Bauhaus-Universität Weimar

Parallele und Verteilte Systeme

Grundlagen zur Programmierung von Mehrkern-Systemen, verteilter Berechnung und massiver Parallelität

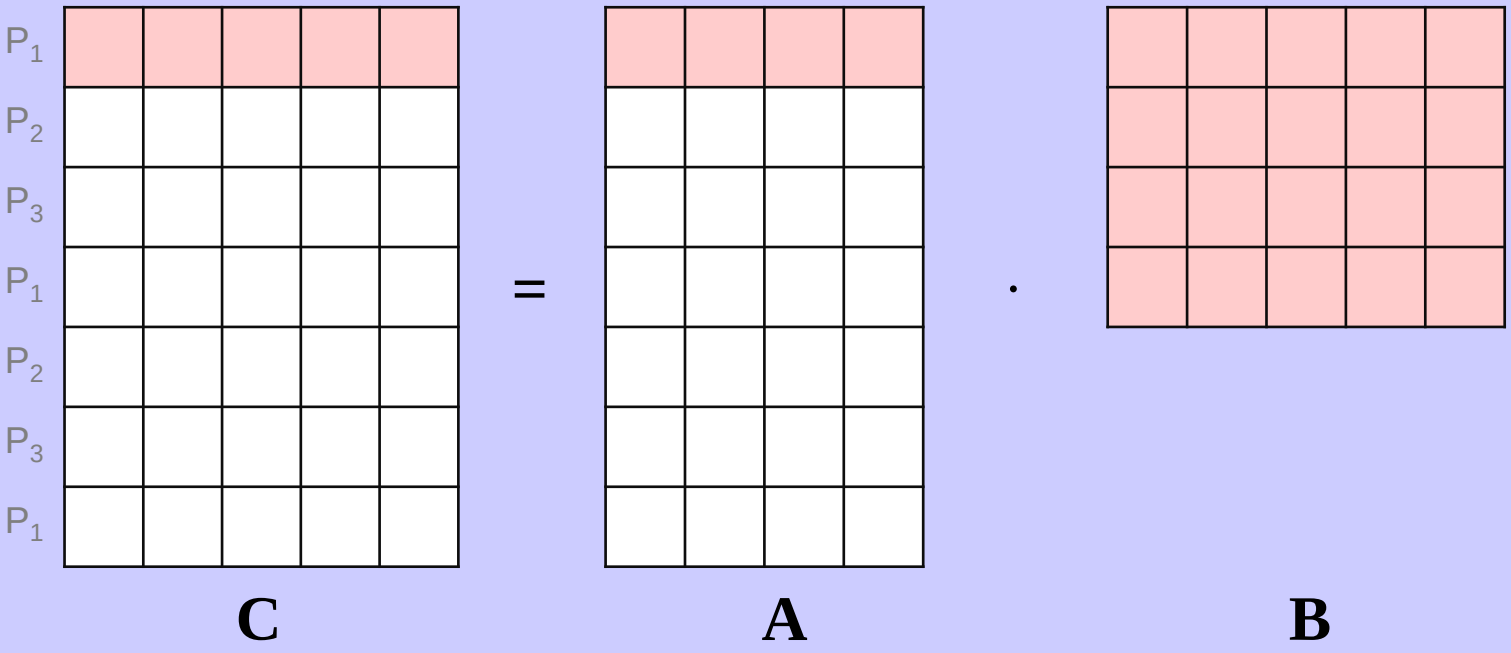
Vorlesung WS 2017/18

(Unterlagen nur für den internen Gebrauch!)

Übersicht

- **Massiv parallele Programmierung**
  - Optimierung der Matrix-Multiplikation
    - Mehrdimensionale Gitter (NDGRID)
  - Parallele Reduktion
  - Speicherorganisation
  - Aufteilung der Kernel-Aufrufe
  - Vektordatentypen
  - Sonstige Funktionen

Matrix-Multiplikation (zeilenweise)



Version 1: C zeilenweise

```
const char *KernelSource =
"#define DIM 1000                                          // Groesse der Matrix \n"
"__kernel void matmult(__global float *A, __global float *B, __global float *C)\n"
"{                                                                               \n"
"    int i, j, k;                                                                \n"
"    i = get_global_id(0);                                                       \n"
"        for (j = 0; j < DIM; j++)                                               \n"
"            for (k = 0; k < DIM; k++)                                           \n"
"                C[i*DIM+j] += A[i*DIM+k] * B[k*DIM+j];                          \n"
"}                                                                               \n"
"\n";
```

Änderungen am Hauptprogramm

```
#define DIM       1000                                   // Matrix Dimensionen
#define DATA_SIZE DIM*DIM*sizeof(float)                  // Matrixgröße im Speicher

void main(int argc, char **argv)
{
  // ...
  float **A, **B, **C;                                                  // Matrizen
  cl_mem Ap, Bp, Cp;
  size_t global[1] = {DIM};                          // Oder: size_t global = DIM;

  A = alloc_mat(DIM, DIM); init_mat(A, DIM, DIM);          // Speicher für Matrizen holen
  B = alloc_mat(DIM, DIM); init_mat(B, DIM, DIM);                  // und initialisieren
  C = alloc_mat(DIM, DIM);
  // ...
  kernel = clCreateKernel(program, "matmult", &err);               // Spezifiziere Kernel
  Ap = clCreateBuffer(context, CL_MEM_READ_ONLY,  DATA_SIZE, NULL, &err); // Erzeuge Puffer
  Bp = clCreateBuffer(context, CL_MEM_READ_ONLY,  DATA_SIZE, NULL, &err);
  Cp = clCreateBuffer(context, CL_MEM_READ_WRITE, DATA_SIZE, NULL, &err);
  clEnqueueWriteBuffer(command_queue, Ap, CL_TRUE, 0, DATA_SIZE, A[0], 0, NULL, NULL);
  clEnqueueWriteBuffer(command_queue, Bp, CL_TRUE, 0, DATA_SIZE, B[0], 0, NULL, NULL);
  clSetKernelArg(kernel, 0, sizeof(cl_mem), &Ap);    // Setzte die Argumentliste für Kernel
  clSetKernelArg(kernel, 1, sizeof(cl_mem), &Bp);
  clSetKernelArg(kernel, 2, sizeof(cl_mem), &Cp);
  // ...
  clEnqueueReadBuffer(command_queue, Cp, CL_TRUE, 0, DATA_SIZE, C[0], 0, NULL, NULL);
  // ...
}
```

Bauhaus-
Universität
Weimar

Optimierungspotential?

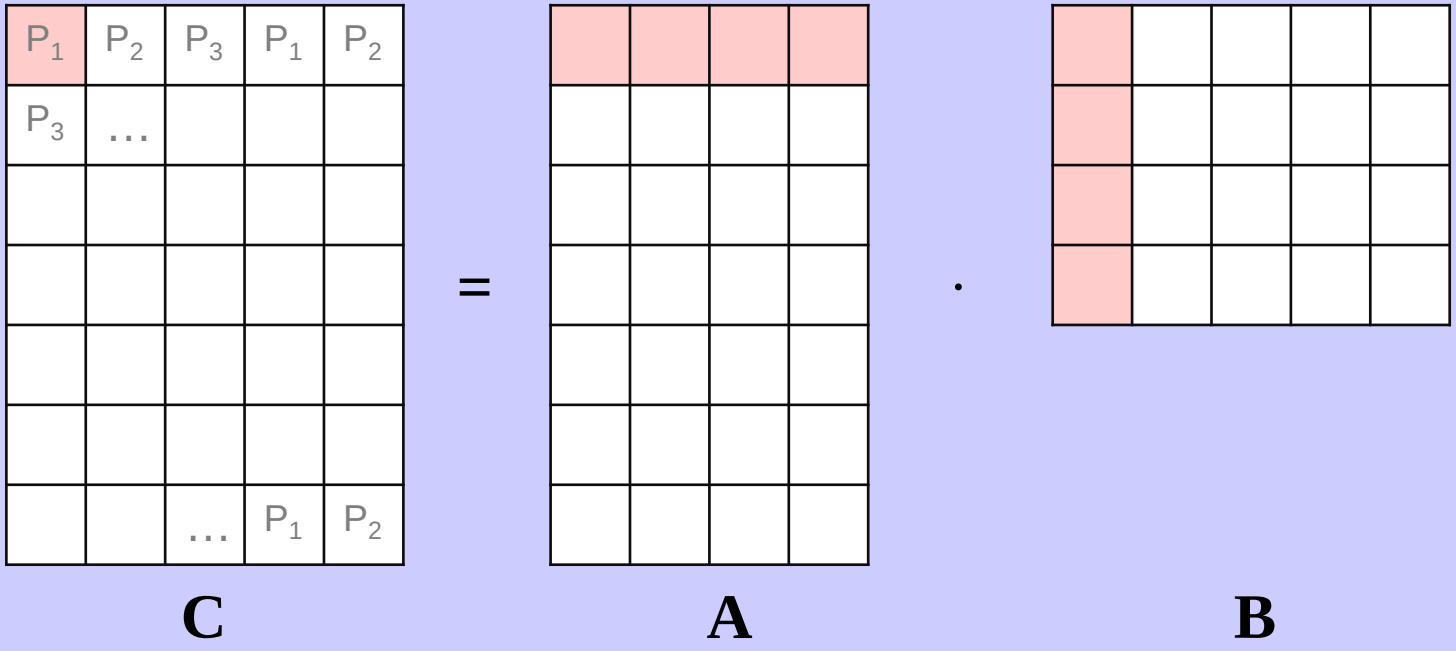
Matrix-Multiplikation mit OpenCL

Version 2: C zeilenweise, schreibend

```
const char *KernelSource =
"#define DIM 1000                                          // Groesse der Matrix \n"
"__kernel void matmult(__global float *A, __global float *B, __global float *C) { \n"
"    int i, j, k;                                                                 \n"
"    float sum;                          // Private Variable für Zwischenergebnisse \n"
"    i = get_global_id(0);                                                        \n"
"    for (j = 0; j < DIM; j++) {                                                  \n"
"        sum = 0.0;                                                               \n"
"        for (k = 0; k < DIM; k++)                                                \n"
"            sum += A[i*DIM+k] * B[k*DIM+j];                                      \n"
"        C[i*DIM+j] = sum;                                                        \n"
"    }                                                                            \n"
"}                                                                                \n"
"\n";
```

```
void main(int argc, char **argv)
{
  // ...
  Cp = clCreateBuffer(context, CL_MEM_WRITE_ONLY, DATA_SIZE, NULL, &err);
```

Matrix-Multiplikation (elementweise)



Version 3: C elementweise

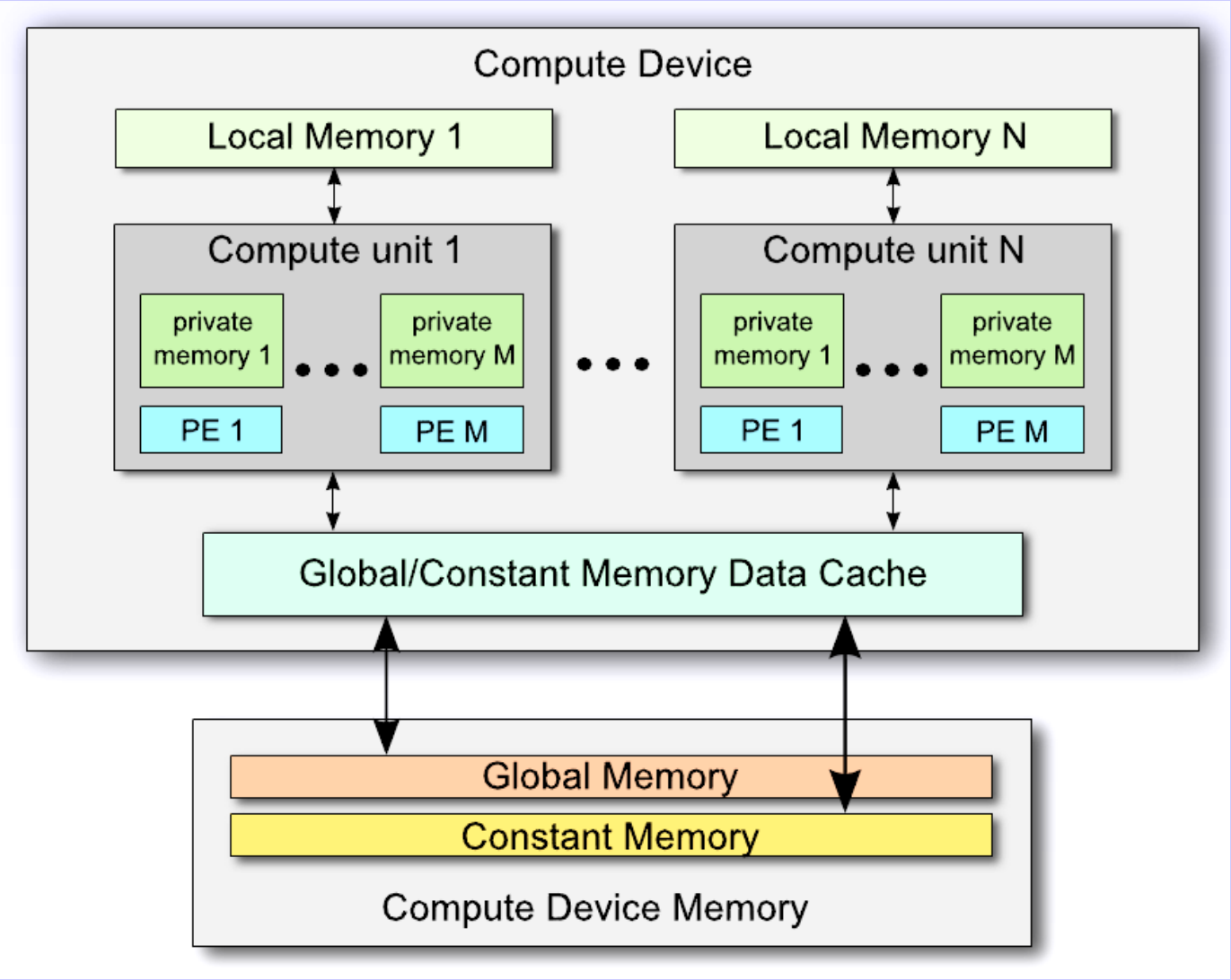
```
const char *KernelSource =
"#define DIM 1000                                                    // Size of matrix \n"
"__kernel void matmult(__global float *A, __global float *B, __global float *C) { \n"
"    int i, j, k;                                                                  \n"
"    float sum = 0.0;                                                              \n"
"    i = get_global_id(0);                                                         \n"
"    j = get_global_id(1);                                                         \n"
"    for (k = 0; k < DIM; k++)                                                     \n"
"        sum += A[i*DIM+k] * B[k*DIM+j];                                           \n"
"    C[i*DIM+j] = sum;                                                             \n"
"}                                                                                 \n"
"\n";
```

```
void main(int argc, char **argv)
{
  size_t global[2] = {DIM, DIM};
  // ...
  clEnqueueNDRangeKernel(command_queue, kernel, 2, NULL, global, NULL, 0, NULL, NULL);
```

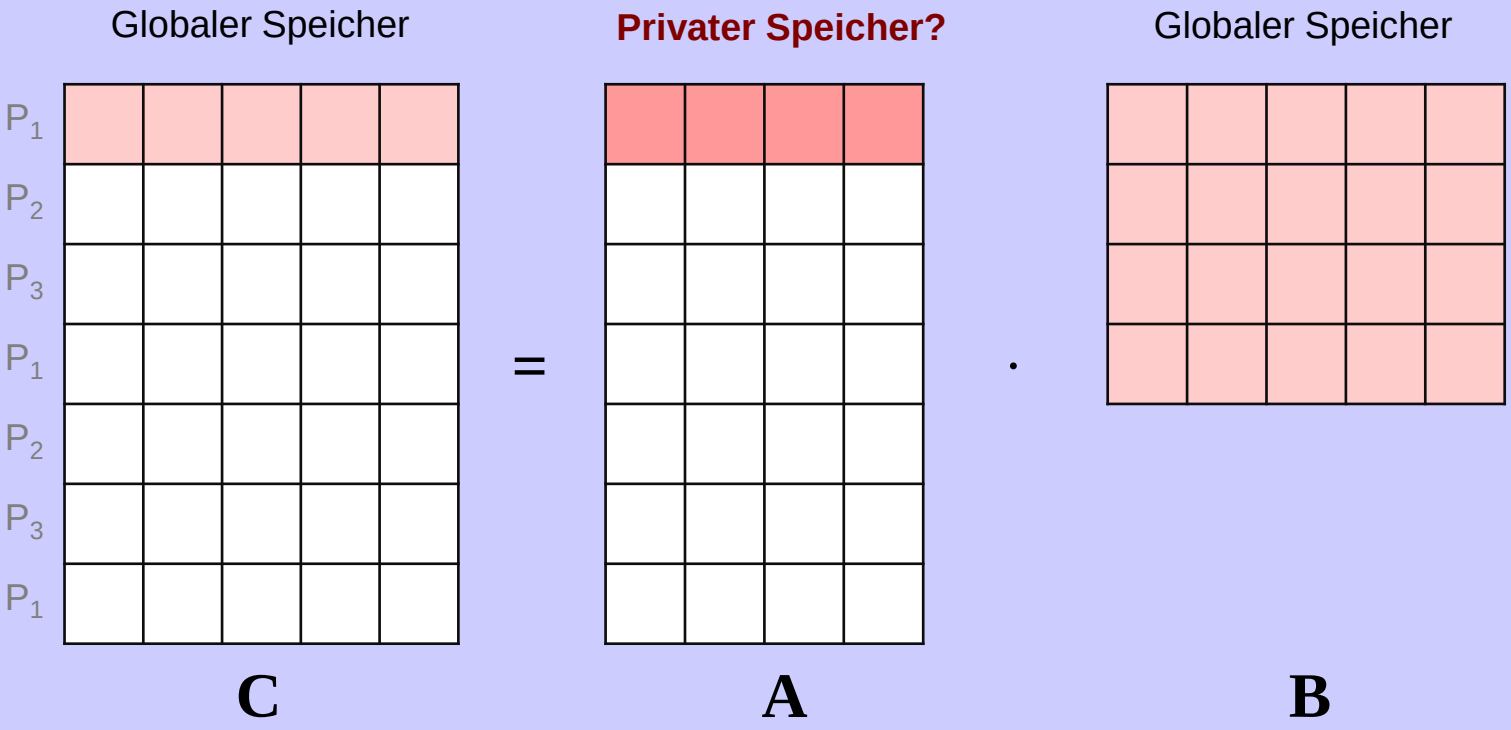
Version 4: Schleifen tauschen?

```
const char *KernelSource =
"#define DIM 1000                                                    // Size of matrix \n"
"__kernel void matmult(__global float *A, __global float *B, __global float *C) { \n"
"    int i, j, k;                                                                  \n"
"    float sum = 0.0;                                                              \n"
"    j = get_global_id(0);                                                         \n"
"    i = get_global_id(1);                                                         \n"
"    for (k = 0; k < DIM; k++)                                                     \n"
"        sum += A[i*DIM+k] * B[k*DIM+j];                                           \n"
"    C[i*DIM+j] = sum;                                                             \n"
"}                                                                                 \n"
"\n";
```

Speicheroptimierung?



Matrix-Multiplikation (zeilenweise)

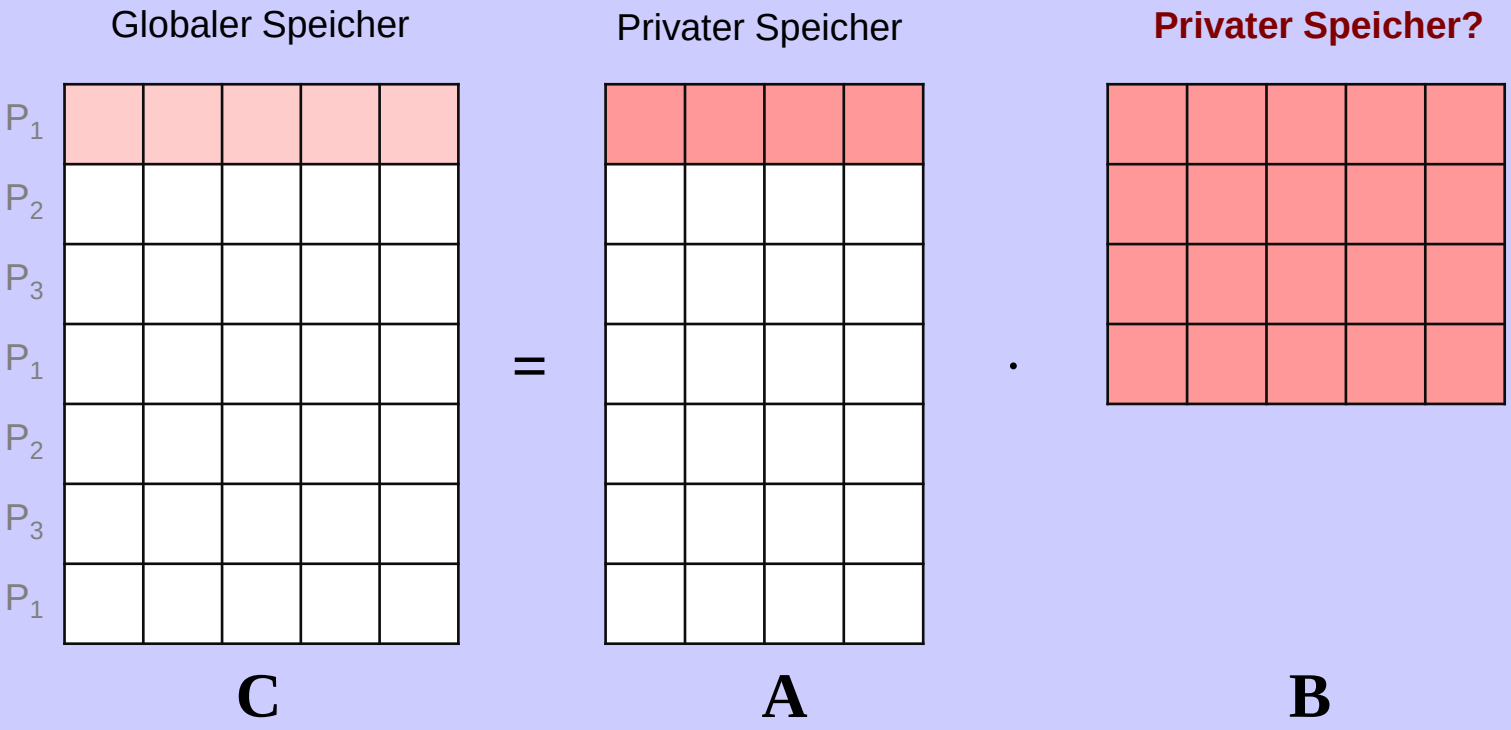


Version 5: C zeilenweise, A privat

```
const char *KernelSource =
"#define DIM 1000                                          // Groesse der Matrix \n"
"__kernel void matmult(__global float *A, __global float *B, __global float *C) { \n"
"    int k, j, i = get_global_id(0);                                              \n"
"    float Al[DIM], sum;                                 // Privater Zeilenpuffer \n"
"    for (k = 0; k < DIM; k++)                                                    \n"
"        Al[k] = A[i*DIM+k];                 // Zeile i von A in den Puffer kopieren \n"
"    for (j = 0; j < DIM; j++) {                                                  \n"
"        sum = 0.0;                                                               \n"
"        for (k = 0; k < DIM; k++)                                                \n"
"            sum += Al[k] * B[k*DIM+j];                                           \n"
"        C[i*DIM+j] = sum;                                                        \n"
"    }                                                                            \n"
"}                                                                                \n"
"\n";
```

```
void main(int argc, char **argv)
{
 size_t global[1] = {DIM};
 // ...
 clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL, global, NULL, 0, NULL, NULL);
```

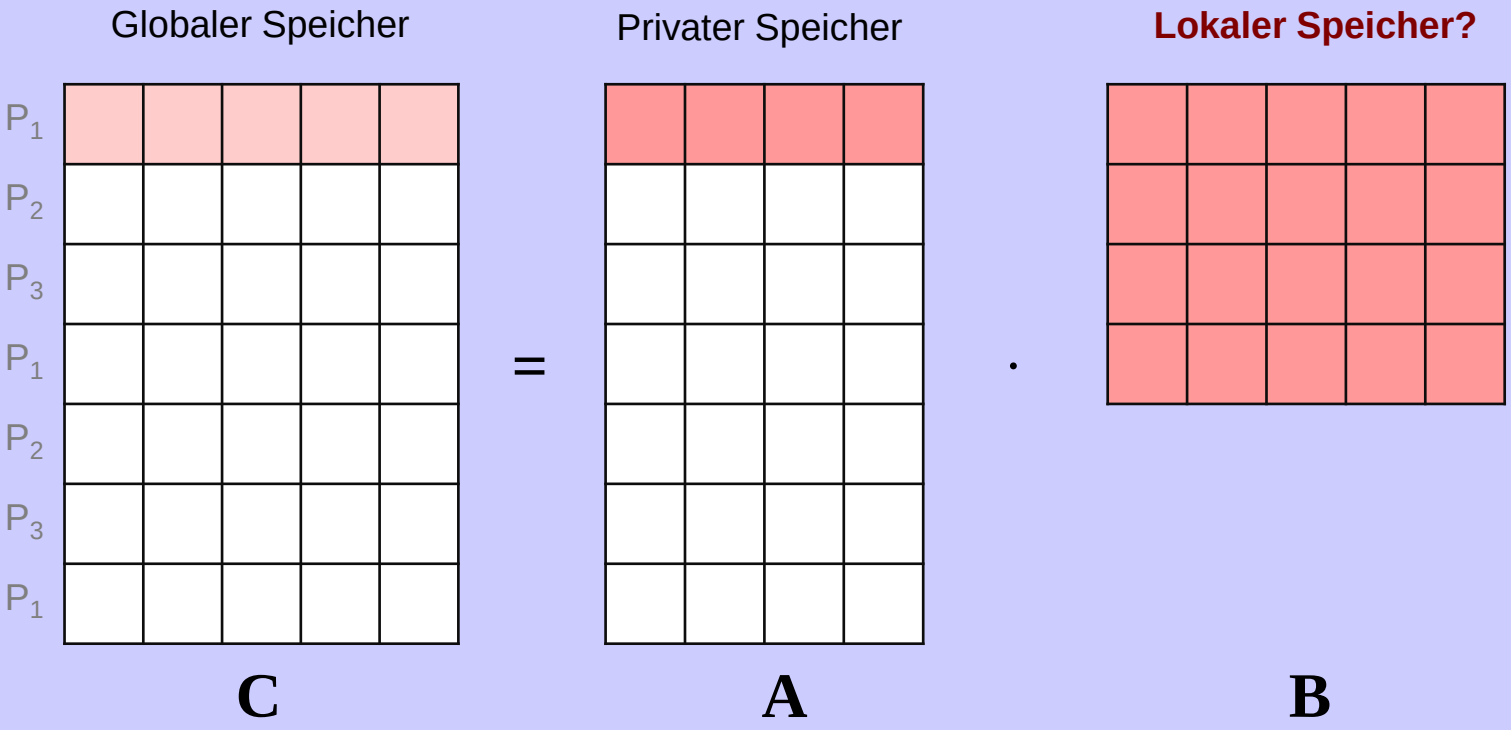
Matrix-Multiplikation (zeilenweise)



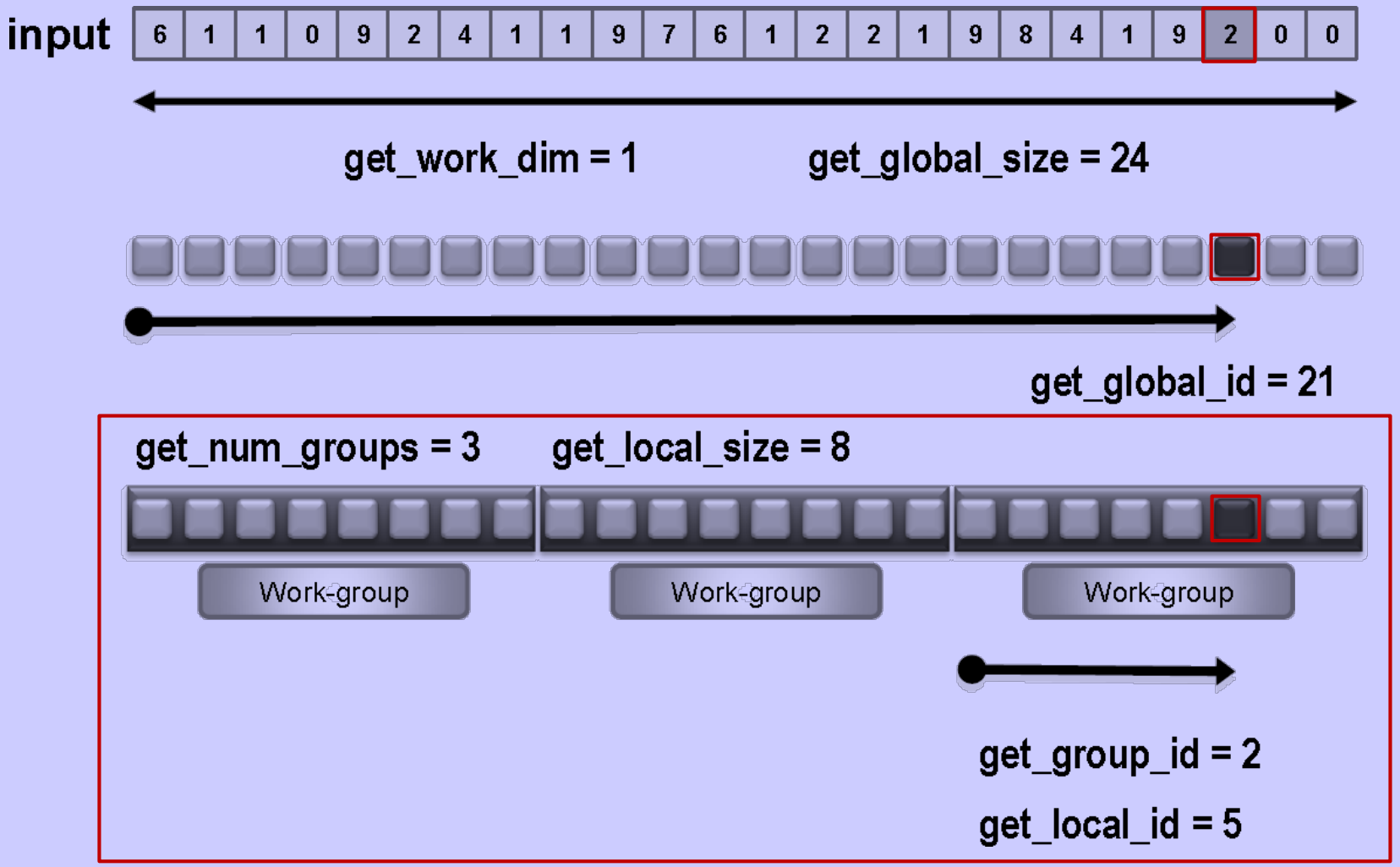
Versuch 1: C zeilenweise, A und B privat

```
const char *KernelSource =
"#define DIM 1000                                          // Groesse der Matrix \n"
"__kernel void matmult(__global float *A, __global float *B, __global float *C) { \n"
"    int k, j, i = get_global_id(0);                                               \n"
"    float Al[DIM], Bl[DIM], sum;                                                  \n"
"    for (k = 0; k < DIM; k++)                                                     \n"
"        Al[k] = A[i*DIM+k];                                                       \n"
"    for (j = 0; j < DIM; j++) {                                                   \n"
"        for (k = 0; k < DIM; k++)                                                 \n"
"            Bl[k] = B[k*DIM+j];                                                   \n"
"        sum = 0.0;                                                                \n"
"        for (k = 0; k < DIM; k++)                                                 \n"
"            sum += Al[k] * Bl[k];                                                 \n"
"        C[i*DIM+j] = sum;                                                         \n"
"    }                                                                             \n"
"}                                                                                 \n"
"\n";
```

Matrix-Multiplikation (zeilenweise)



Lokaler Speicher und Arbeitsgruppen



Versuch 2: C zeilenweise, A priv. B lokal

```
const char *KernelSource =
"#define DIM 1000                                          // Groesse der Matrix \n"
"__kernel void matmult(__global float *A, __global float *B, __global float *C,   \n"
"                      __local float *Bl) {                                       \n"
"    int k, j, i = get_global_id(0);                                              \n"
"    float Al[DIM], sum;                                                          \n"
"    int il = get_local_id(0);          // n work-items teilen sich das Kopieren \n"
"    int nl = get_local_size(0);                                                  \n"
"    for (k = 0; k < DIM; k++)                                                    \n"
"        Al[k] = A[i*DIM+k];                                                      \n"
"    for (j = 0; j < DIM; j++) {                                                  \n"
"        for (k = il; k < DIM; k += nl)     // nur jede n-te Zeile von B kopieren \n"
"            Bl[k] = B[k*DIM+j];                                                  \n"
"        sum = 0.0;                                                               \n"
"        for (k = 0; k < DIM; k++)                                                \n"
"            sum += Al[k] * Bl[k];                                                \n"
"        C[i*DIM+j] = sum;                                                        \n"
"    }                                                                            \n"
"}                                                                                \n"
"\n";
```

Versuch 2: C zeilenweise, A priv. B lokal

```
const char *KernelSource =
"#define DIM 1000                                          // Groesse der Matrix \n"
"__kernel void matmul(__global float *A, __global float *B, __global float *C,   \n"
"                     __local float *Bl) {                                        \n"
"    // ...                                                                      \n"
"}                                                                               \n"
"\n";
```

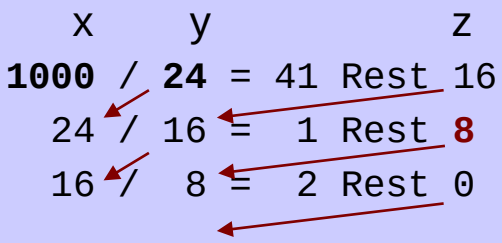
Explizite Definition der Work-group Größe (ganzzahlig teilbar):

```
void main(int argc, char **argv)
{
  size_t global[1] = {DIM}, local[1];
  // ...
  clGetDeviceInfo(device_id, CL_DEVICE_MAX_COMPUTE_UNITS, sizeof(size_t), local, NULL);
  local[0] = ggt(global[0], local[0]); // Groesster gemeinsamer Teiler ggt(1000, 24) = 8
  // ...
  clSetKernelArg(kernel, 3, sizeof(float)*DIM, NULL);
  clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL, global, local, 0, NULL, NULL);
```

Größter gemeinsamer Teiler

```
int ggt(int x, int y)
{
    int z;

    while (y) {
        z = x % y;
        x = y;
        y = z;
    }
    return x;
}
```

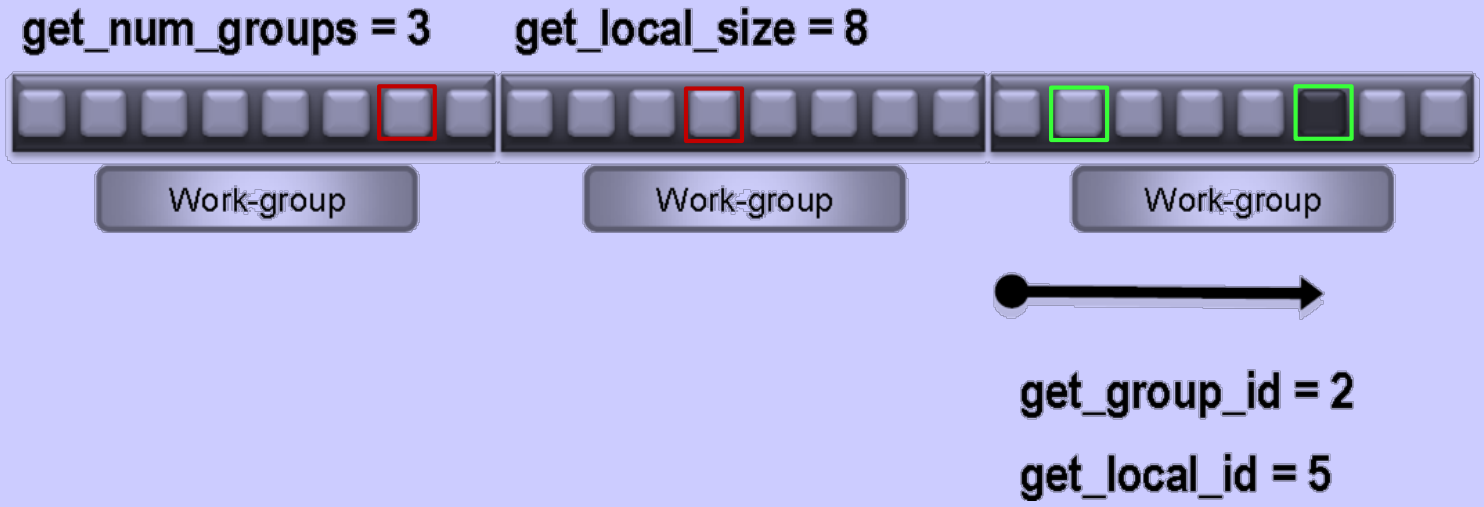
Beispiel:



Version 6: C zeilenweise, A priv. B lokal

```
const char *KernelSource =
"#define DIM 1000                                          // Groesse der Matrix \n"
"__kernel void matmult(__global float *A, __global float *B, __global float *C,   \n"
"                      __local float *Bl) {                                       \n"
"    int k, j, i = get_global_id(0);                                              \n"
"    float Al[DIM], sum;                                                          \n"
"    int il = get_local_id(0);                                                    \n"
"    int nl = get_local_size(0);                                                  \n"
"    for (k = 0; k < DIM; k++)                                                    \n"
"        Al[k] = A[i*DIM+k];                                                      \n"
"    for (j = 0; j < DIM; j++) {                                                  \n"
"        for (k = il; k < DIM; k+=nl)                                             \n"
"            Bl[k] = B[k*DIM+j];                                                  \n"
"        barrier(CLK_LOCAL_MEM_FENCE);       // Warten, dass alle B kopiert haben \n"
"        sum = 0.0;                                                               \n"
"        for (k = 0; k < DIM; k++)                                                \n"
"            sum += Al[k] * Bl[k];                                                \n"
"        C[i*DIM+j] = sum;                                                        \n"
"    }                                                                            \n"
"}                                                                                \n"
"\n";
```

Synchronisation und Arbeitsgruppen



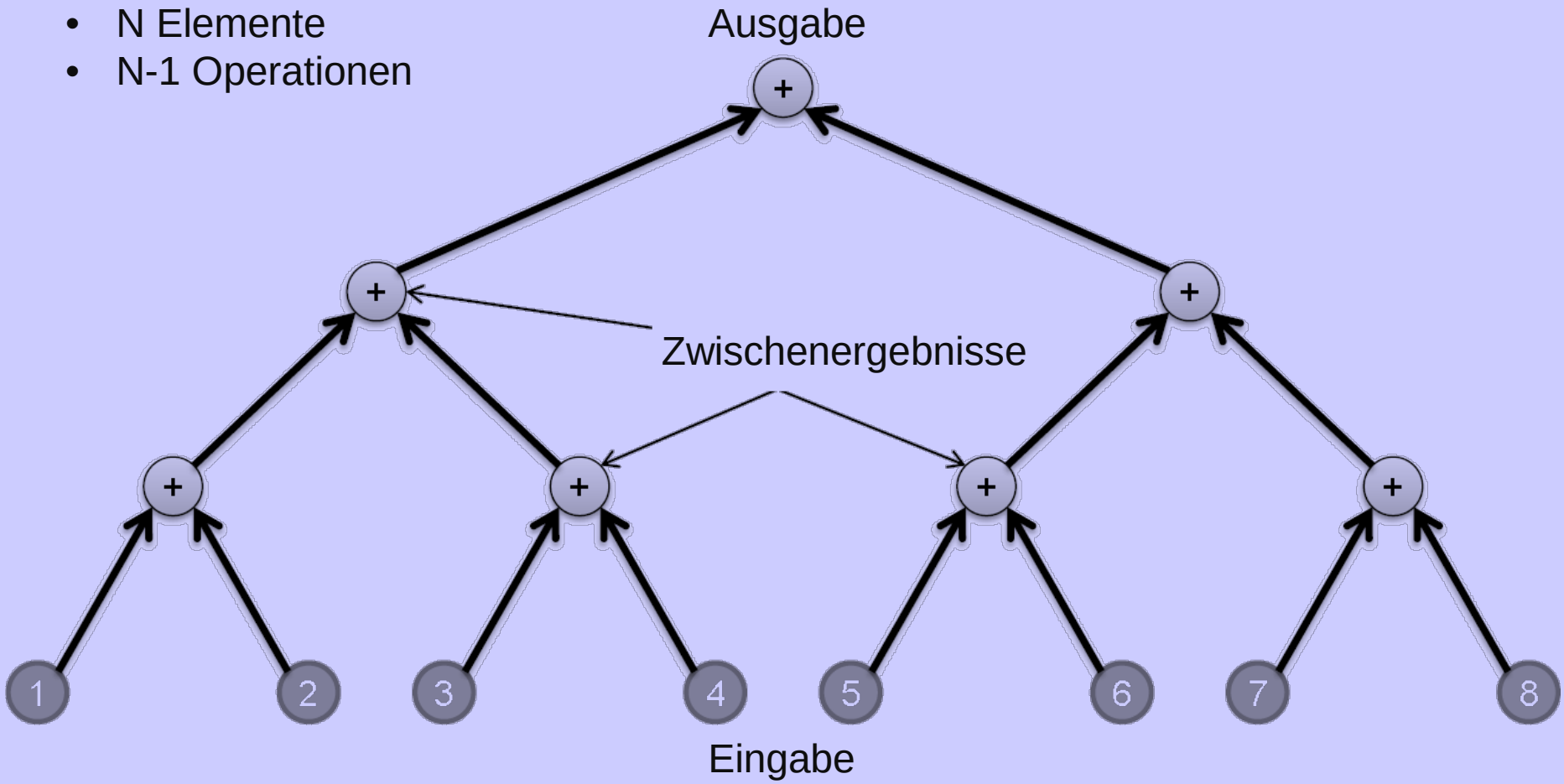
Beispiel für Synchronisation

- **Ziel**: Reduziere eine Zahlenmenge auf einen einzelnen Wert
- **Beispiel**: Finde die Summe aller Elemente in einem Feld
- **Sequentieller Code**

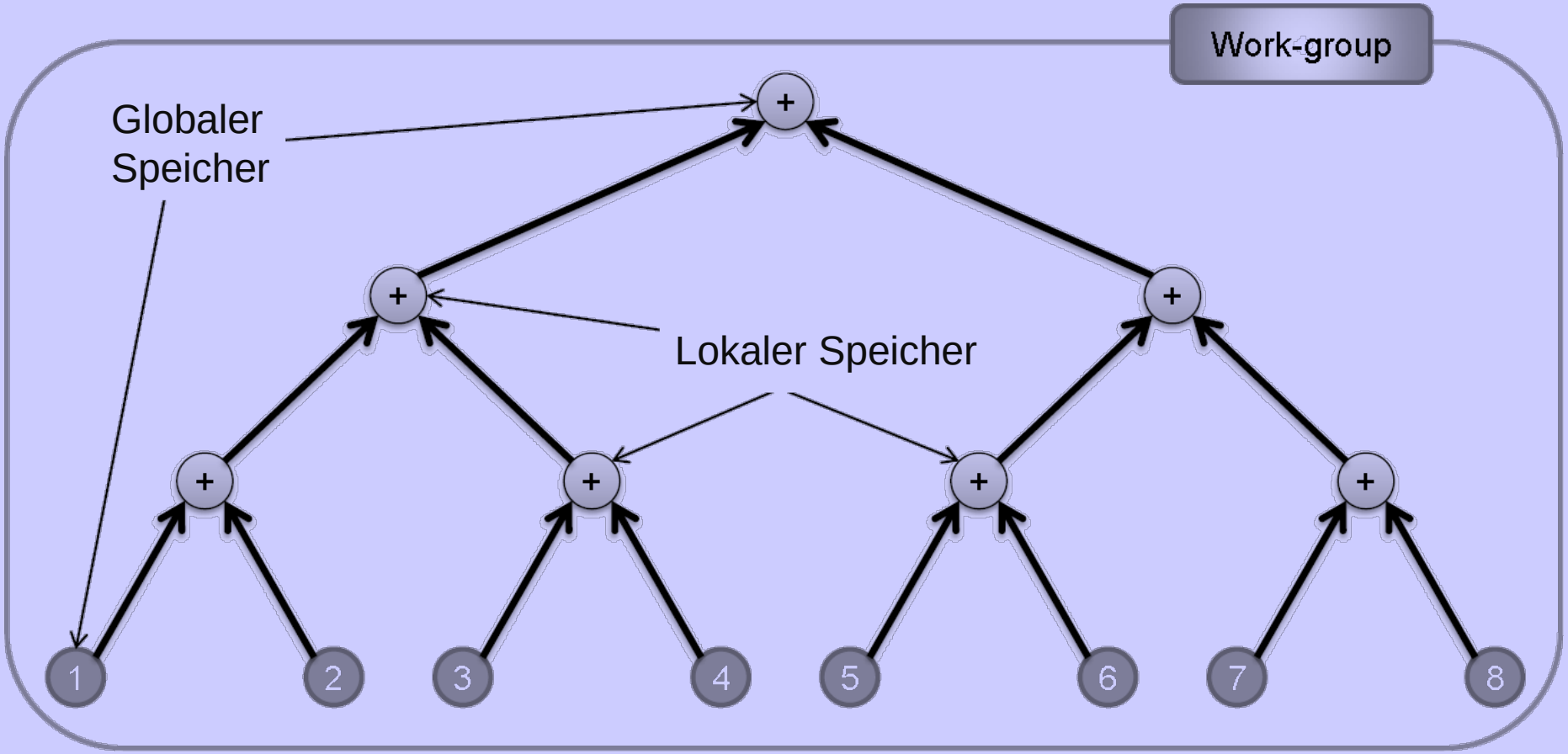
```
int reduce(int *A, int dim)
{
    int sum = 0;
    for (int i=0; i<dim; i++)
        sum += A[i];
    return sum;
}
```

Parallele Reduktion

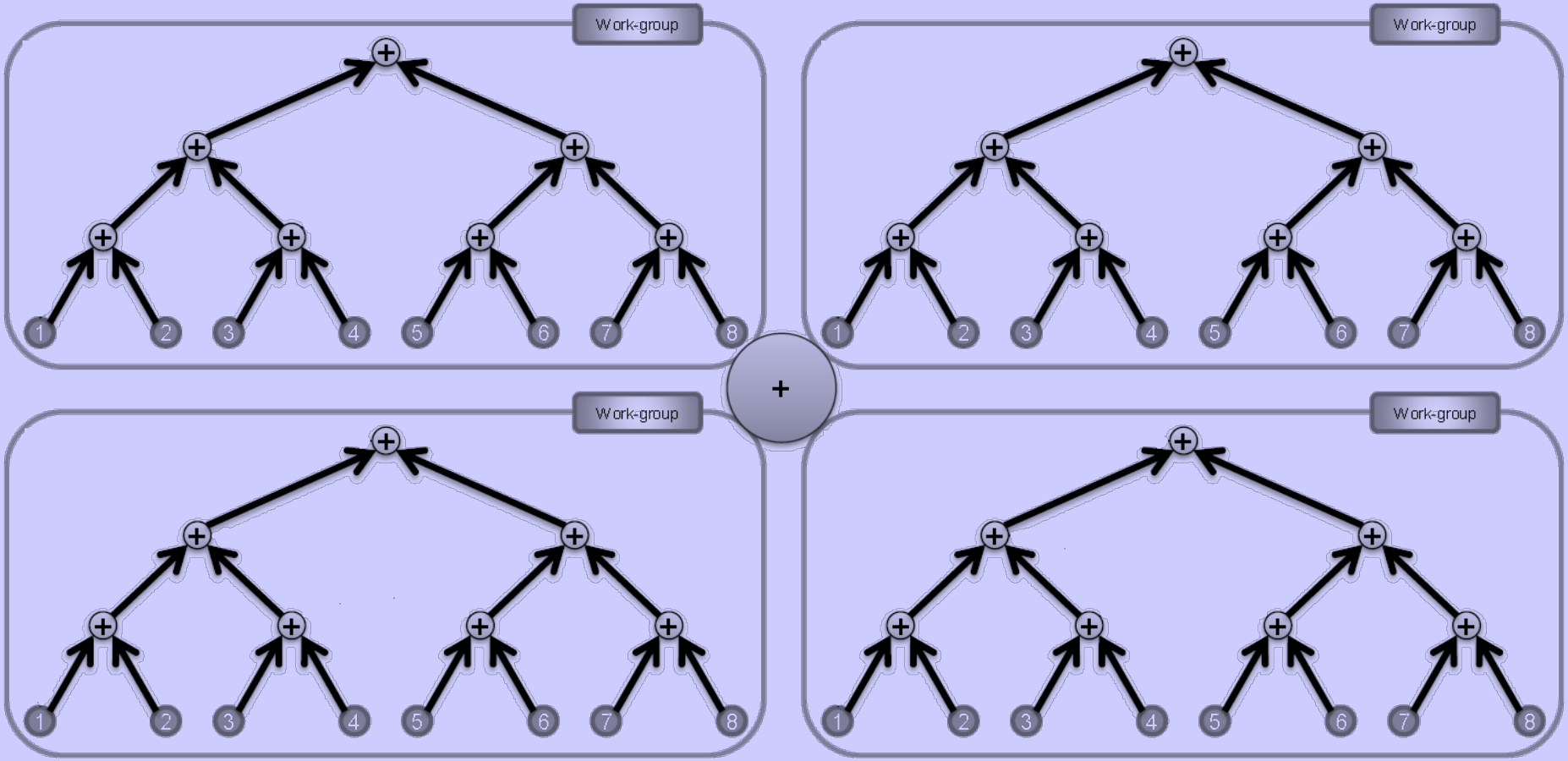
- N Elemente
- N-1 Operationen



Parallele Reduktion in OpenCL



Reduktion in OpenCL

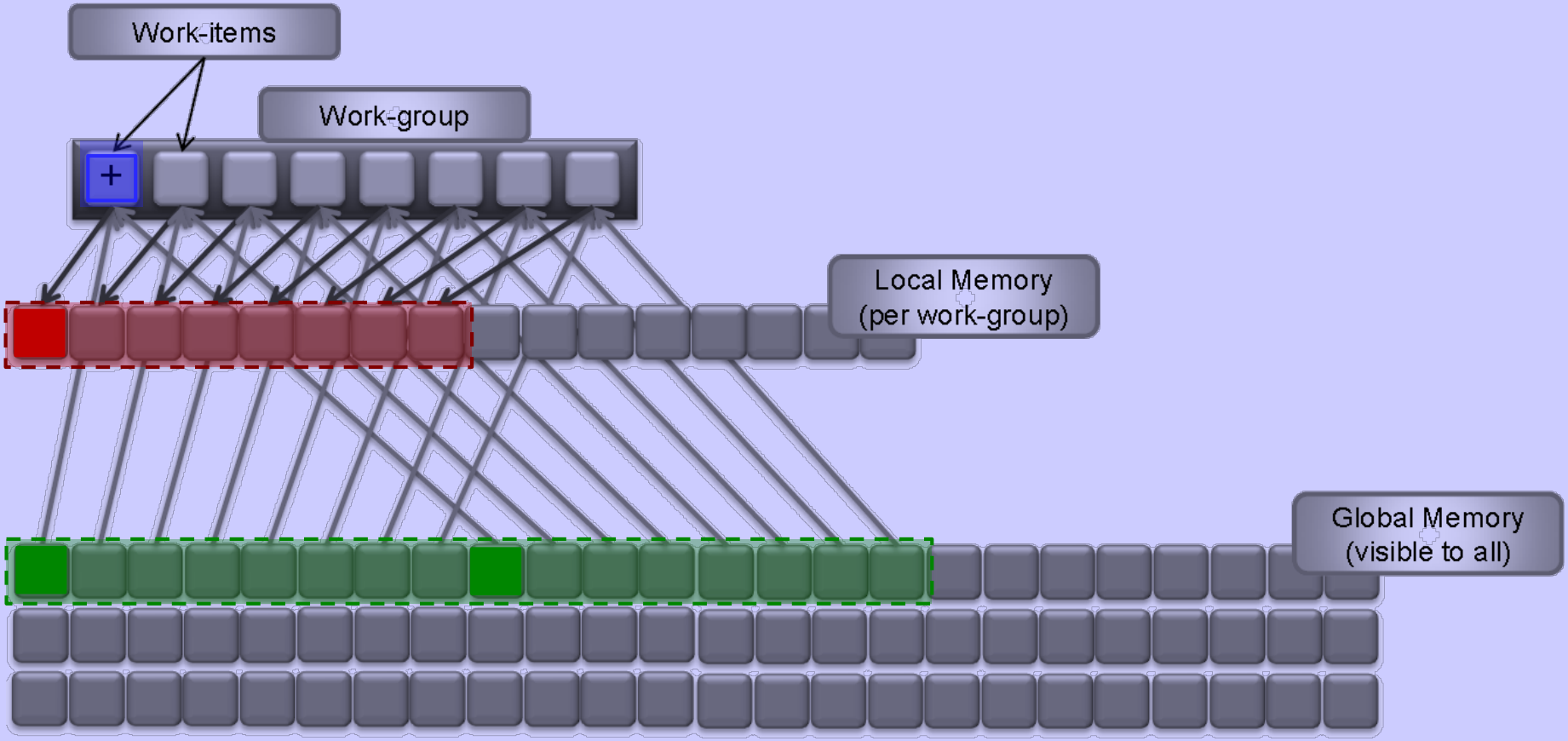


OpenCL Code 1

```
__kernel void
reduce(__global float *input,
       __global float *output,
       __local float *lmem)
{
  int gsize = get_global_size(0);
  int lsize = get_local_size(0);

  int gid   = get_global_id(0);
  int lid   = get_local_id(0);
  int oid   = get_group_id(0);

  // ...
```

```
// 1. Daten in lokalen Speicher kopieren
lmem[lid] = input[gid] + input[gid + gsize];
barrier(CLK_LOCAL_MEM_FENCE);
```

Schritt 1: In lokalen Speicher kopieren



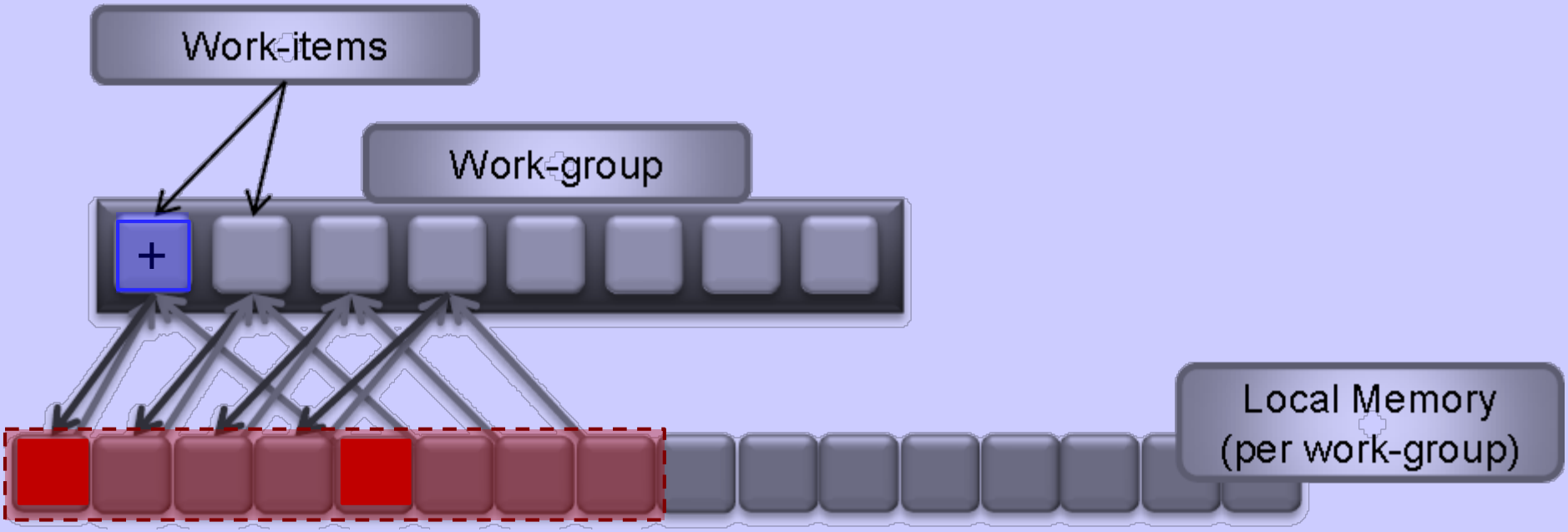
OpenCL Code 2

```
__kernel void
reduce(__global float *input,
       __global float *output,
       __local float *lmem)
{
  int gsize = get_global_size(0);
  int lsize = get_local_size(0);

  int gid   = get_global_id(0);
  int lid   = get_local_id(0);
  int oid   = get_group_id(0);

  // ...
```

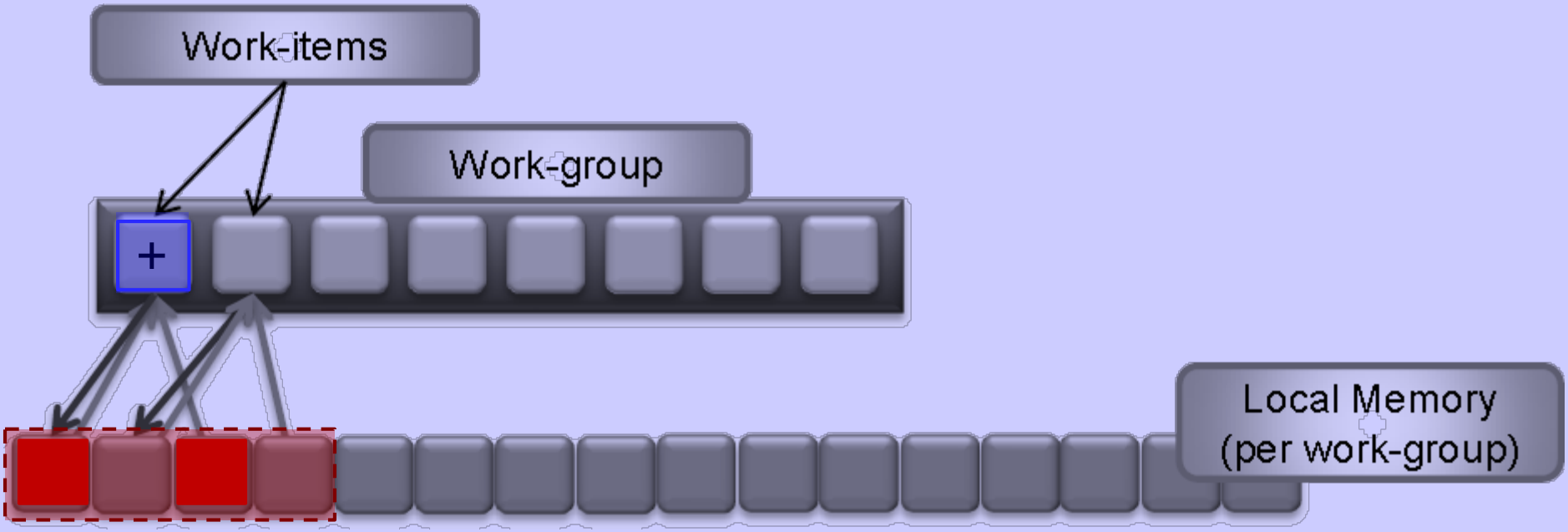
```
// 1. Daten in lokalen Speicher kopieren
lmem[lid] = input[gid] + input[gid + gsize];
barrier(CLK_LOCAL_MEM_FENCE);
```

```
// 2. Wiederhole lokale Reduktion
for (int s = lsize/2; s > 1; s /= 2) {
    if (lid < s)
        lmem[lid] += lmem[lid + s];
    barrier(CLK_LOCAL_MEM_FENCE);
}
```

Schritt 2: Lokale Reduktion



Schritt 2: Wiederholte Reduktion



OpenCL Code 3

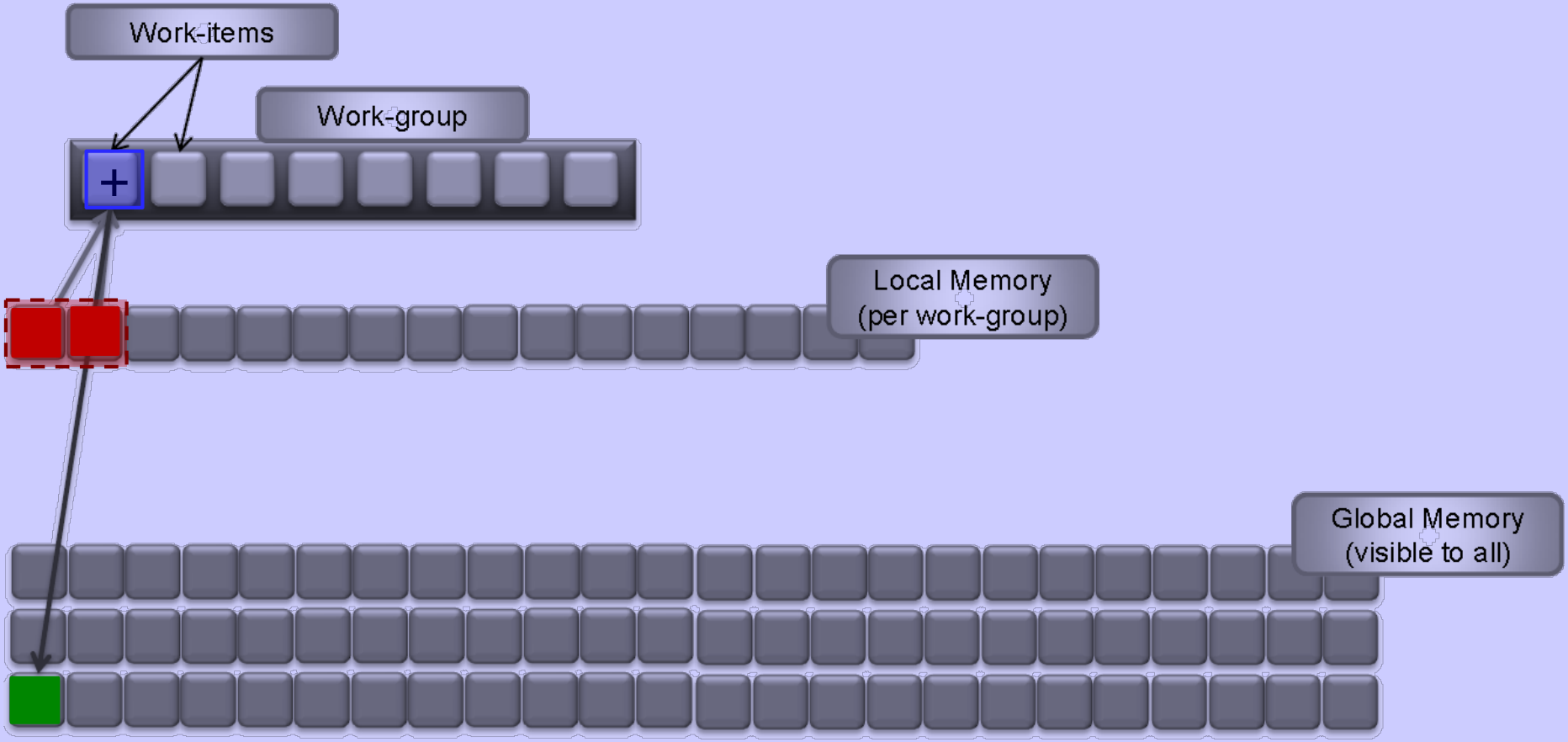
```
__kernel void
reduce(__global float *input,
       __global float *output,
       __local float *lmem)
{
  int gsize = get_global_size(0);
  int lsize = get_local_size(0);

  int gid   = get_global_id(0);
  int lid   = get_local_id(0);
  int oid   = get_group_id(0);

  // ...
```

```
// 1. Daten in lokalen Speicher kopieren
lmem[lid] = input[gid] + input[gid + gsize];
barrier(CLK_LOCAL_MEM_FENCE);
```

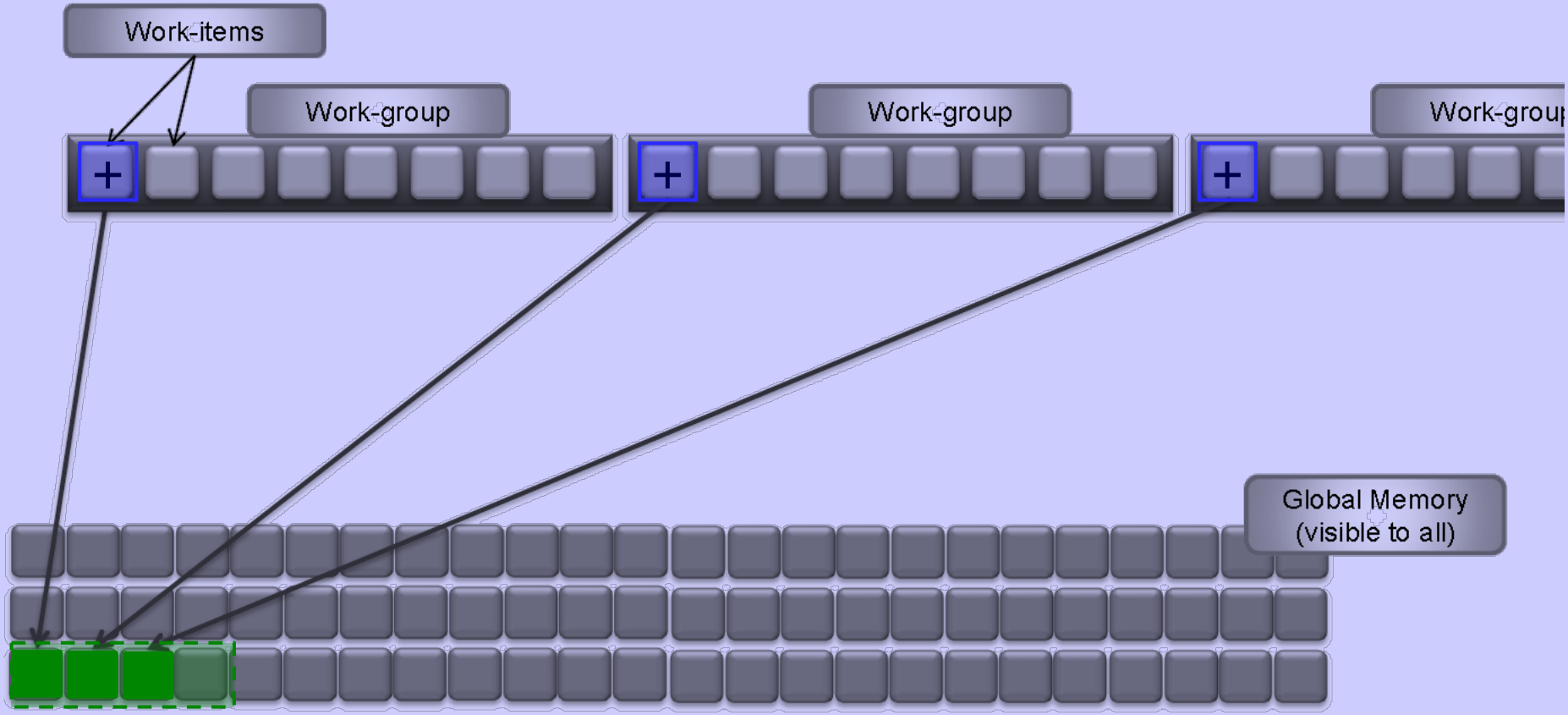
```
// 2. Wiederhole lokale Reduktion
for (int s = lsize/2; s > 1; s /= 2) {
    if (lid < s)
        lmem[lid] += lmem[lid + s];
    barrier(CLK_LOCAL_MEM_FENCE);
}
```

```
// 3. Schreibe Ergebnis in globalen Speicher
if (lid == 0)
    output[oid] = lmem[0] + lmem[1];
```

Schritt 3: Ergebnis zurückkopieren



Schritt 3: Alle Teilergebnisse



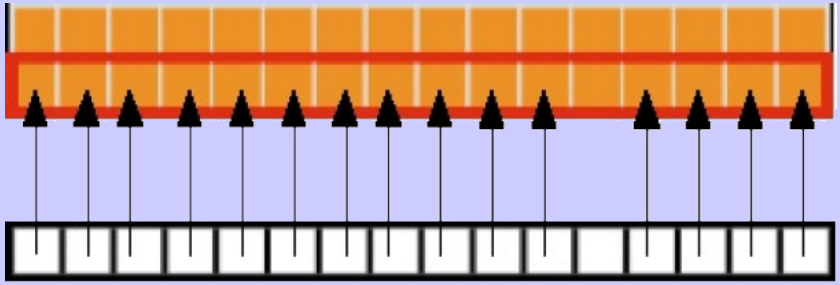
Die einzelnen Teilergebnisse der Work-groups mittels CPU aufsummieren oder den Kernel noch einmal innerhalb einer Work-group laufen lassen!

Bauhaus-
Universität
Weimar

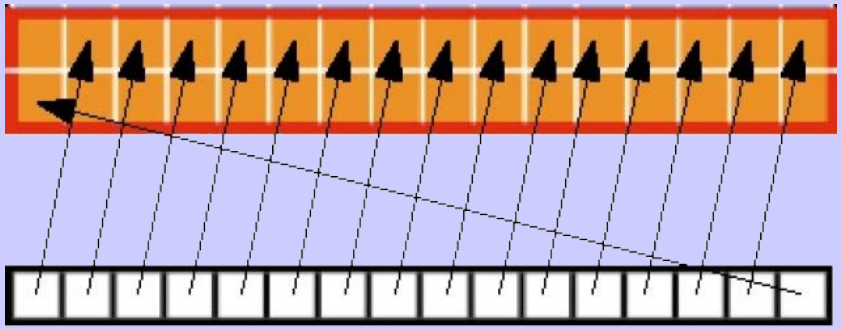
Speicherorganisation

Datenreihenfolge im Speicher

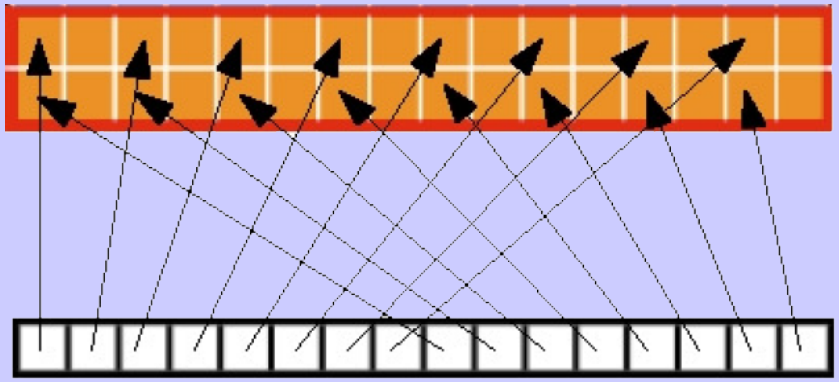
Alle Work-items können gleichzeitig auf Speicher zugreifen



Falsch ausgerichteter Speicher

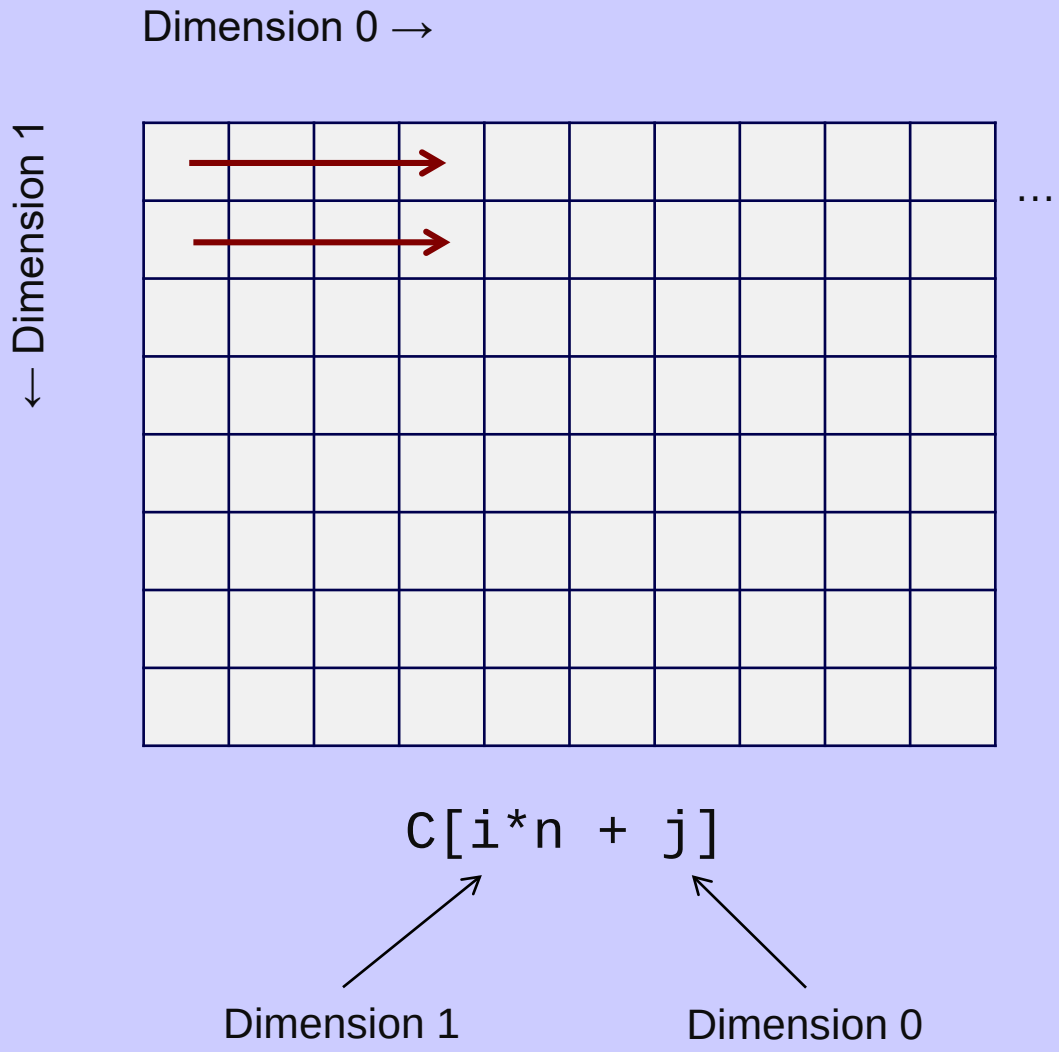


```
int i = get_global_id(0) + offset;
out[i] = in[i];
```



```
int i = get_global_id(0) * stride;
out[i] = in[i];
```

Matrix Datenreihenfolge



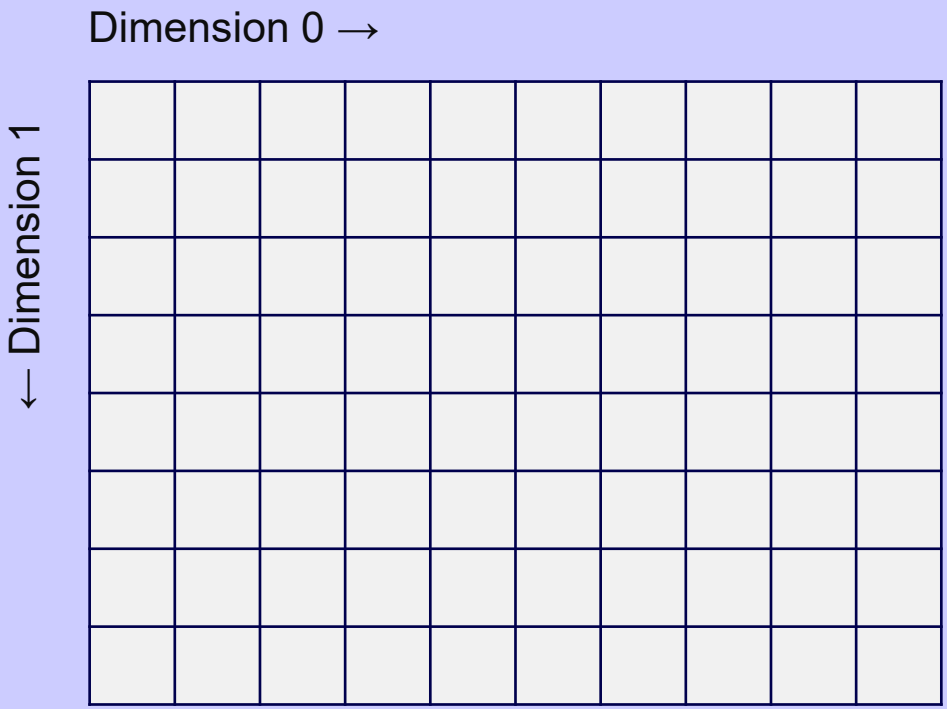
NDRange Kernel Offset

- Bestimme Topologie (NDRange) und führe Kernel aus
- **Parameter:**

```
cl_int clEnqueueNDRangeKernel(
       cl_command_queue command_queue,
       cl_kernel        kernel,
       cl_uint          work_dim,
       const size_t    *global_work_offset,
       const size_t    *global_work_size,
       const size_t    *local_work_size,
       cl_uint          num_events_in_wait_list,
       const cl_event  *event_wait_list,
       cl_event        *event)
```

Beispiel: Aufteilung des NDRange

- work_dim = 2, global_work_size = { 10, 8 }



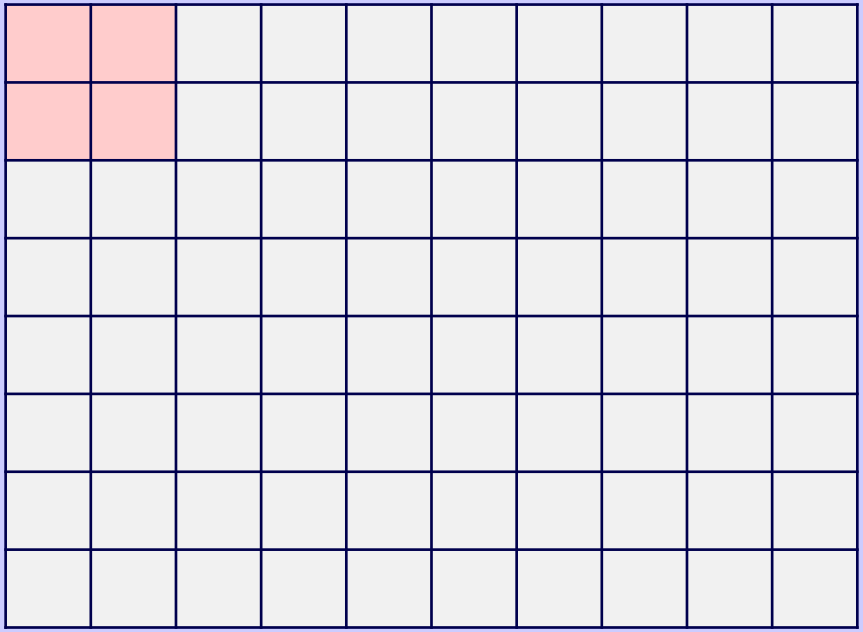
NDRange Kernel Offset

- Bestimme Topologie (NDRange) und führe Kernel aus
- **Parameter:**

```
cl_int clEnqueueNDRangeKernel(
       cl_command_queue command_queue,
       cl_kernel        kernel,
       cl_uint          work_dim,
       const size_t    *global_work_offset,
       const size_t    *global_work_size,
       const size_t    *local_work_size,
       cl_uint          num_events_in_wait_list,
       const cl_event  *event_wait_list,
       cl_event        *event)
```

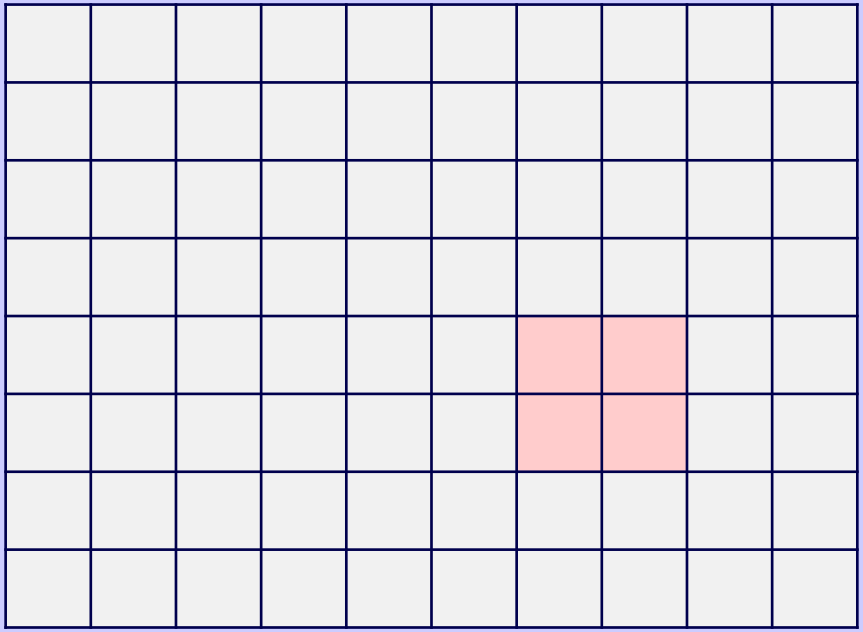
Beispiel: Aufteilung des NDRange

- offset = { 0, 0}, work_size = { 2, 2}



Beispiel: Aufteilung des NDRange

- offset = { 6, 4}, work_size = { 2, 2}



Aufteilen der Kernel-Aufrufe

```
#define PART 10                                          // DIM wird in PART Teile aufgeteilt

size_t part[2] = {DIM/PART, DIM/PART}, off[2];           // Teilbereiche von global[]
unsigned int i, j;

for (j = 0; j < PART; j++) {
   off[1] = j * part[1];
   for (i = 0; i < PART; i++) {
      off[0] = i * part[0];
      clEnqueueNDRangeKernel(command_queue, kernel, 2, off, part, NULL, 0, NULL, NULL);
   }
}
```

Problem beim Aufteilen: Global Size?

```
__kernel void matmult(__global float *A, __global float *B, __global float *C, int n)
{
    int i, j, k;
    float sum = 0;

    i = get_global_id(0);
    j = get_global_id(1);

    for (k = 0; k < n; k++) {
        sum += A[i*n+k] * B[k*n+j];
    }
    C[i*n+j] = sum;
}
```

Bemerkung:

- Dimension **n** der Matrix steht nicht mehr in **`get_global_size()`**, sondern muss als **`#define`** oder Funktionsargument übergeben werden.

Problem beim Aufteilen: Zeitmessung?

```
#define PART 10                                          // DIM wird in PART Teile aufgeteilt

cl_event event1, event2;                                    // Events für Zeitmessung
cl_int num = DIM;
size_t part[2] = {DIM/PART, DIM/PART}, off[2];           // Teilbereiche von global[]
unsigned int i, j;

clSetKernelArg(kernel, 3, sizeof(cl_int), &num);       // Matrixdimension als Parameter

for (j = 0; j < PART; j++) {
   off[1] = j * part[1];
   for (i = 0; i < PART; i++) {
      off[0] = i * part[0];
      clEnqueueNDRangeKernel(command_queue, kernel, 2, off, part, NULL, 0, NULL, &event2)
   }
   if ((i == 0) && (j == 0)) event1 = event2;                  // Ersten Event merken
}
clFinish(command_queue);

clGetEventProfilingInfo(event1, CL_PROFILING_COMMAND_START, sizeof(start), &start, NULL);
clGetEventProfilingInfo(event2, CL_PROFILING_COMMAND_END, sizeof(end), &end, NULL);
printf("gpu: %.1f ms\n", ((end - start) * 1.0e-6));
```

Bauhaus-
Universität
Weimar

Weitere Konzepte in OpenCL

Weitere Datentypen: Vektoren

`char`*n*	Ein Vektor von n 8-bit signed integer Werten
`uchar`*n*	Ein Vektor von n 8-bit unsigned integer Werten
`short`*n*	Ein Vektor von n 16-bit signed integer Werten
`ushort`*n*	Ein Vektor von n 16-bit unsigned integer Werten
`int`*n*	Ein Vektor von n 32-bit signed integer Werten
`uint`*n*	Ein Vektor von n 32-bit unsigned integer Werten
`long`*n*	Ein Vektor von n 64-bit signed integer Werten
`ulong`*n*	Ein Vektor von n 64-bit unsigned integer Werten
`float`*n*	Ein Vektor von n 32-bit floating-point Werten
`double`*n*	Ein Vektor von n 64-bit floating-point Werten
`half`*n*	Ein Vektor von n 16-bit floating-point Werten

n kann dabei `2, 3, 4, 8` oder `16` sein

Doppelte Genauigkeit

Fließkommazahl mit doppelter Genauigkeit (**double**, 8Byte)

```
#if defined(cl_khr_fp64)  // Khronos Erweiterung verfügbar?
#pragma OPENCL EXTENSION cl_khr_fp64 : enable
#elseif defined(cl_amd_fp64)  // AMD Erweiterung verfügbar?
#pragma OPENCL EXTENSION cl_amd_fp64 : enable
#endif
```

Vektordatentypen

```
// Ein Vektor vom Typ float4 mit 4 Gleitkommazahlen
float4 a = (float4)(0.0, 1.0, 2.0, 3.0);

// Ein Vektor vom Typ float2 mit 2 Gleitkommazahlen
float2 b = (float2)(1.0, 2.0);

// Äquivalent zu (0.0, 0.0)
float2 c = (float2)(0.0);

// Kombination von zwei float2 zu einem float4
float4 d = (float4)(b, c);
```

Arithmetische Operationen

```
// Deklaration der Variablen
float4 a, b, c;
int8 d, e, f, g;

// Komponentenweise Addition von zwei float4 Vektoren
c = a + b;

// Komponentenweise Multiplikation von zwei int8 Vektoren
f = d * e;

// Multiplikation aller Komponenten eines int8 Vektors mit einer Zahl
g = 2 * f;
```

Zugriff auf die Vektorkomponenten

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
float2 v;	v.x, v.s0	v.y, v.s1														
float4 v;	v.x, v.s0	v.y, v.s1	v.z, v.s2	v.w, v.s3												
float8 v;	v.s0	v.s1	v.s2	v.s3	v.s4	v.s5	v.s6	v.s7								
float16 v;	v.s0	v.s1	v.s2	v.s3	v.s4	v.s5	v.s6	v.s7	v.s8	v.s9	v.sa, v.sA	v.sb, v.sB	v.sc, v.sC	v.sd, v.sD	v.se, v.sE	v.sf, v.sF

Beispiele:

```
float2 pos;
pos.x = 1.0f;
pos.z = 1.0f; // ist illegal!

float4 c;
c.z = 1.0f;
c.xy = (float2)(3.0f, 4.0f);
c.xyzw = (float4)(1.0f, 2.0f, 3.0f, 4.0f);

float4 pos = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
float4 swiz = pos.wzyx; // (4, 3, 2, 1)
float4 dup = pos.xxyy;  // (1, 1, 2, 2)
```

Das **Mischen** ist nicht erlaubt:

```
float4 f;
float4 A = f.xs123;
float4 B = f.s012w;
```

Auswahl von Vektorkomponenten

	v.lo	v.hi	v.odd	v.even
float2	v.x, v.s0	v.y, v.s1	v.y, v.s1	v.x, v.s0
float4	v.s01, v.xy	v.s23, v.zw	v.s13, v.yw	v.s02, v.xz
float8	v.s0123	v.s4567	v.s1357	v.s0246
float16	v.s01234567	v.s89abcdef	v.s13579bdf	v.s02468ace

Mathematik Funktionen

Math Built-in Functions [6.11.2]

T is type float or float*n* (or optionally double, double*n*, or half*n*). int*n*, uint*n*, and ulong*n* must be scalar when *T* is scalar. The symbol **HN** indicates that Half and Native variants are available by prepending "half_" or "native_" to the function name, as in half_cos() and native_cos(). **Optional extensions enable double, double*n*, and half*n* types.**

T **acos** (*T*)		Arc cosine
T **acosh** (*T*)		Inverse hyperbolic cosine
T **acospi** (*T x*)		acos (*x*) / π
T **asin** (*T*)		Arc sine
T **asinh** (*T*)		Inverse hyperbolic sine
T **asinpi** (*T x*)		asin (*x*) / π
T **atan** (*T y_over_x*)		Arc tangent
T **atan2** (*T y*, *T x*)		Arc tangent of *y* / *x*
T **atanh** (*T*)		Hyperbolic arc tangent
T **atanpi** (*T x*)		atan (*x*) / π
T **atan2pi** (*T x*, *T y*)		atan2 (*x*, y) / π
T **cbrt** (*T*)		cube root
T **ceil** (*T*)		Round to integer toward + infinity
T **copysign** (*T x*, *T y*)		*x* with sign changed to sign of *y*
T **cos** (*T*)	HN	cosine
T **cosh** (*T*)		hyperbolic consine
T **cospi** (*T x*)		cos (π *x*)
T **half_divide** (*T x*, *T y*) *T* **native_divide** (*T x*, *T y*)		*x* / *y*
T **erfc** (*T*)		Complementary error function
T **erf** (*T*)		Calculates error function of *T*
T **exp** (*T x*)	HN	Exponential base e
T **exp2** (*T*)	HN	Exponential base 2
T **exp10** (*T*)	HN	Exponential base 10
T **expm1** (*T x*)		e^*x* -1.0
T **fabs** (*T*)		Absolute value
T **fdim** (*T x*, *T y*)		"Positive difference" between *x* and *y*
T **floor** (*T*)		Round to integer toward - infinity
T **fma** (*T a*, *T b*, *T c*)		Multiply and add, then round
T **fmax** (*T x*, *T y*) half*n* **fmax** (half*n x*, half)		Return *y* if *x* < *y*, otherwise it returns *x*
T **fmin** (*T x*, *T y*) half*n* **fmin** (half*n x*, half)		Return *y* if *y* < *x*, otherwise it returns *x*
T **fmod** (*T x*, *T y*)		Modulus. Returns *x* – *y* * trunc (*x*/*y*)
T **fract** (*T x*, *T* **iptr*)		Fractional value in *x*
T **frexp** (*T x*, int*n* **exp*)		Extract mantissa and exponent
T **hypot** (*T x*, *T y*)		square root of *x*^2+ *y*^2
int*n* **ilogb** (*T x*)		Return exponent as an integer value
T **ldexp** (*T x*, int*n n*) *T* **ldexp** (*T x*, int *n*)		*x* * 2^*n*
T **lgamma** (*T x*) *T* **lgamma_r** (*T* x, int*n* **signp*)		Log gamma function
T **log** (*T*)	HN	Natural logarithm
T **log2** (*T*)	HN	Base 2 logarithm
T **log10** (*T*)	HN	Base 10 logarithm
T **log1p** (*T x*)		ln (1.0 + *x*)
T **logb** (*T x*)		exponent of *x*
T **mad** (*T a*, *T b*, *T c*)		Approximates *a* * *b* + *c*
T **modf** (*T x*, *T* **iptr*)		Decompose a floating-point number
float **nan** (uint*n nancode*) float*n* **nan** (uint*n nancode*) half*n* **nan** (ushort*n nancode*) double*n* **nan** (ulong*n nancode*) double*n* **nan** (uint*n nancode*)		Quiet NaN
T **nextafter** (*T x*, *T y*)		Next representable floating-point value following *x* in the direction of *y*
T **pow** (*T x*, *T y*)		Compute *x* to the power of *y* (*x*^*y*)
T **pown** (*T x*, int*n y*)		Compute *x*^*y*, where *y* is an integer
T **powr** (*T x*, *T y*)	HN	Compute *x*^*y*, where *x* is >= 0
T **half_recip** (*T x*) *T* **native_recip** (*T x*)		1 / *x*
T **remainder** (*T x*, *T y*)		Floating point remainder function
T **remquo** (*T x*, *T y*, int*n* **quo*)		Floating point remainder and quotient function
T **rint** (*T*)		Round integer to nearest even integer
T **rootn** (*T x*, int*n y*)		Compute *x* to the power of 1/*y*
T **round** (*T x*)		Integral value nearest to *x* rounding
T **rsqrt** (*T*)	HN	Inverse square root
T **sin** (*T*)	HN	sine
T **sincos** (*T x*, *T* **cosval*)		sine and cosine of *x*
T **sinh** (*T*)		hyperbolic sine
T **sinpi** (*T x*)		sin (π *x*)
T **sqrt** (*T*)	HN	square root
T **tan** (*T*)	HN	tangent
T **tanh** (*T*)		hyperbolic tangent
T **tanpi** (*T x*)		tan (π *x*)
T **tgamma** (*T*)		gamma function
T **trunc** (*T*)		Round to integer toward zero

Beispiel: Neue Native-/Fast-Funktionen

```
kernel void regularFuncs()
{
   for (int i=0; i<5000; i++)
   {
       float a=1, b=2, c=3, d=4;
       float e = a*b+c;
       e = a*b+c*d;
       e = sin(a);
       e = cos(b);

       float4 vec1 = (float4)(1, 2, 3, 0);
       float4 vec2 = (float4)(-1, 3, 1, 0);
       float4 vec = distance(vec1, vec2);

       double x=1, y=2, z=3;
       double resp = x*y+z;
   }
}
```

```
kernel void nativeFuncs()
{
   for (int i=0; i<5000; i++)
   {
       float a=1, b=2, c=3, d=4;
       float e = mad(a,b,c);
       e = mad(a,b,c*d);
       e = native_sin(a);
       e = native_cos(b);

       float4 vec1 = (float4)(1, 2, 3, 0);
       float4 vec2 = (float4)(-1, 3, 1, 0);
       float4 vec = fast_distance(vec1, vec2);

       double x=1, y=2, z=3;
       double resp = mad(x,y,z);
   }
}
```

Geometrie Funktionen

Geometric Built-in Functions [6.11.5]

Vector types may have 2 or 4 components. **Optional extensions enable double, double*n*, and half*n* types.**

float4 **cross** (float4 *p0*, float4 *p1*) double4 **cross** (double4 *p0*, double4 *p1*) half4 **cross** (half4 *p0*, half4 *p1*)	Cross product
float **dot** (float *p0*, float *p1*) float **dot** (float*n* *p0*, float*n* *p1*) double **dot** (double *p0*, double *p1*) double **dot** (double*n* *p0*, double*n* *p1*) half **dot** (half *p0*, half *p1*) half **dot** (half*n* *p0*, half*n* *p1*)	Dot product
float **distance** (float *p0*, float *p1*) float **distance** (float*n* *p0*, float*n* *p1*) double **distance** (double *p0*, double *p1*) double **distance** (double*n* *p0*, double*n* *p1*) half **distance** (half *p0*, half *p1*) half **distance** (half*n* *p0*, half*n* *p1*)	Vector distance
float **length** (float *p*) float **length** (float*n* *p*) double **length** (double *p*) double **length** (double*n* *p*) half **length** (half *p*) half **length** (half*n* *p*)	Vector length
float **normalize** (float *p*) float*n* **normalize** (float*n* *p*) double **normalize** (double *p*) double*n* **normalize** (double*n* *p*) half **normalize** (half *p*) half*n* **normalize** (half*n* *p*)	Normal vector length 1
float **fast_distance** (float *p0*, float *p1*) float **fast_distance** (float*n* *p0*, float*n* *p1*)	Vector distance
float **fast_length** (float *p*) float **fast_length** (float*n* *p*)	Vector length
float **fast_normalize** (float *p*) float*n* **fast_normalize** (float*n* *p*)	Normal vector length 1

Common Built-in Functions [6.11.4]

T is type float or float*n* (or optionally double, double*n*, or half*n*). **Optional extensions enable double, double*n*, and half*n* types.**

T **clamp** (*T x*, *T min*, *T max*) float*n* **clamp** (float*n* *x*, float *min*, float *max*) double*n* **clamp** (double*n* *x*, double *min*, double *max*) half*n* **clamp** (half*n* *x*, half *min*, half *max*)	Clamp *x* to range given by *min*, *max*
T **degrees** (*T radians*)	*radians* to degrees
T **max** (*T x*, *T y*) float*n* **max** (float*n* *x*, float *y*) double*n* **max** (double*n* *x*, double *y*) half*n* **max** (half*n* *x*, half *y*)	Max of *x* and *y*
T **min** (*T x*, *T y*) float*n* **min** (float*n* *x*, float *y*) double*n* **min** (double*n* *x*, double *y*) half*n* **min** (half*n* *x*, half *y*)	Min of *x* and *y*
T **mix** (*T x*, *T y*), *T a*) float*n* **mix** (float*n* *x*, float *y*), *float a*) double*n* **mix** (double*n* *x*, double *y*), *a*) half*n* **mix** (half*n* *x*, half *y*), *a*)	Linear blend of *x* and *y*
T **radians** (T *degrees*)	*degrees* to radians
T **step** (*T edge*, *T x*) float*n* **step** (float *edge*, float*n* *x*) double*n* **step** (double *edge*, double*n* *x*) half*n* **step** (half *edge*, half*n* *x*)	0.0 if *x* < *edge*, else 1.0
T **smoothstep** (*T edge0*, *T edge1*, *T x*) float*n* **smoothstep** (float *edge0*, float *edge1*, float*n* *x*) double*n* **smoothstep** (double *edge0*, double *edge1*, double*n* *x*) half*n* **smoothstep** (half *edge0*, half *edge1*, half*n* *x*)	Step and interpolate
T **sign** (*T x*)	Sign of *x*

Vergleichsoperatoren 1

Relational Built-in Functions [6.11.6]

T is type float, floatn, char, charn, uchar, ucharn, short, shortn, ushort, ushortn, int, intn, uint, uintn, long, longn, ulong, or ulongn (and optionally double, doublen). *S* is type char, charn, short, shortn, int, intn, long, or longn. *U* is type uchar, ucharn, ushort, ushortn, uint, uintn, ulong, or ulongn. Optional extensions enable double, doublen, and halfn types.

int **isequal** (float *x*, float *y*) intn **isequal** (floatn *x*, floatn *y*) int **isequal** (double *x*, double *y*) longn **isequal** (doublen *x*, doublen *y*) int **isequal** (half *x*, half *y*) shortn **isequal** (halfn *x*, halfn *y*)	Compare of *x* == *y*
int **isnotequal** (float *x*, float *y*) intn **isnotequal** (floatn *x*, floatn *y*) int **isnotequal** (double *x*, double *y*) longn **isnotequal** (doublen *x*, doublen *y*) int **isnotequal** (half *x*, half *y*) shortn **isnotequal** (halfn *x*, halfn *y*)	Compare of *x* != *y*
int **isgreater** (float *x*, float *y*) intn **isgreater** (floatn *x*, floatn *y*) int **isgreater** (double *x*, double *y*) longn **isgreater** (doublen *x*, doublen *y*) int **isgreater** (half *x*, half *y*) shortn **isgreater** (halfn *x*, halfn *y*)	Compare of *x* > *y*
int **isgreaterequal** (float *x*, float *y*) intn **isgreaterequal** (floatn *x*, floatn *y*) int **isgreaterequal** (double *x*, double *y*) longn **isgreaterequal** (doublen *x*, doublen *y*) int **isgreaterequal** (half *x*, half *y*) shortn **isgreaterequal** (halfn *x*, halfn *y*)	Compare of *x* >= *y*
int **isinf** (float) intn **isinf** (floatn) int **isinf** (double) longn **isinf** (doublen) int **isinf** (half) shortn **isinf** (halfn)	Test for +ve or –ve infinity
int **isnan** (float) intn **isnan** (floatn) int **isnan** (double) longn **isnan** (doublen) int **isnan** (half) shortn **isnan** (halfn)	Test for a NaN
int **isnormal** (float) intn **isnormal** (floatn) int **isnormal** (double) longn **isnormal** (doublen) int **isnormal** (half) shortn **isnormal** (halfn)	Test for a normal value
int **isordered** (float *x*, float *y*) intn **isordered** (floatn *x*, floatn *y*) int **isordered** (double *x*, double *y*) longn **isordered** (doublen *x*, doublen *y*) int **isordered** (half *x*, half *y*) shortn **isordered** (halfn *x*, halfn *y*)	Test if arguments are ordered
int **isunordered** (float *x*, float *y*) intn **isunordered** (floatn *x*, floatn *y*) int **isunordered** (double *x*, double *y*) longn **isunordered** (doublen *x*, doublen *y*) int **isunordered** (half *x*, half *y*) shortn **isunordered** (halfn *x*, halfn *y*)	Test if arguments are unordered

Vergleichsoperatoren 2

int **isless** (float *x*, float *y*) int*n* **isless** (float*n* *x*, float*n* *y*) int **isless** (double *x*, double *y*) long*n* **isless** (double*n* *x*, double*n* *y*) int **isless** (half *x*, half *y*) short*n* **isless** (half*n* *x*, half*n* *y*)	Compare of *x* < *y*
int **islessequal** (float *x*, float *y*) int*n* **islessequal** (float*n* *x*, float*n* *y*) int **islessequal** (double *x*, double *y*) long*n* **islessequal** (double*n* *x*, double*n* *y*) int **islessequal** (half *x*, half *y*) short*n* **islessequal** (half*n* *x*, half*n* *y*)	Compare of *x* <= *y*
int **islessgreater** (float *x*, float *y*) int*n* **islessgreater** (float*n* *x*, float*n* *y*) int **islessgreater** (double *x*, double *y*) long*n* **islessgreater** (double*n* *x*, double*n* *y*) int **islessgreater** (half *x*, half *y*) short*n* **islessgreater** (half*n* *x*, half*n* *y*)	Compare of (*x* < *y*) || (*x* > *y*)
int **isfinite** (float) int*n* **isfinite** (float*n*) int **isfinite** (double) long*n* **isfinite** (double*n*) int **isfinite** (half) short*n* **isfinite** (half*n*)	Test for finite value

int **signbit** (float) int*n* **signbit** (float*n*) int **signbit** (double) long*n* **signbit** (double*n*) int **signbit** (half) short*n* **signbit** (half*n*)	Test for sign bit
int **any** (*S x*)	1 if MSB in any component of *x* is set; else 0
int **all** (*S x*)	1 if MSB in all components of *x* are set; else 0
T **bitselect** (*T a*, *T b*, *T c*) half*n* **bitselect** (half*n a*, half*n b*, half*n c*) double*n* **bitselect** (double*n a*, double*n b*, double*n c*)	Each bit of result is corresponding bit of *a* if corresponding bit of *c* is 0
T **select** (*T a*, *T b*, *S c*) *T* **select** (*T a*, *T b*, *U c*) double*n* **select** (double*n*, double*n*, long*n*)	For each component of a vector type, result[*i*] = if MSB of *c*[*i*] is set ? *b*[*i*] : *a*[*i*] For scalar type, result = *c* ? *b* : *a*

2D und 3D Bildobjekte

Image Objects

Create Image Objects [5.2.4]

```
cl_mem clCreateImage2D (
    cl_context context, cl_mem_flags flags,
    const cl_image_format *image_format,
    size_t image_width, size_t image_height,
    size_t image_row_pitch, void *host_ptr,
    cl_int *errcode_ret)
```

flags: CL_MEM_READ_WRITE, CL_MEM_WRITE_ONLY, CL_MEM_READ_ONLY, CL_MEM_USE_HOST_PTR, CL_MEM_ALLOC_HOST_PTR, CL_MEM_COPY_HOST_PTR

```
cl_mem clCreateImage3D (
    cl_context context, cl_mem_flags flags,
    const cl_image_format *image_format,
    size_t image_width, size_t image_height,
    size_t image_depth, size_t image_row_pitch,
    size_t image_slice_pitch, void *host_ptr,
    cl_int *errcode_ret)
```

flags: CL_MEM_READ_WRITE, CL_MEM_WRITE_ONLY, CL_MEM_READ_ONLY, CL_MEM_USE_HOST_PTR, CL_MEM_ALLOC_HOST_PTR, CL_MEM_COPY_HOST_PTR

Query List of Supported Image Formats [5.2.5]

```
cl_int clGetSupportedImageFormats (
    cl_context context, cl_mem_flags flags,
    cl_mem_object_type image_type,
    cl_uint num_entries,
    cl_image_format *image_formats,
    cl_uint *num_image_formats)
```

flags: CL_MEM_READ_WRITE, CL_MEM_WRITE_ONLY, CL_MEM_READ_ONLY, CL_MEM_USE_HOST_PTR, CL_MEM_ALLOC_HOST_PTR, CL_MEM_COPY_HOST_PTR

Copy Between Image and Buffer Objects [5.2.7]

```
cl_int clEnqueueCopyImageToBuffer (
    cl_command_queue command_queue,
    cl_mem src_image, cl_mem dst_buffer,
    const size_t src_origin[3], const size_t region[3],
    size_t dst_offset, cl_uint num_events_in_wait_list,
    const cl_event *event_wait_list, cl_event *event)
```

```
cl_int clEnqueueCopyBufferToImage (
    cl_command_queue command_queue,
    cl_mem src_buffer, cl_mem dst_image,
    size_t src_offset, const size_t dst_origin[3],
    const size_t region[3],
    cl_uint num_events_in_wait_list,
    const cl_event *event_wait_list, cl_event *event)
```

Map and Unmap Image Objects [5.2.8]

```
void * clEnqueueMapImage (
    cl_command_queue command_queue,
    cl_mem image, cl_bool blocking_map,
    cl_map_flags map_flags, const size_t origin[3],
    const size_t region[3], size_t *image_row_pitch,
    size_t *image_slice_pitch,
    cl_uint num_events_in_wait_list,
    const cl_event *event_wait_list,
    cl_event *event, cl_int *errcode_ret)
```

Read, Write, Copy Image Objects [5.2.6]

```
cl_int clEnqueueReadImage (
    cl_command_queue command_queue,
    cl_mem image, cl_bool blocking_read,
    const size_t origin[3], const size_t region[3],
    size_t row_pitch, size_t slice_pitch, void *ptr,
    cl_uint num_events_in_wait_list,
    const cl_event *event_wait_list, cl_event *event)
```

```
cl_int clEnqueueWriteImage (
    cl_command_queue command_queue,
    cl_mem image, cl_bool blocking_write,
    const size_t origin[3], const size_t region[3],
    size_t input_row_pitch, size_t input_slice_pitch,
    const void *ptr, cl_uint num_events_in_wait_list,
    const cl_event *event_wait_list, cl_event *event)
```

```
cl_int clEnqueueCopyImage (
    cl_command_queue command_queue,
    cl_mem src_image, cl_mem dst_image,
    const size_t src_origin[3], const size_t dst_origin[3],
    const size_t region[3],
    cl_uint num_events_in_wait_list,
    const cl_event *event_wait_list, cl_event *event)
```

Query Image Objects [5.2.9]

```
cl_int clGetMemObjectInfo (cl_mem memobj,
    cl_mem_info param_name,
    size_t param_value_size, void *param_value,
    size_t *param_value_size_ret)
```

param_name: CL_MEM_TYPE, CL_MEM_FLAGS, CL_MEM_SIZE, CL_MEM_HOST_PTR, CL_MEM_MAP_COUNT, CL_MEM_REFERENCE_COUNT, CL_MEM_CONTEXT

```
cl_int clGetImageInfo (cl_mem image,
    cl_image_info param_name,
    size_t param_value_size, void *param_value,
    size_t *param_value_size_ret)
```

param_name: CL_IMAGE_FORMAT, CL_IMAGE_ELEMENT_SIZE, CL_IMAGE_ROW_PITCH, CL_IMAGE_SLICE_PITCH, CL_IMAGE_HEIGHT, CL_IMAGE_WIDTH, CL_IMAGE_DEPTH

Image Formats [5.2.4.1, 9.8]

Supported image formats: image_channel_order with image_channel_data_type.

Built-in support:

CL_RGBA: CL_HALF_FLOAT, CL_FLOAT, CL_UNORM_INT{8|16}, CL_SIGNED_INT{8|16|32}, CL_UNSIGNED_INT{8|16|32}
CL_BGRA: CL_UNORM_INT8

Optional support:

CL_R, CL_A: CL_HALF_FLOAT, CL_FLOAT, CL_UNORM_INT{8|16}, CL_SIGNED_INT{8|16|32}, CL_UNSIGNED_INT{8|16|32}, CL_SNORM_INT{8|16}
CL_INTENSITY: CL_HALF_FLOAT, CL_FLOAT, CL_UNORM_INT{8|16}, CL_SNORM_INT{8|16}
CL_LUMINANCE: CL_UNORM_INT{8|16}, CL_HALF_FLOAT, CL_FLOAT, CL_SNORM_INT{8|16}
CL_RG, CL_RA: CL_HALF_FLOAT, CL_FLOAT, CL_UNORM_INT{8|16}, CL_SIGNED_INT{8|16|32}, CL_UNSIGNED_INT{8|16|32}, CL_SNORM_INT{8|16}
CL_RGB: CL_UNORM_SHORT_{555|565}, CL_UNORM_INT_101010
CL_ARGB: CL_UNORM_INT8, CL_SIGNED_INT8, CL_UNSIGNED_INT8, CL_SNORM_INT8
CL_BGRA: CL_SIGNED_INT8, CL_UNSIGNED_INT8, CL_SNORM_INT8

Lesen und Schreiben von Bildern

Image Read and Write Built-in Functions [6.11.8, 9.8]

The built-in functions defined in this section can only be used with image memory objects created with **clCreateImage2D** or **clCreateImage3D**. OPT = Optional function.

float4 **read_imagef** (image2d_t *image*, sampler_t *sampler*, int2 *coord*) float4 **read_imagef** (image2d_t *image*, sampler_t *sampler*, float2 *coord*) int4 **read_imagei** (image2d_t *image*, sampler_t *sampler*, int2 *coord*) int4 **read_imagei** (image2d_t *image*, sampler_t *sampler*, float2 *coord*) unsigned int4 **read_imageui** (image2d_t *image*, sampler_t *sampler*, int2 *coord*) unsigned int4 **read_imageui** (image2d_t *image*, sampler_t *sampler*, float2 *coord*)	Read an element from a 2D image. *sampler* specifies the addressing and filtering mode to use.
half4 **read_imageh** (image2d_t *image*, sampler_t *sampler*, int2 *coord*) OPT half4 **read_imageh** (image2d_t *image*, sampler_t *sampler*, float2 *coord*) OPT	
void **write_imagef** (image2d_t *image*, int2 *coord*, float4 *color*) void **write_imagei** (image2d_t *image*, int2 *coord*, int4 *color*) void **write_imageui** (image2d_t *image*, int2 *coord*, unsigned int4 *color*)	Write *color* value to (*x*, *y*) location specified by *coord* in the 2D image
void **write_imageh** (image2d_t *image*, int2 *coord*, half4 *color*) OPT	
float4 **read_imagef** (image3d_t *image*, sampler_t *sampler*, int4 *coord*) float4 **read_imagef** (image3d_t *image*, sampler_t *sampler*, float4 *coord*) int4 **read_imagei** (image3d_t *image*, sampler_t *sampler*, int4 *coord*) int4 **read_imagei** (image3d_t *image*, sampler_t *sampler*, float4 *coord*) unsigned int4 **read_imageui** (image3d_t *image*, sampler_t *sampler*, int4 *coord*) unsigned int4 **read_imageui** (image3d_t *image*, sampler_t *sampler*, float4 *coord*)	Read an element from a 3D image. *sampler* specifies the addressing and filtering mode to use.
half4 **read_imageh** (image3d_t *image*, sampler_t *sampler*, int4 *coord*) OPT half4 **read_imageh** (image3d_t *image*, sampler_t *sampler*, float4 *coord*) OPT	
int **get_image_width** (image2d_t *image*) int **get_image_width** (image3d_t *image*)	2D or 3D image width in pixels
int **get_image_height** (image2d_t *image*) int **get_image_height** (image3d_t *image*)	2D or 3D image height in pixels
int **get_image_depth** (image3d_t *image*)	3D image depth in pixels
int **get_image_channel_data_type** (image2d_t *image*) int **get_image_channel_data_type** (image3d_t *image*)	image channel data type
int **get_image_channel_order** (image2d_t *image*) int **get_image_channel_order** (image3d_t *image*)	image channel order
int2 **get_image_dim** (image2d_t *image*)	2D image width and height
int4 **get_image_dim** (image3d_t *image*)	3D image width, height, and depth
void **write_imageh** (image3d_t *image*, int4 *coord*, half4 *color*) OPT	Writes *color* value to (*x*, *y*, *z*) location specified by *coord* in the 3D image.
void **write_imagef** (image3d_t *image*, int4 *coord*, float4 *color*) OPT void **write_imagei** (image3d_t *image*, int4 *coord*, int4 *color*) OPT void **write_imageui** (image3d_t *image*, int4 *coord*, unsigned int4 *color*) OPT	Writes *color* at *coord* in the 3D image. Include this pragma to enable these functions: #pragma OPENCL EXTENSION cl_khr_3d_image_writes : enable

Schnittstelle zu OpenGL

OpenCL/OpenGL Sharing APIs [Appendix B]

Creating OpenCL memory objects from OpenGL objects using the functions **clCreateFromGLBuffer**, **clCreateFromGLTexture2D**, **clCreateFromGLTexture3D**, or **clCreateFromGLRenderbuffer** ensures that the underlying storage of that OpenGL object will not be deleted while the corresponding OpenCL memory object still exists. (Items shown in red are optional)

CL Buffer Objects > GL Buffer Objects [B.1.1]

cl_mem **clCreateFromGLBuffer** (cl_context *context*,
cl_mem_flags *flags*, GLuint *bufobj*,
int *errcode_ret*)

flags: CL_MEM_{READ_ONLY, CL_MEM_WRITE_ONLY, CL_MEM_READ_WRITE

CL Image Objects > GL Textures [B.1.2]

cl_mem **clCreateFromGLTexture2D** (
cl_context *context*, cl_mem_flags *flags*,
GLenum *target*, GLint *miplevel*, GLuint *texture*,
int *errcode_ret*)

flags: (Same as for clCreateFromGLBuffer)

target: GL_TEXTURE_2D, GL_TEXTURE_RECTANGLE, GL_TEXTURE_CUBE_MAP_POSITIVE_{X | Y | Z}, GL_TEXTURE_CUBE_MAP_NEGATIVE_{X | Y | Z}

cl_mem **clCreateFromGLTexture3D** (
cl_context *context*, cl_mem_flags *flags*,
GLenum *target*, GLint *miplevel*, GLuint *texture*,
int *errcode_ret*)

flags: (Same as for clCreateFromGLBuffer)

target: GL_TEXTURE_3D

CL Image Objects > GL Renderbuffers [B.1.3]

cl_mem **clCreateFromGLRenderbuffer** (
cl_context *context*, cl_mem_flags *flags*,
GLuint *renderbuffer*, int *errcode_ret*)

flags: (Same as for clCreateFromGLBuffer)

Query Information [B.1.4]

cl_int **clGetGLObjectInfo** (cl_mem *memobj*,
cl_gl_object_type *gl_object_type*,
GLuint *gl_object_name*)

gl_object_type: CL_GL_OBJECT_BUFFER, CL_GL_OBJECT_TEXTURE2D, CL_GL_OBJECT_TEXTURE_RECTANGLE, CL_GL_OBJECT_TEXTURE3D, CL_GL_OBJECT_RENDERBUFFER

cl_int **clGetGLTextureInfo** (cl_mem *memobj*,
cl_gl_texture_info *param_name*,
size_t *param_value_size*, void *param_value*,
size_t *param_value_size_ret*)

param_name: CL_GL_TEXTURE_TARGET, CL_GL_MIPMAP_LEVEL

Share Objects [B.1.5]

cl_int **clEnqueueAcquireGLObjects** (
cl_command_queue *command_queue*,
cl_uint *num_objects*, const cl_mem *mem_objects*,
cl_uint *num_events_in_wait_list*,
const cl_event *event_wait_list*, cl_event *event*)

cl_int **clEnqueueReleaseGLObjects** (
cl_command_queue *command_queue*,
cl_uint *num_objects*,
const cl_mem *mem_objects*,
cl_uint *num_events_in_wait_list*,
const cl_event *event_wait_list*, cl_event *event*)

Querying CL Devices in GL Context [9.11]

cl_int **clGetGLContextInfoKHR** (
const cl_context_properties *properties*,
cl_gl_context_info *param_name*,
size_t *param_value_size*, void *param_value*,
size_t *param_value_size_ret*)

param_name: CL_DEVICES_FOR_GL_CONTEXT_KHR, CL_CURRENT_DEVICE_FOR_GL_CONTEXT_KHR

OpenCL Empfehlungen

- Gute Parallelisierung eines sequentiellen Algorithmus (**Datenparallelität** mit ‚**leichtem**‘ **Kernel** ist optimal)
- **Datentransfer** zwischen Hauptcomputer und Gerät **minimieren**
- Verwendung von globalem Speicher minimieren. **Privaten** und **lokalen Speicher bevorzugen**
- **Verschiedene Ausführungspfade** im Kernel **vermeiden**
- Verwenden der **optimierten OpenCL Mathe-Bibliothek** (evtl. auch mit reduzierter Genauigkeit)

(**Optimierung**: NVIDIA OpenCL Best Practices Guide 1.0, PDF)