Bauhaus-Universität Weimar

Parallele und Verteilte Systeme

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

Vorlesung WS 2017/18

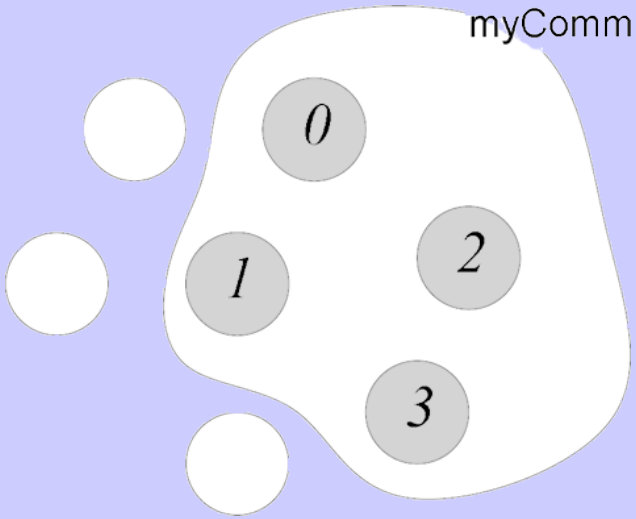
(Unterlagen nur für den internen Gebrauch!)

Übersicht

- **Verteilte Systeme mit Open MPI**
  - Kommunikatoren und Gruppen
- **Massiv parallele Programmierung**
  - Einführung in OpenCL

Kommunikatoren und Gruppen

- **Unterscheidung** verschiedener Kontexte
- Konfliktfreie Organisation von **Gruppen**
- Kommunikatoren können nicht explizit erzeugt werden. Vielmehr kann ein Kommunikator nur aus ein einem **bestehenden** Kommunikator oder einer bestehenden Gruppe **abgeleitet** werden
- **Vordefinierte** Kommunikatoren
  - `MPI_COMM_WORLD`
  - `MPI_COMM_NULL`
  - `MPI_COMM_SELF`



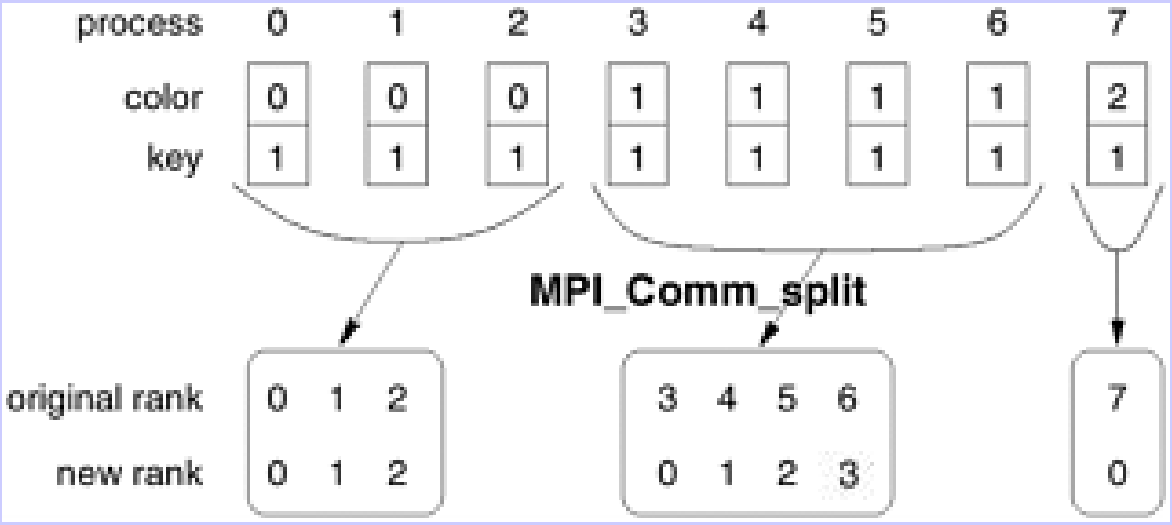
Kommunikatoren duplizieren

- `int MPI_Comm_dup(MPI_Comm cOld, MPI_Comm *cNew);`
- Erzeugt eine **Kopie** `cNew` vom Kommunikator `cOld`
- Erlaubt z.B. eindeutige Abgrenzung/Charakterisierung von Nachrichten
- **Beispiel**

```
MPI_COMM myworld;
...
MPI_Comm_dup(MPI_COMM_WORLD, &myworld)
```

Kommunikatoren teilen

- `int MPI_Comm_split(MPI_Comm cOld, int color, int key, MPI_Comm *cNew);`
- Unterteilt Kommunikator **`comm`** in mehrere Kommunikatoren mit disjunkten Prozessgruppen
- Prozesse mit gleicher Farbe **`color`** bilden gemeinsamen neuen Kommunikator
- **`key`** steuert die Zuordnung der Ränge
- **`MPI_Comm_split`** muss von **allen** Prozessen in **`comm`** aufgerufen werden



Beispiel: Kommunikator teilen

```
#include <mpi.h>

int main (int argc, char *argv[])
{
    int col, key, rank, size;
    MPI_Comm comm0, comm1, comm2;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    col = rank % 3;                              // Color = 0,1,2
    key = size–rank-1;                          // Key = size-1,…,0
    if (col == 0) {
        MPI_Comm_split(MPI_COMM_WORLD, MPI_UNDEFINED, 0, &comm0);
    } else if (col == 1) {
        MPI_Comm_split(MPI_COMM_WORLD, col, key, &comm1);
    } else { // col == 2
        MPI_Comm_split(MPI_COMM_WORLD, col, key, &comm2);
    }
    MPI_Finalize();
}
```

Ergebnis der Aufteilung np = 9

MPI_COMM_WORLD

Rang	P0	P1	P2	P3	P4	P5	P6	P7	P8
color	$\perp$	1	2	$\perp$	1	2	$\perp$	1	2
key	0	7	6	0	4	3	0	1	0

MPI_COMM_WORLD

comm1

P1	P4	P7
2	1	0

comm2

P2	P5	P8
2	1	0

P0	P3	P6
0	1	2

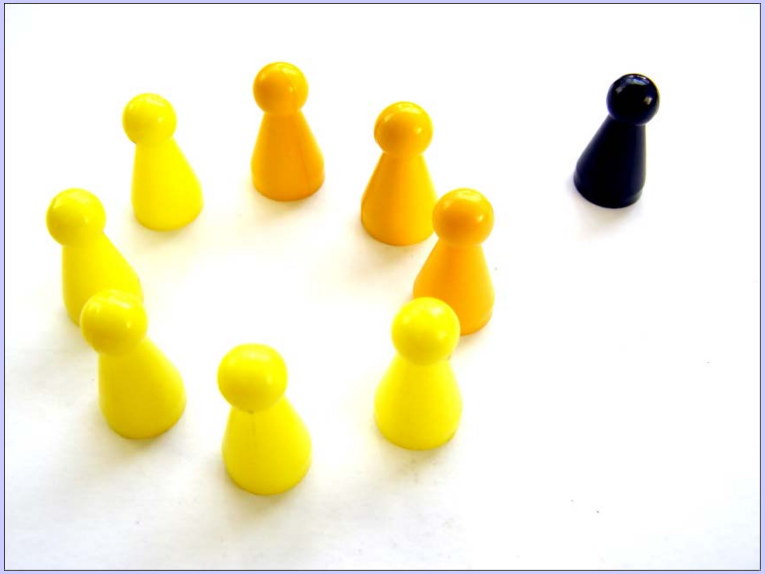
Kommunikatoren auflösen

- `int MPI_Comm_free(MPI_Comm *comm);`
- Löschen des Kommunikators **`comm`**
- Die von **`comm`** belegten Ressource werden von MPI freigegeben
- Kommunikator hat nach dem Aufruf den Wert des Null-Handles

  `MPI_COMM_NULL`
- Funktion muss von **allen** Prozessen aus **`comm`** aufgerufen werden

Prozessgruppen

- Zu jedem **Kommunikator** lässt sich die **Gruppe** ermitteln
- Aus jeder **Gruppe** kann man einen **Kommunikator** konstruieren
- **Prozessgruppen** bestehen aus einer Menge **durchgehend** und **eindeutig** nummerierter Prozessidentifikatoren
- Außerdem legt sie fest, welche Prozesse in eine **kollektive Operation** einbezogen sind



Funktionen für Gruppen

- `int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)`
  Zugriff auf die Prozess**gruppe** eines Kommunikators
- `int MPI_Comm_create(MPI_Comm cOld, MPI_Group group, MPI_Comm *cNew)`
  Erzeugen eines **Kommunikators** aus einer Gruppe
- `int MPI_Group_incl(MPI_Group gOld, int nranks, int *ranks, MPI_Group *gNew)`
  Hinzufügen von Prozessen in eine Gruppe
- `int MPI_Group_excl(MPI_Group gOld, int nranks, int *ranks, MPI_Group *gNew)`
  Herauslösen von Prozessen aus einer Gruppe
- `int MPI_Group_range_incl(MPI_Group gOld, int nranges, int ranges[][3], MPI_Group *gNew)`
  Bilden einer Gruppe aus einfachen **Mustern** (Anfang, Ende, Abstand)
- `int MPI_Group_range_excl(MPI_Group gOld, int nranges, int ranges[][3], MPI_Group *gNew)`
  Ausschließen von Prozessen mit einfachen Mustern

Beispiel: Gruppe mit geraden PIDs

```
int main(int argc, char *argv[])
{
    MPI_Group group_world, even_group;;
    int i, p, Neven, members[8];

    //...
    MPI_Comm_size(MPI_COMM_WORLD, &p);          // für p <= 16
    MPI_Comm_group(MPI_COMM_WORLD, &group_world);

    Neven = (p+1)/2;
    for (i = 0; i < Neven; i++) {
        members[i] = 2*i;
    }
    MPI_Group_incl(group_world, Neven, members, &even_group);
    //...
}
```

Beispiel: Ausgewählte Bereiche

```
#define  MAX 2

void main(int argc, char *argv[])
{
    MPI_Group  group, newgroup;
    int        ranges[MAX][3];

    // ...
    MPI_Comm_group(MPI_COMM_WORLD, &group);
    ranges[0][0] = 1;  // Erster
    ranges[0][1] = 5;  // Letzter
    ranges[0][2] = 2;  // Schrittweite
    ranges[1][0] = 6;  // Erster
    ranges[1][1] = 7;  // Letzter
    ranges[1][2] = 1;  // Schrittweite
    // Aus mindestens 8 Prozessen werden 1, 3, 5, 6 und 7 ausgewählt
    MPI_Group_range_incl(group, MAX, ranges, &newgroup);
    // ...
}
```

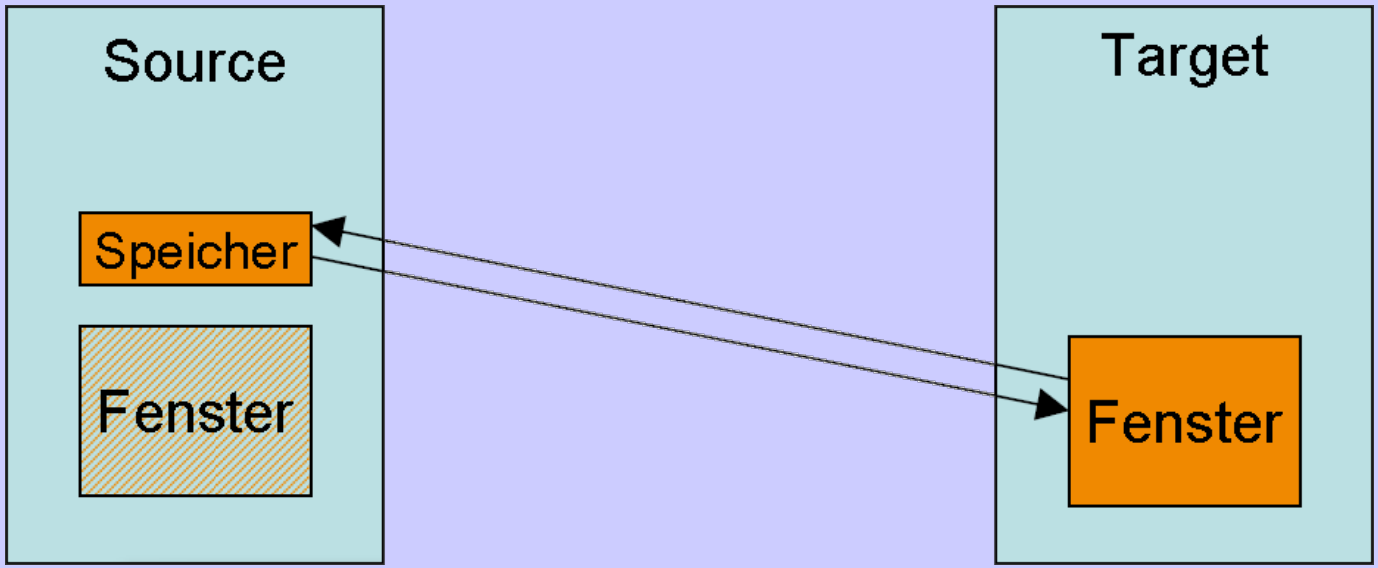
Operationen auf Kommunikatorgruppen

- Darüber hinaus existieren weitere **Funktionen zur Gruppierung**:
- **Zusammenfassen** von Gruppen
```
int MPI_Group_union(MPI_Group g1, MPI_Group g2, MPI_Group *gRes)
```
- **Schnittmenge** von Gruppen
```
int MPI_Group_intersection(MPI_Group g1, MPI_Group g2,
                           MPI_Group *gRes)
```
- **Differenz** von Gruppen
```
int MPI_Group_difference(MPI_Group g1, MPI_Group g2,
                         MPI_Group *gRes)
```
- **Vergleich** von Gruppen (MPI_IDENT, MPI_SIMILAR, MPI_UNEQUAL)
```
int MPI_Group_compare(MPI_Group g1, MPI_Group g2, int *result)
```
- **Auflösen** von Gruppen
```
int MPI_Group_free(MPI_Group *group)
```
- **Größe** einer Gruppe
```
int MPI_Group_size(MPI_Group group, int *size)
```
- **Rang** einer Gruppe
```
int MPI_Group_rank(MPI_Group group, int *rank)
```

Ausblick: Entwicklung MPI

- Neue Version des Standards, erweitert MPI in einigen Bereichen
- Loslösung vom starren **Prozessmodell**
  - **Starten** weiterer Prozesse **zur Laufzeit** möglich.
  - Mit den neu gestarteten Prozessen kann kommuniziert werden.
- Kommunikation mit schon **laufenden MPI-Anwendungen** möglich.
  - Kommunikation über sog. Ports.
  - Finden benannter Ports über eine Registry.
- **Einseitige Kommunikation**: Put, Get, Accumulate, sowie Methoden zur Synchronisation
- Parallele **Datei-**Ein- und Ausgabe**operation**
- Anbindung für C++

Einseitige Kommunikation



- Knoten „Source“ greift mittels `get()` und `put()` auf das Fenster des Knotens „Target“ zu.
- Für Quell- bzw. Zieldaten können im Prinzip beliebige Speicherbereiche verwendet werden.
- Knoten „Target“ stellt sein Fenster zur Verfügung. Evtl. geänderte Daten sind erst nach der Synchronisation sichtbar.

Bauhaus-Universität Weimar

Einführung in OpenCL

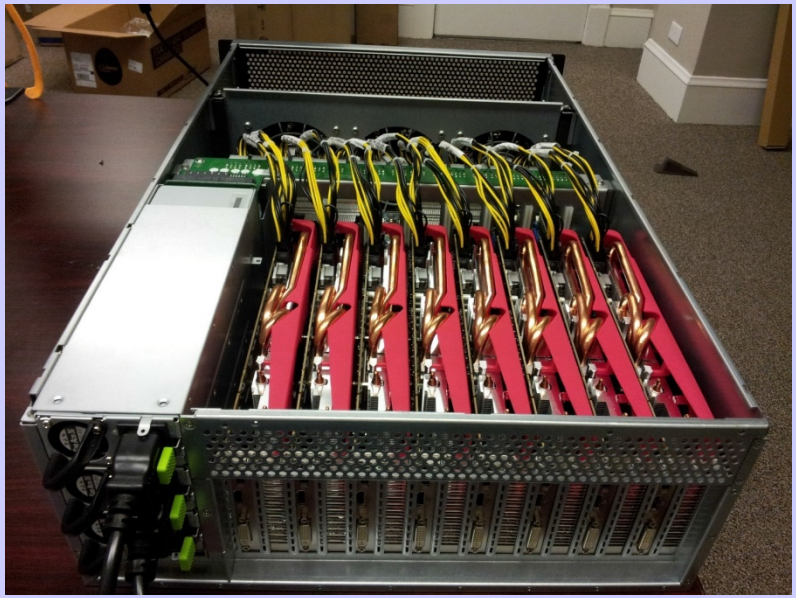
Massive Parallelität mittels Grafikprozessoren (GPUs)

Top500 Supercomputer Juni 2016

Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	National Supercomputing Center in Wuxi China	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway NRCPC	10,649,600	93,014.6	125,435.9	15,371
2	National Super Computer Center in Guangzhou China	**Tianhe-2 (MilkyWay-2)** - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT	3,120,000	33,862.7	54,902.4	17,808
3	DOE/SC/Oak Ridge National Laboratory United States	**Titan** - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560,640	17,590.0	27,112.5	8,209
4	DOE/NNSA/LLNL United States	**Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM	1,572,864	17,173.2	20,132.7	7,890
5	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu	705,024	10,510.0	11,280.4	12,660
6	DOE/SC/Argonne National Laboratory United States	**Mira** - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM	786,432	8,586.6	10,066.3	3,945
7	DOE/NNSA/LANL/SNL United States	**Trinity** - Cray XC40, Xeon E5-2698v3 16C 2.3GHz, Aries interconnect Cray Inc.	301,056	8,100.9	11,078.9	
8	Swiss National Supercomputing Centre (CSCS) Switzerland	**Piz Daint** - Cray XC30, Xeon E5-2670 8C 2.600GHz, Aries interconnect , NVIDIA K20x Cray Inc.	115,984	6,271.0	7,788.9	2,325
9	HLRS - Höchstleistungsrechenzentrum Stuttgart Germany	**Hazel Hen** - Cray XC40, Xeon E5-2680v3 12C 2.5GHz, Aries interconnect Cray Inc.	185,088	5,640.2	7,403.5	
10	King Abdullah University of Science and Technology Saudi Arabia	**Shaheen II** - Cray XC40, Xeon E5-2698v3 16C 2.3GHz, Aries interconnect Cray Inc.	196,608	5,537.0	7,235.2	2,834

	GPU	CPU
Model	Nvidia Tesla K20X	Intel Xeon E5-2670 v3
Architecture	Kepler	Haswell
Launch	Nov-2012	Sep-2014
# of transistors	7.1billion	3.84billion
# of cores	2688 (simple)	12 (functional)
Core clock	732MHz	2.6GHz, up to 3.5GHz
Peak Flops (single precision)	3.95TFLOPS	998.4GFLOPS (with AVX2)
DRAM size	6GB, GDDR5	768GB/socket, DDR4
Memory band	250GB/s	68GB/s
Power consumption	235W	135W
Price	$3,000	$2,094

(GP-)GPU-Cluster



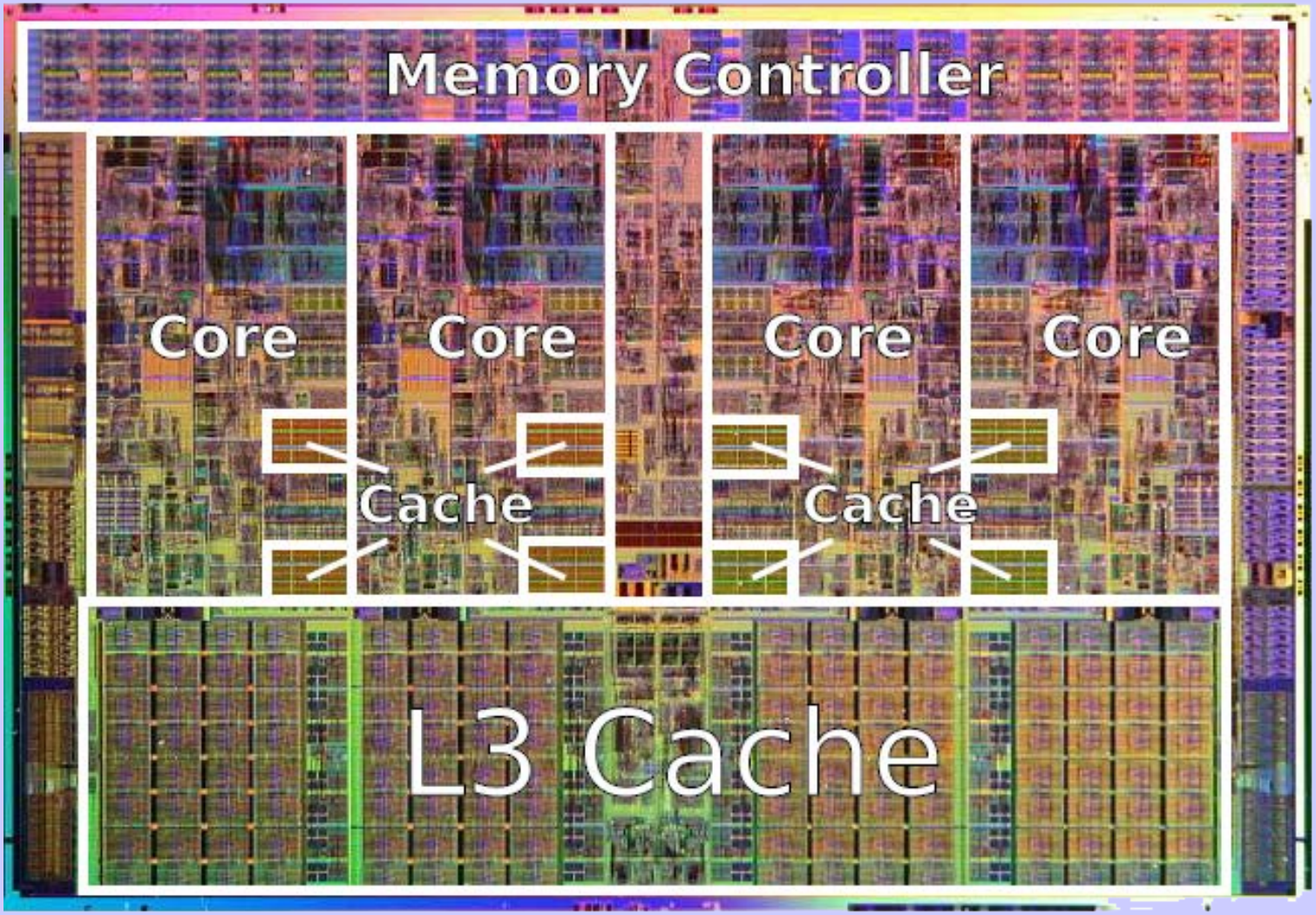
John the Ripper: Rekorde im Passwort-Knacken mit GPUs (25 AMD GPUs in 5 Servern)

Deep Learning mit NVIDIA (Tesla/Geforce)

Open Computing Language (OpenCL)

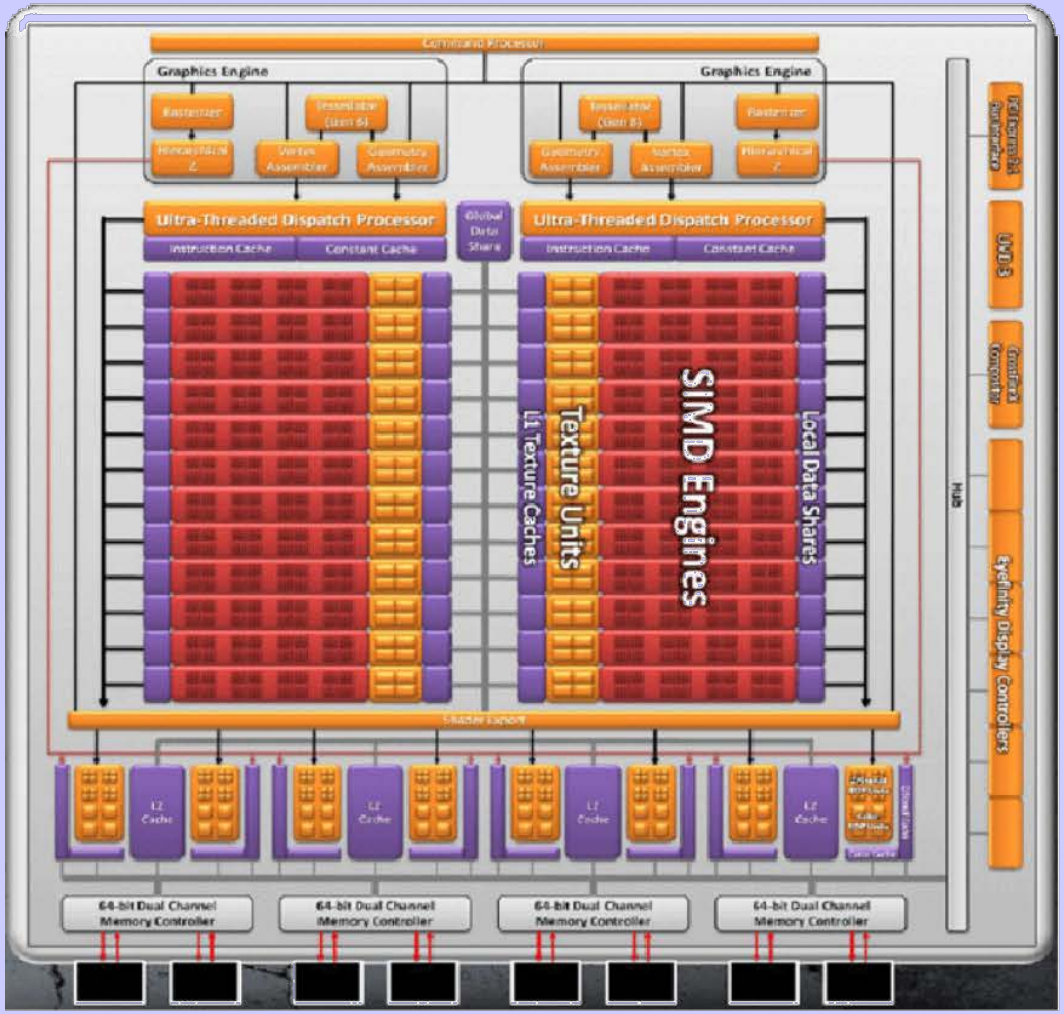
- OpenCL ist ein **offener** Standard für **plattformunabhängiges** Rechnen
- Programmierschnittstelle für Parallelrechner, die mit **Haupt**-, **Grafik**- oder **digitalen Signalprozessoren** ausgestattet sind
- Die zugehörige Programmiersprache ist **OpenCL C**
- Initiiert von Apple in Kooperation mit AMD, ARM, IBM, Intel und Nvidia
- Standardisierung betreut bei der **Khronos Group** (`www.khronos.org`)
- Version 1.0 (2008) bis Version 2.2 (Mai 2017)
- Nützliche Links
  - OpenCL für **Nvidia**:
    `https://developer.nvidia.com/cuda-zone`
    (**CUDA** Toolkit SDK 9.0)
  - OpenCL für **AMD**:
    `http://developer.amd.com/tools-and-sdks/opencl-zone/`
    (Accelerated Parallel Processing (**APP**) SDK 3.0)

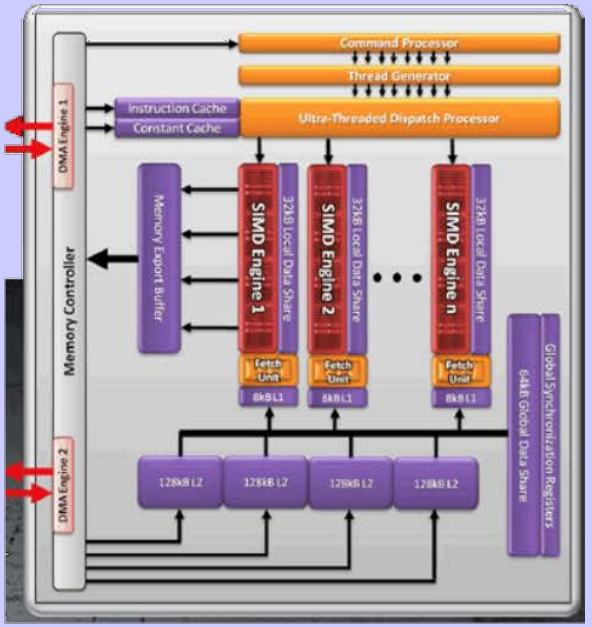
Hauptprozessor Intel Core i7

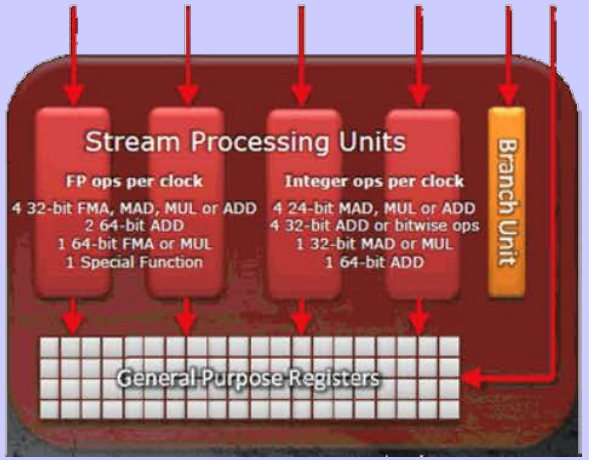


Nvidia GF100

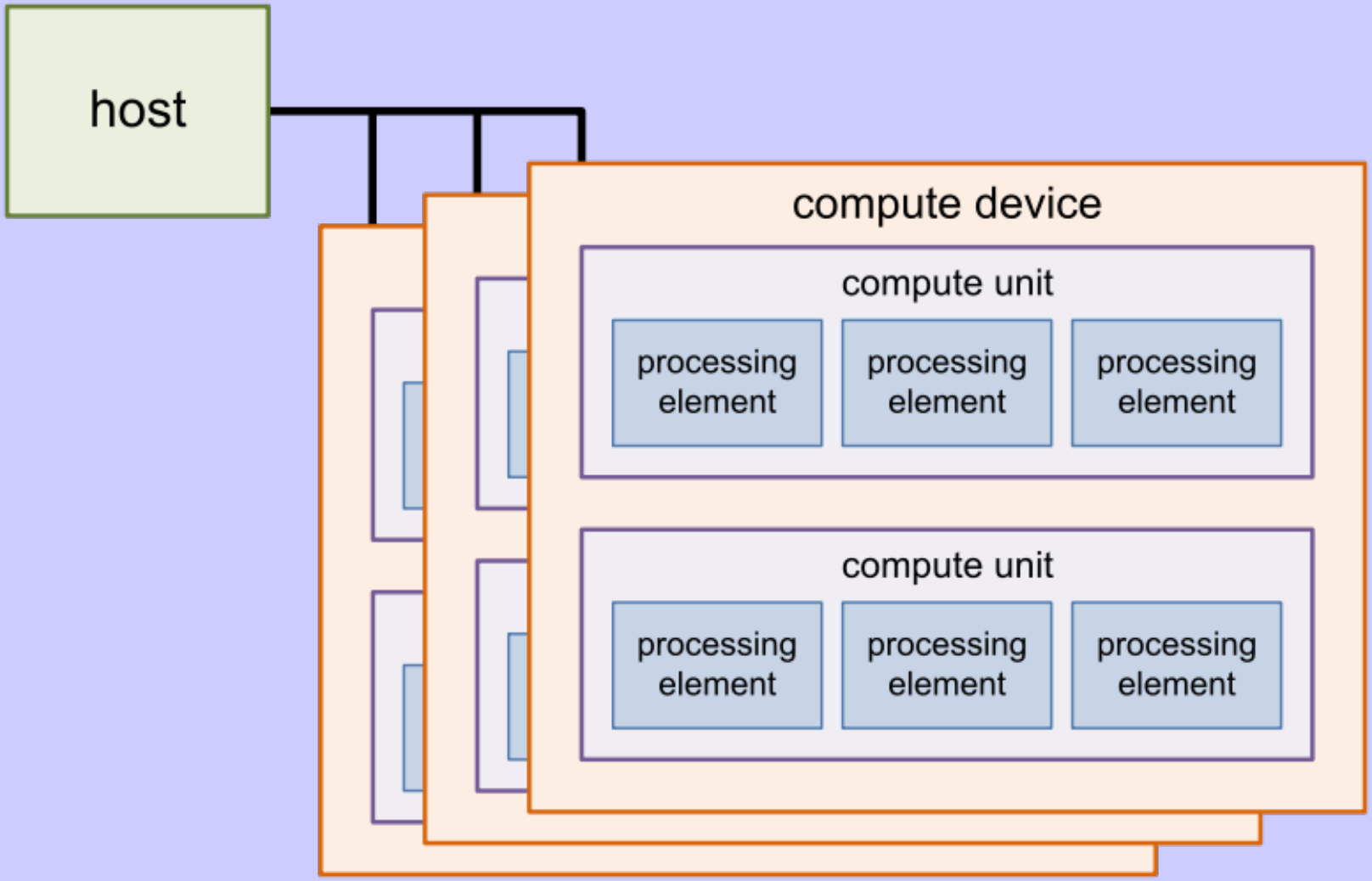
AMD Cayman


Command Processor
Graphics Engine
Graphics Engine
Ultra-Threaded Dispatch Processor
Ultra-Threaded Dispatch Processor
Instruction Cache
Constant Cache
Global Data Share
SIMD Engines
Texture Units
L1 Texture Caches
Local Data Shares
Hub
UVD
Eyefinity Display Controllers
64-bit Dual Channel Memory Controller
64-bit Dual Channel Memory Controller
64-bit Dual Channel Memory Controller
64-bit Dual Channel Memory Controller


Command Processor
Thread Generator
Instruction Cache
Constant Cache
Ultra-Threaded Dispatch Processor
DMA Engine 1
DMA Engine 2
Memory Controller
Memory Export Buffer
SIMD Engine 1
SIMD Engine 2
SIMD Engine n
32kB Local Data Share
Fetch Unit
8kB L1
128kB L2
Global Synchronization Registers
64kB Global Data Share


Stream Processing Units
FP ops per clock
4 32-bit FMA, MAD, MUL or ADD
2 64-bit ADD
1 64-bit FMA or MUL
1 Special Function
Integer ops per clock
4 24-bit MAD, MUL or ADD
4 32-bit ADD or bitwise ops
1 32-bit MAD or MUL
1 64-bit ADD
Branch Unit
General Purpose Registers

Architektur



Beispiel: "Hello World " auf GPUs?

- Grafikkarten erzeugen üblicherweise **keine Textausgabe** auf der Konsole!
- Einfaches **Beispiel-Programm**
  - Quadrieren von Elementen eines Feldes (auf die besonders harte Weise!)
- **Ziel** des Beispiels
  - Demonstrieren der **Initialisierung** von OpenCL
  - Bereitstellen eines einfachen OpenCL **Kerns**
  - Zeigen, dass obwohl **viele Schritte** benötigt werden, es nicht wirklich kompliziert ist (= Gebrauchsanweisung)

Kern „hello“ mit $output_i = input_i^2$

```
#define DATA_SIZE 10
__kernel void hello(__global float *input, __global float *output)
{
    size_t i = get_global_id(0);
    output[i] = input[i] * input[i];
}
```

- **(Rechen)Kern / Kernel**
  - Code, der auf jedem einzelnen Processing Element ausgeführt wird
- **Hinweise**:
  - **`float`** mit **einfacher** Genauigkeit bevorzugt!
  - 2D-Matrizen sollten als **1D-Vektoren** angesprochen werden (z.B. bei dynamischen Matrizen z.B. mit **`A[0]`**)

Spracheigenschaften OpenCL C/C++

- Kernel erlauben im wesentlichen C99, ohne
  - Funktionszeiger
  - Rekursion
  - Arrays variabler Länge
  - Strukturen
- Seit OpenCL 2.2 (2017) auch C++14
  - Klassen
  - Templates
  - Lambda-Ausdrücke
- Optionale Features
  - Gleitkommazahlen mit doppelter Genauigkeit

Aufruf-Beispiel: Hello.cpp

```
#include "CL/cl.h"
int main(void)
{
  // ...
  clGetPlatformIDs(1, &platform_id, &num_of_platforms);
  clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_GPU, 1, &device_id, &num_of_devices);
  clGetDeviceInfo(device_id, CL_DEVICE_NAME, sizeof(name), name, NULL);
  context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
  command_queue = clCreateCommandQueue(context, device_id, 0, &err);
  program = clCreateProgramWithSource(context, 1, (const char **)&KernelSource, NULL, &err);
  clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
  kernel = clCreateKernel(program, "hello", &err);
  input  = clCreateBuffer(context, CL_MEM_READ_ONLY,  MEM_SIZE, NULL, &err);
  output = clCreateBuffer(context, CL_MEM_WRITE_ONLY, MEM_SIZE, NULL, &err);
  clEnqueueWriteBuffer(command_queue, input, CL_TRUE, 0, MEM_SIZE, data, 0, NULL, NULL);
  clSetKernelArg(kernel, 0, sizeof(cl_mem), &input);
  clSetKernelArg(kernel, 1, sizeof(cl_mem), &output);
  clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL, global, NULL, 0, NULL, NULL);
  clFinish(command_queue);
  clEnqueueReadBuffer(command_queue, output, CL_TRUE, 0, MEM_SIZE, results, 0, NULL, NULL);
  // ...
}
```

Aufgaben auf dem Host

1. Abrufen und Auswählen der zu verwendenden **Geräte**
2. Öffnen eines OpenCL-**Kontextes** und Erstellen einer **Befehlswarteschlange**
3. Online-**Kompilierung** des Rechenkerns und Erstellung des **Programms**
4. Erzeugen und Zuweisen von OpenCL-**Speicherobjekten** für Ein- und Ausgaben des Rechenkerns
5. Starten der Rechenkern-**Ausführung** und Sammeln der Ergebnisse

1. Initialisierung der Geräte

```
int main(void)
{
    cl_platform_id platform_id;
    cl_device_id device_id;
    cl_uint num_of_platforms = 0, num_of_devices = 0;
    cl_int err;
    char name[48];

    // Plattform ?
    if (clGetPlatformIDs(1, &platform_id, &num_of_platforms) != CL_SUCCESS) {
        printf("Unable to get platform id\n"); return 1;
    }
    // GPU Device ?
    if (clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_GPU, 1,
                       &device_id, &num_of_devices) != CL_SUCCESS) {
        printf("Unable to get device id\n"); return 1;
    }
    // Device Information ausgeben
    clGetDeviceInfo(device_id, CL_DEVICE_NAME, sizeof(name), name, NULL);
    printf("Device: %s\n", name);
    // ...
```

1.1 Get Platform ID

- Abfragen der Anzahl verfügbarer Plattformen und ihrer ID
- **Beispiel:**

```
clGetPlatformIDs(1, &platform_id, &num_of_platforms);
```

- **Parameter:**

```
cl_int clGetPlatformIDs(
       cl_uint          num_entries,
       cl_platform_id *platforms,
       cl_uint          *num_platforms)
```

- **Rückgabe:**

```
// Error Codes aus cl.h
#define CL_SUCCESS 0
#define CL_DEVICE_NOT_FOUND -1
#define CL_DEVICE_NOT_AVAILABLE -2
...
```

1.2 Get Device ID

- Abfragen der Anzahl verfügbarer Geräte und ihrer ID
- **Beispiel:**

```
clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_GPU, 1,
               &device_id, &num_of_devices);
```

- **Parameter:**

```
cl_int clGetDeviceIDs(
       cl_platform_id platform,
       cl_device_type device_type,
       cl_uint        num_entries,
       cl_device_id  *devices,
       size_uint     *num_devices)
```

- **Optionen:**

```
CL_DEVICE_TYPE_ALL, CL_DEVICE_TYPE_CPU,
CL_DEVICE_TYPE_ACCELERATOR, ...
```

1.3 Get Device Info

- Informationen über Geräte abfragen
- **Beispiel**:

```
clGetDeviceInfo(device_id, CL_DEVICE_NAME, sizeof(name),
                name, NULL);
```

- **Parameter:**

```
cl_int clGetDeviceInfo(
       cl_device_id   device,
       cl_device_info param_name,
       size_t         param_value_size,
       void          *param_value,
       size_t         param_value_size_ret)
```

- **Optionen:**

```
CL_DEVICE_TYPE, CL_DEVICE_VENDOR_ID, CL_DEVICE_MAX_COMPUTE_UNITS,
CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS, CL_DEVICE_MAX_WORK_ITEM_SIZES,
CL_DEVICE_MAX_WORK_GROUP_SIZE, ...
```

Aufgaben auf dem Host

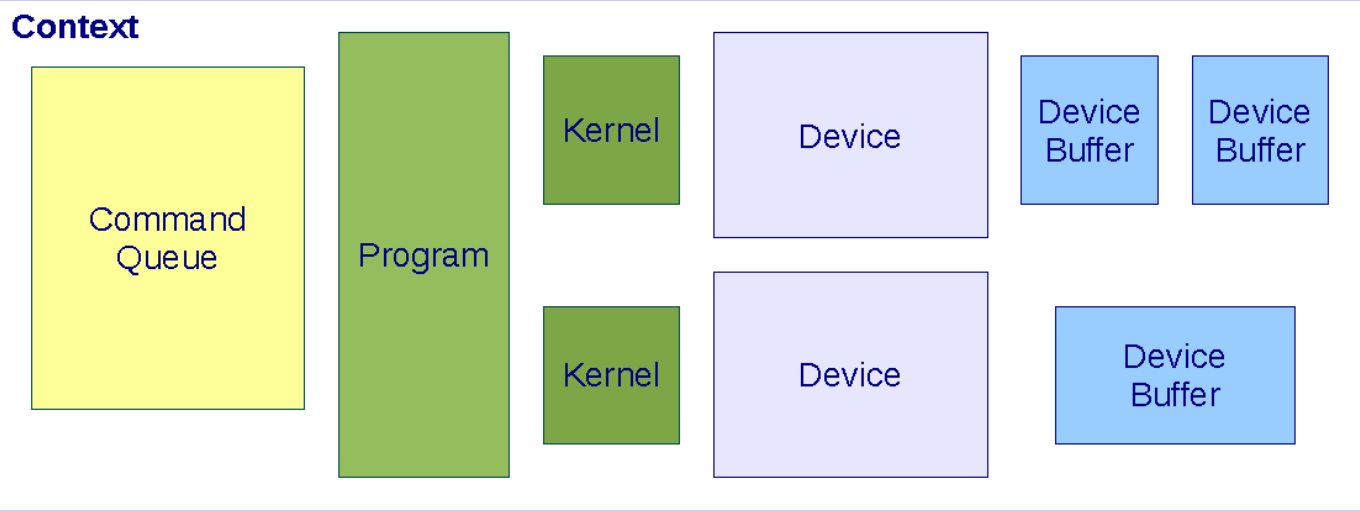
1. Abrufen und Auswählen der zu verwendenden **Geräte**
2. Öffnen eines OpenCL-**Kontextes** und Erstellen einer **Befehlswarteschlange**
3. Online-**Kompilierung** des Rechenkerns und Erstellung des **Programms**
4. Erzeugen und Zuweisen von OpenCL-**Speicherobjekten** für Ein- und Ausgaben des Rechenkerns
5. Starten der Rechenkern-**Ausführung** und Sammeln der Ergebnisse

2. Kontext und Befehlswarteschlange

```
int main(void)
{
    cl_context context;
    cl_command_queue command_queue;

    // ... Kontext oeffnen
    context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);

    // Erzeugen einer Befehlswarteschlange (FIFO)
    command_queue = clCreateCommandQueue(context, device_id, 0, &err);
    // ...
```



2.1 Create Context

- Erzeuge einen OpenCL Kontext
- **Beispiel:**

```
context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
```

- **Parameter:**

```
cl_context clCreateContext(
           cl_context_properties *properties,
           cl_uint                num_devices,
           cl_device_id          *devices,
           void (CL_CALLBACK     *pfn_notify)
                (const char *errinfo, const void
                 *private_info, size_t cb,void *user_data),
           void                  *user_data,
           cl_int                *errcode_ret)
```

- **Optionen:**

```
properties[] = {CL_CONTEXT_PLATFORM,
                (cl_context_properties)platform, 0};
```

2.2 Create Command Queue

- Eine Befehlswarteschlange erzeugen
- **Beispiel:**

```
command_queue = clCreateCommandQueue(context, device_id,
                                     0, &err);
```

- **Parameter:**

```
cl_command_queue clCreateCommandQueue(
                 cl_context                    context,
                 cl_device_id                  device,
                 cl_command_queue_properties properties,
                 cl_int                       *errcode_ret)
```

- **Optionen**:

```
CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE,
CL_QUEUE_PROFILING_ENABLE
```

Aufgaben auf dem Host

1. Abrufen und Auswählen der zu verwendenden **Geräte**
2. Öffnen eines OpenCL-**Kontextes** und Erstellen einer **Befehlswarteschlange**
3. Online-**Kompilierung** des Rechenkerns und Erstellung des **Programms**
4. Erzeugen und Zuweisen von OpenCL-**Speicherobjekten** für Ein- und Ausgaben des Rechenkerns
5. Starten der Rechenkern-**Ausführung** und Sammeln der Ergebnisse

3. Kern kompilieren & Programm erstellen

```
const char *KernelSource =         // Quelltext des Kerns als Zeichenkette
  "#define DATA_SIZE 10                                                  \n" \
  "__kernel void hello(__global float *input, __global float *output)\n" \
  "{                                                                     \n" \
  "    size_t i = get_global_id(0);                                      \n" \
  "    output[i] = input[i] * input[i];                                  \n" \
  "}                                                                     \n" \
  "\n";

int main(void)
{
    cl_program program;
    cl_kernel kernel;

    // ... Erzeuge online ein Programm vom Quellcode des Kerns
    program = clCreateProgramWithSource(context, 1, (const char **)
                                        &KernelSource, NULL, &err);
    if (clBuildProgram(program, 0, NULL, NULL, NULL, NULL) != CL_SUCCESS) {
        printf("Error building program\n"); return 1;
    }
    kernel = clCreateKernel(program, "hello", &err); // Waehle Funktion im Kern
    // ...
```

3.1 Create Program With Source

- **Erstelle ein Programm**
- **Beispiel:**

```
program = clCreateProgramWithSource(context, 1,
          (const char **) &KernelSource, NULL, &err);
```

- **Parameter:**

```
cl_program clCreateProgramWithSource(
           cl_context    context,
           cl_uint       count,
           const char  **strings,
           const size_t *lengths, // (NULL if \0 terminated)
           cl_int       *errcode_ret);
```

- **Alternativen:**

```
clCreateProgramWithBinary()
```

3.2 Build Program

- Kompiliere und Linke den Kernel-Quelltext
- **Beispiel:**
```
clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
```
- **Parameter:**
```
cl_int clBuildProgram(
        cl_program           program,
        cl_uint              num_devices,
        const cl_device_id *device_list,
        const char          *options,
        void (CL_CALLBACK  *pfn_notify)
             (cl_program program, void *user_data),
        void                *user_data)
```

3.3 Create Kernel

- Definiere den Kernel Einsprungspunkt
- **Beispiel:**

```
kernel = clCreateKernel(program, "hello", &err);
```

- **Parameter:**

```
cl_kernel clCreateKernel(
          cl_program  program,
          const char *kernel_name,
          cl_int     *errcode_ret)
```

Aufgaben auf dem Host

1. Abrufen und Auswählen der zu verwendenden **Geräte**
2. Öffnen eines OpenCL-**Kontextes** und Erstellen einer **Befehlswarteschlange**
3. Online-**Kompilierung** des Rechenkerns und Erstellung des **Programms**
4. Erzeugen und Zuweisen von OpenCL-**Speicherobjekten** für Ein- und Ausgaben des Rechenkerns
5. Starten der Rechenkern-**Ausführung** und Sammeln der Ergebnisse

4. Speicher für Ein- und Ausgabe zuweisen

```
#define DATA_SIZE 10                                    // Anzahl der Daten
#define MEM_SIZE  DATA_SIZE*sizeof(float)  // Groesse der Daten im Speicher

int main(void)
{
  cl_mem input, output;
  float data[DATA_SIZE] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

  // ... Erzeuge Puffer für Ein- und Ausgabe
  input  = clCreateBuffer(context, CL_MEM_READ_ONLY,  MEM_SIZE, NULL, &err);
  output = clCreateBuffer(context, CL_MEM_WRITE_ONLY, MEM_SIZE, NULL, &err);

  // Kopiere zusammenhängende Daten aus 'data' in den Eingabe-Puffer 'input'
  clEnqueueWriteBuffer(command_queue, input, CL_TRUE, 0, MEM_SIZE, data, 0, NULL, NULL);

  // Definiere die Reihenfolge der Argumente des Kerns: hello(input, output)
  clSetKernelArg(kernel, 0, sizeof(cl_mem), &input);
  clSetKernelArg(kernel, 1, sizeof(cl_mem), &output);
  // ...
```

4.1 Create Buffer

- Erzeugen eines OpenCL Puffers
- **Beispiel:**

```
input = clCreateBuffer(context, CL_MEM_READ_ONLY, MEM_SIZE,
                       NULL, &err);
```

- **Parameter:**

```
cl_mem clCreateBuffer(
       cl_context   context,
       cl_mem_flags flags,
       size_t       size,
       void        *host_ptr,
       cl_int      *errcode_ret)
```

- **Optionen:**

```
CL_MEM_READ_WRITE, CL_MEM_READ_ONLY, CL_MEM_WRITE_ONLY,
CL_MEM_USE_HOST_PTR, CL_MEM_COPY_HOST_PTR, …
```

4.2 Enqueue Write Buffer

- **Kopieren von Hauptspeicher in einen Puffer**
- **Beispiel:**

```
clEnqueueWriteBuffer(command_queue, input, CL_TRUE, 0,
                     MEM_SIZE, data, 0, NULL, NULL);
```

- **Parameter:**

```
cl_int clEnqueueWriteBuffer(
       cl_command_queue command_queue,
       cl_mem           buffer,
       cl_bool          blocking_write,
       size_t           offset,
       size_t           size,
       const void      *ptr,
       cl_uint          num_events_in_wait_list,
       const cl_event  *event_wait_list,
       cl_event        *event)
```

4.3 Set Kernel Arg

- Definiere Reihenfolge der Kernel Argumente
- **Beispiel**

```
clSetKernelArg(kernel, 0, sizeof(cl_mem), &input);
```

- **Parameter**

```
cl_int clSetKernelArg(
       cl_kernel   kernel,
       cl_uint     arg_index,
       size_t      arg_size,
       const void *arg_value)
```

Aufgaben auf dem Host

1. Abrufen und Auswählen der zu verwendenden **Geräte**
2. Öffnen eines OpenCL-**Kontextes** und Erstellen einer **Befehlswarteschlange**
3. Online-**Kompilierung** des Rechenkerns und Erstellung des **Programms**
4. Erzeugen und Zuweisen von OpenCL-**Speicherobjekten** für Ein- und Ausgaben des Rechenkerns
5. Starten der Rechenkern-**Ausführung** und Sammeln der Ergebnisse

5. Ausführung und Ergebnisse sammeln

```
#define DATA_SIZE 10                                    // Anzahl der Daten
#define MEM_SIZE  DATA_SIZE*sizeof(float)  // Groesse der Daten im Speicher

int main(void)
{
  size_t global[1] = {DATA_SIZE};
  results[DATA_SIZE] = {0};

  // ... Einreihen des Kerns in die Befehlswarteschlange und Aufteilungsbereich angeben
  clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL, global, NULL, 0, NULL, NULL);
  // Auf die Beendigung der Operation warten
  clFinish(command_queue);

  // Kopiere die Ergebnisse vom Ausgabe-Puffer 'output' in das Ergebnisfeld 'results'
  clEnqueueReadBuffer(command_queue, output, CL_TRUE, 0, MEM_SIZE, results, 0, NULL, NULL);
  // ...
```

5.1 Enqueue NDRange Kernel

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

```
clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL,
                       global, NULL, 0, NULL, NULL);
```

- **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)
```

5.2 Enqueue Read Buffer

- Lesen vom Puffer in den Hauptspeicher
- **Beispiel**

```
clEnqueueReadBuffer(command_queue, output, CL_TRUE, 0,
                    MEM_SIZE, results, 0, NULL, NULL);
```

- **Parameter**

```
cl_int clEnqueueReadBuffer(
       cl_command_queue command_queue,
       cl_mem           buffer,
       cl_bool          blocking_read,
       size_t           offset,
       size_t           size,
       const void      *ptr,
       cl_uint          num_events_in_wait_list,
       const cl_event  *event_wait_list,
       cl_event        *event)
```

OpenCL Ressourcen wieder freigeben

```
int main(void)
{
    // ... Aufraeumen der OpenCL Ressourcen
    clReleaseMemObject(input);
    clReleaseMemObject(output);

    clReleaseProgram(program);
    clReleaseKernel(kernel);
    clReleaseCommandQueue(command_queue);
    clReleaseContext(context);

    return 0;
}
```