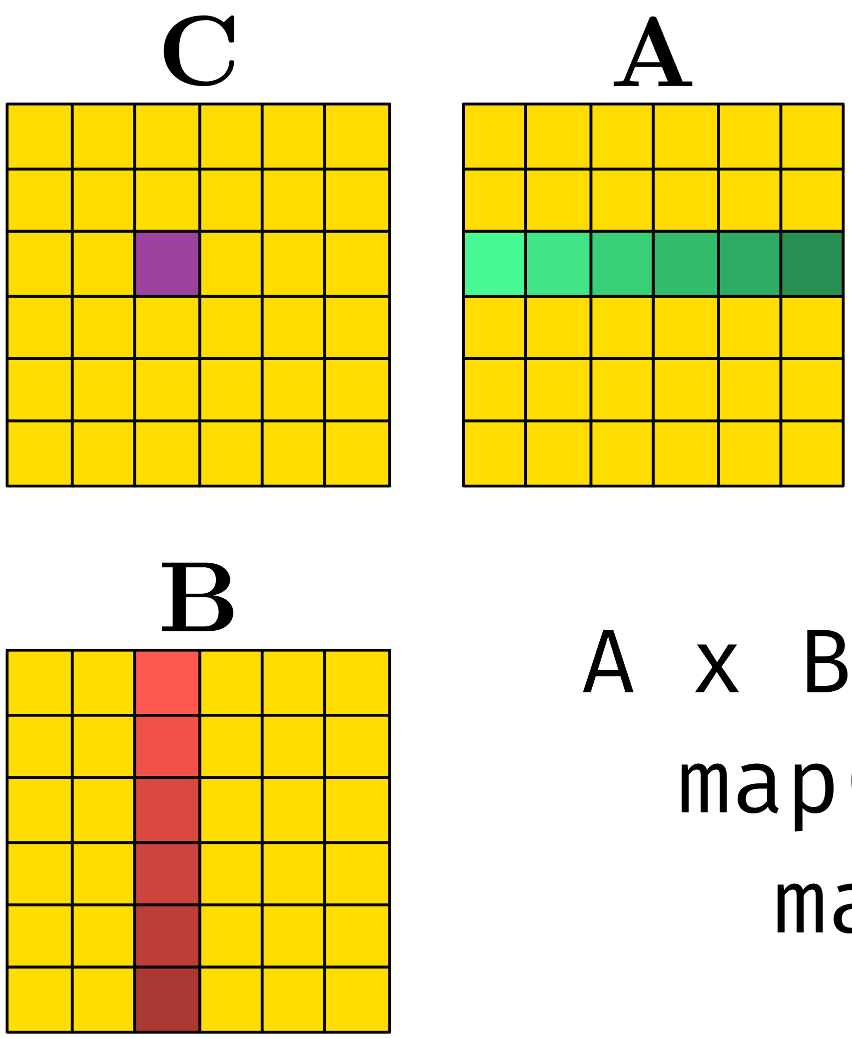
```
kernel
void reduce0(global float* g_idata,
             global float* g_odata,
             unsigned int n,
             local float* l_data) {
  unsigned int tid = get_local_id(0);
  unsigned int i   = get_global_id(0);
  l_data[tid] = (i < n) ? g_idata[i] : 0;
  barrier(CLK_LOCAL_MEM_FENCE);

  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];
}
```

Nicht Optimierte Implementierung

```
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];
}
```

Voll Optimierte Implementierung

Matrix Multiplikation




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

Suche für Matrix Multiplikation

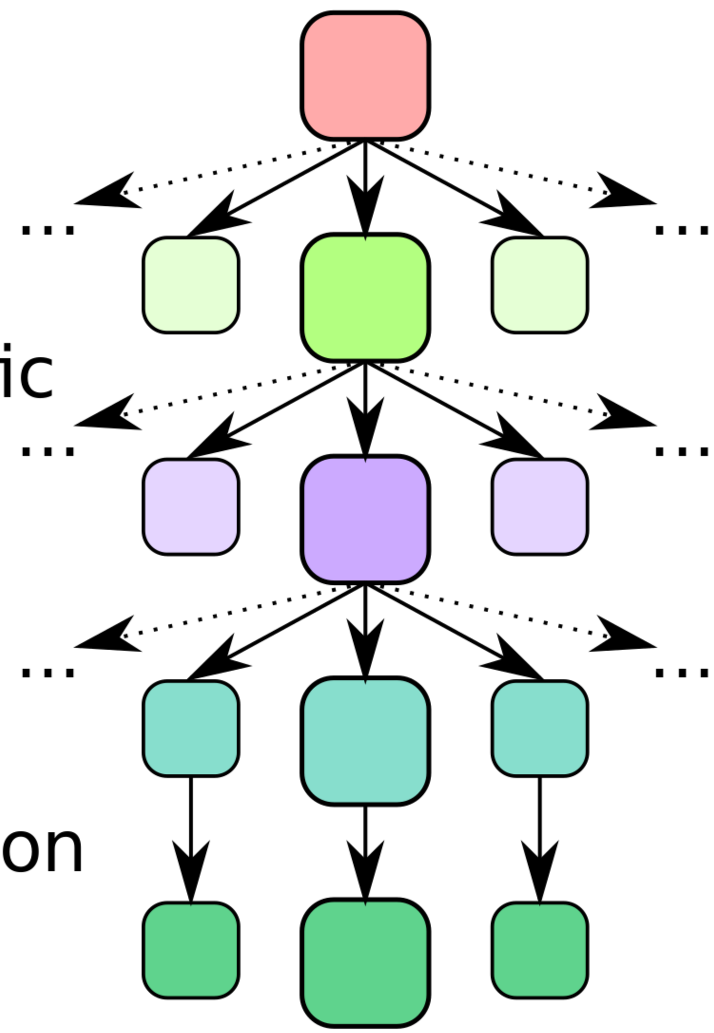
Phases:

Algorithmic Exploration

OpenCL specific Exploration

Parameter Exploration

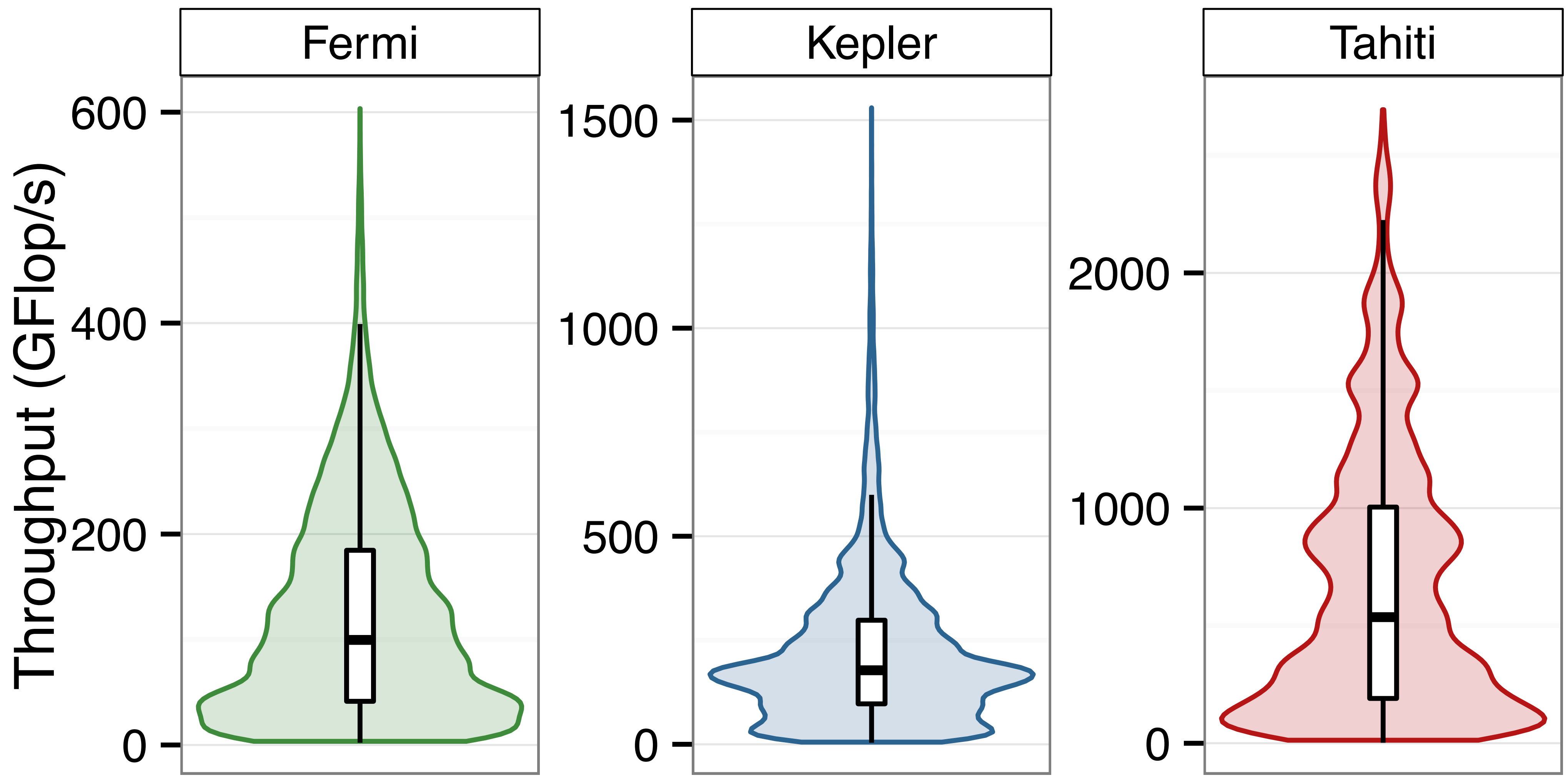
Code Generation



Program Variants:

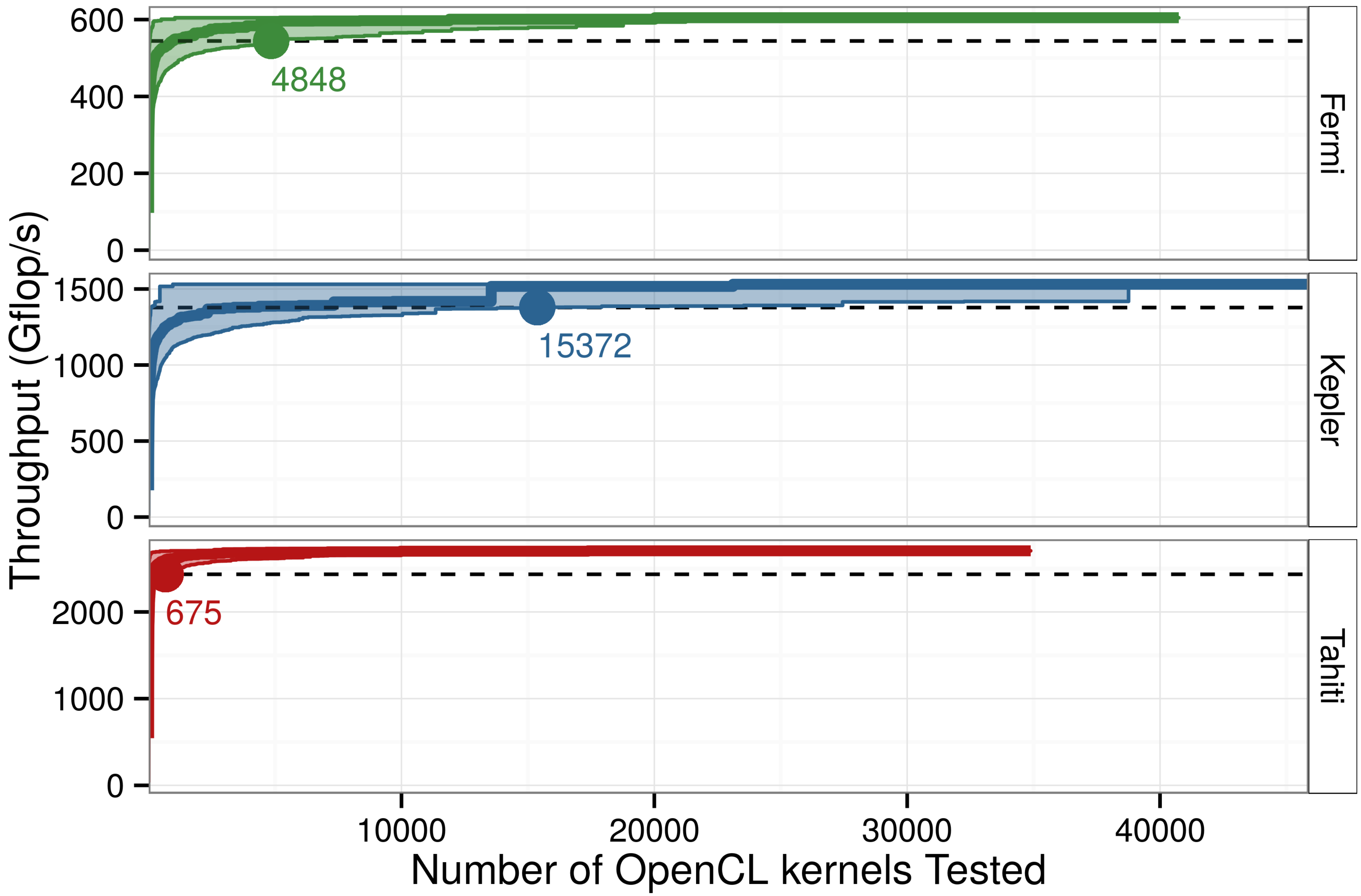
High-Level Program	1
Algorithmic Rewritten Program	8
OpenCL Specific Program	760
Fully Specialized Program	46,000
OpenCL Code	46,000

Suchraum für Matrix Multiplikation



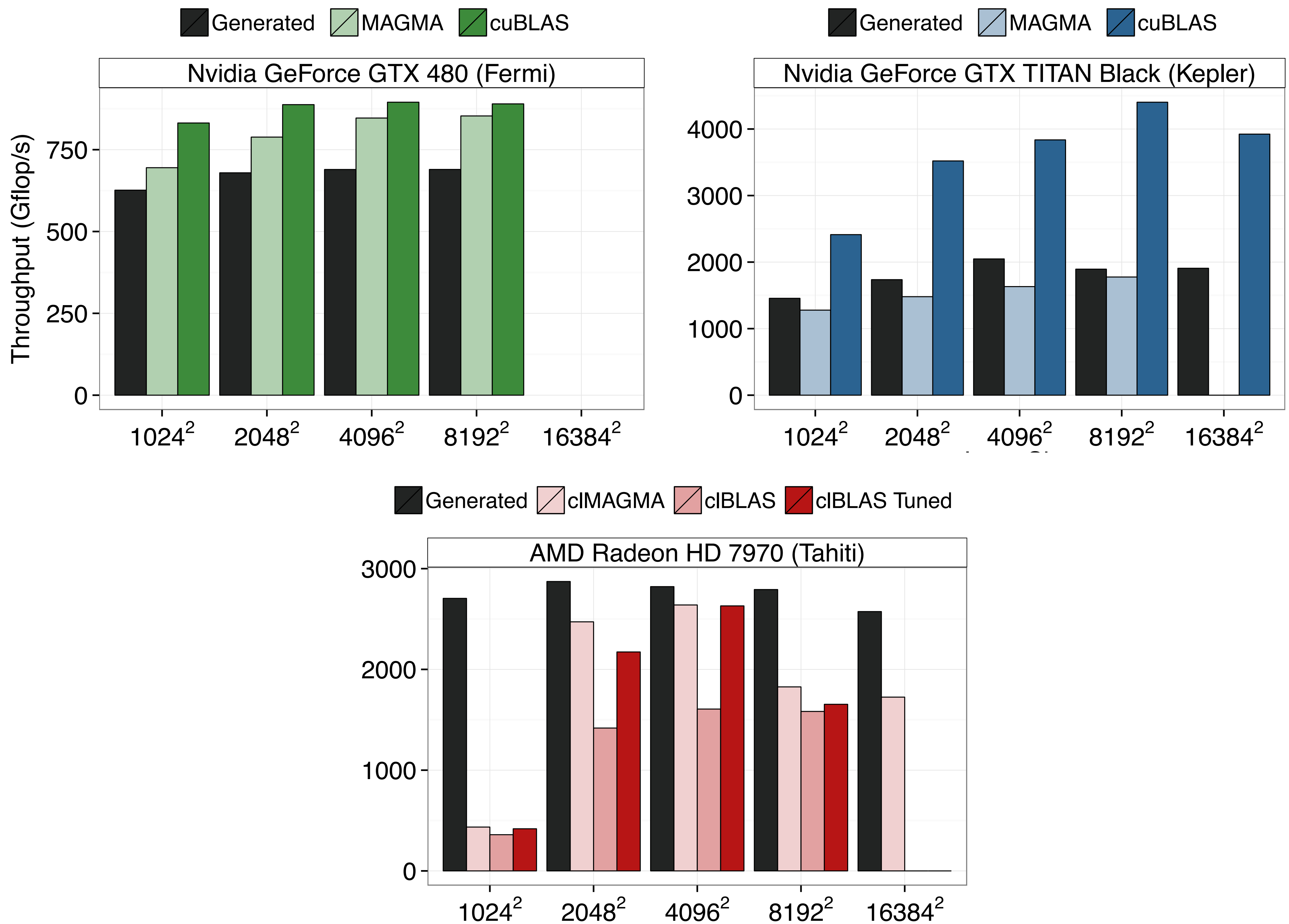
Nur einige generierte OpenCL Programme mit sehr guter Performance

Performance Entwicklung für Matrix Multiplikation



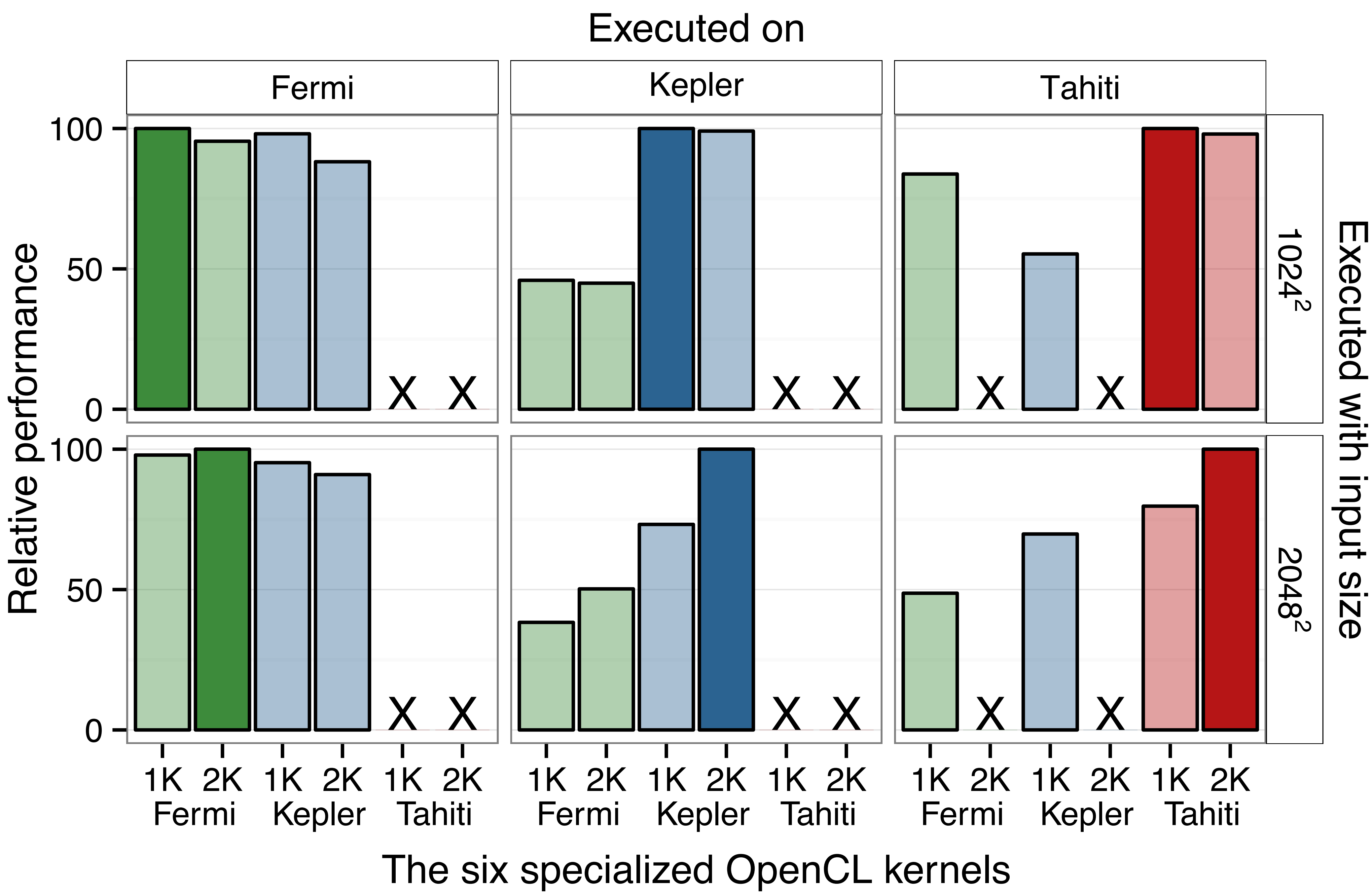
Selbst mit einer einfachen zufälligen Strategie kann man erwarten schnell ein Program mit guter Performance zu finden

Evaluation für Matrix Multiplikation



Performance nahe oder sogar besser als handoptimierte MAGMA Bibliothek

Performance-Portabilität von Matrix Multiplikation



Generierte Programmer sind spezialisiert für GPU und Eingabegröße