



**CSCS**

Centro Svizzero di Calcolo Scientifico  
Swiss National Supercomputing Centre

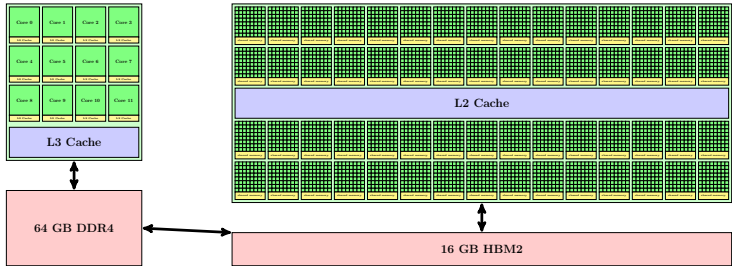
**ETH** zürich



# CUDA runtime API and core libraries

Sebastian Keller, Javier Otero, Prashanth Kanduri  
and Ben Cumming, CSCS

# Hardware and memory on a Piz Daint Node



# Host and Device Memory Spaces

- The GPU has separate memory to the host CPU
  - The host CPU has 64 GB of DDR4 **host memory**
  - The P100 GPU has 16 GB of HBM2 **device memory**
- Kernels executing on the GPU only have fast access to device memory
  - Kernel accesses to host memory are copied to GPU memory first over the (slow) PCIe connection.

<b>host</b> ↔ <b>device</b>	11×2 GB/s	PCIe gen3
<b>host memory</b>	45 GB/s	DDR4
<b>device memory</b>	558 GB/s	HBM2

- **Optimization tip:** The massive bandwidth of HBM2 on P100 GPUs can only help if data is in the right memory space **before** computation starts.

# The CUDA runtime API

- Is a **host** library for orchestrating interactions with the device
  - allocate memory on the device
  - copy data between host and device
  - launch device functions, i.e. kernels
- API functions start with `cuda...`
  - `cudaMalloc`
  - `cudaMemcpy`
  - `<<<...>>>` kernel launch
- Calls are made from **CPU** code

# Allocating Device Memory with `cudaMalloc`

- Can't be read from host
  - host has the pointer to device memory
  - but the host cannot de-reference the pointer
- Need to manually copy data to and from host.
- For memory that should always reside on device.

## Allocating device memory

```
cudaMalloc(void** ptr, size_t size)
```

- `size` number of **bytes** to allocate
- `ptr` points to allocated memory on return

## Freeing device memory

```
cudaFree(void* ptr)
```

## Allocate memory for 100 doubles on device

```
double* v; // C pointer that will point to device memory
auto bytes = 100*sizeof(double); // size in bytes!
cudaMalloc(&v, bytes); // allocate memory
cudaFree(v);           // free memory
```

# Copying Memory with `cudaMemcpy`

- Accepts device pointers obtained with `cudaMalloc`
- Uses the PCI-Express bus to copy between the host and device
- Can also be used for copies within the device

## Perform blocking copy (host waits for copy to finish)

```
cudaMemcpy(void *dst, void *src, size_t size, cudaMemcpyKind kind)
```

- `dst` destination pointer
- `src` source pointer
- `size` number of **bytes** to copy to `dst`
- `kind` enumerated type specifying **direction** of copy:  
one of `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`,  
`cudaMemcpyDeviceToDevice`, `cudaMemcpyHostToHost`

## Copy 100 doubles to device, then back to host

```
auto size = 100*sizeof(double); // size in bytes
double *v_d;
cudaMalloc(&v_d, size);           // allocate on device
double *v_h = (double*)malloc(size); // allocate on host
cudaMemcpy(v_d, v_h, size, cudaMemcpyHostToDevice);
cudaMemcpy(v_h, v_d, size, cudaMemcpyDeviceToHost);
```



## Errors happen...

All API functions return error codes that indicate either:

- success;
- an error in the API call;
- an error in an earlier asynchronous call.

The return value is the enum type `cudaError_t`

- e.g. 

```
cudaError_t status = cudaMalloc(&v, 100);
```

  - status is { `cudaSuccess`, `cudaErrorMemoryAllocation` }

## Handling errors

```
const char* cudaGetErrorString(status)
```

- returns a string describing status

```
cudaError_t cudaGetLastError()
```

- returns the last error
- resets status to `cudaSuccess`

## Copy 100 doubles to device **with error checking**

```
double *v_d;
auto size = sizeof(double)*100;
double *v_host = (double*)malloc(size);
cudaError_t status;

status = cudaMalloc(&v_d, size);
if(status != cudaSuccess) {
    printf("cuda error : %s\n", cudaGetErrorString(status));
    exit(1);
}

status = cudaMemcpy(v_d, v_h, size, cudaMemcpyHostToDevice);
if(status != cudaSuccess) {
    printf("cuda error : %s\n", cudaGetErrorString(status));
    exit(1);
}
```

It is essential to test for errors

But it is tedious and obfuscates our source code if it is done in line for every API and kernel call...

# Exercise: Device Memory API

Open `topics/cuda/practicals/api/util.hpp`

1. what does `cuda_check_status()` do?
2. look at the template wrappers `malloc_host` & `malloc_device`
  - what do they do?
  - what are the benefits over using `cudaMalloc` and `free` directly?
  - do we need corresponding functions for `cudaFree` and `free`?
3. write a wrapper around `cudaMemcpy` for copying data `host→device` & `device→host`
  - remember to check for errors!
4. compile the test and run
  - it will pass with no errors on success

```
> make explicit
> srun ./explicit 8
```

# Exercise: Device Memory API

What does the nvprof profile look like?

```
> strun nvprof -o explicit.nvvp --profile-from-start off -f  
    ./explicit 25  
> nvvp explicit.nvvp &
```

## Note about **nvprof**

For devices newer than the P100, the functionality of **nvprof** is now offered in two new tools:

- **nsight-sys**
- **nsight-compute** .

# Some remarks about cuBLAS

excerpt from the cuBLAS example

```
#include <cublas_v2.h>

cublasHandle_t cublas_handle;
cublasCreate(&cublas_handle);

auto cublas_status =
    cublasDaxpy(cublas_handle, n, &alpha, x_device, 1, y_device, 1)
    ;
```

- Implements BLAS operations for the device
- Compiled library: need an include file and link against `-lcublas`
- Expects device pointers (from `cudaMalloc`)
- Data transfer to/from the device is the user's responsibility
- Launched on the host (device-launched version is a separate library)

# Core libraries: CUB and Thrust

- CUB (Cuda UnBound) and Thrust are header-only
- requires `nvcc` to compile kernel code
- CUB
  - is CUDA specific
  - contains header functions for use in device kernel code
  - contains higher-level operations to launch from host
- Thrust
  - is platform agnostic
  - implements algorithms of the C++ STL
  - CUDA backend built on top of CUB
  - launched from host
- both are built on top of and inter-operable with the CUDA runtime API

# Some Thrust examples

## host and device vectors

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>

thrust::device_vector<double> d_vector;
thrust::host_vector<double> h_vector(10);

// performs cudaMalloc and cudaMemcpy host->device
d_vector = h_vector;

// performs cudaMemcpy device->host
h_vector = d_vector;
```

## sorting

```
#include <thrust/sort.h>

thrust::sort(thrust::device, d_vector.begin(), d_vector.end());
```

## reductions

```
#include <thrust/reduce.h>

thrust::reduce(thrust::device, d_vector.begin(), d_vector.end(), 0)
;
```

# Thrust interoperability with the runtime API

## thrust sort with C-pointers

```
#include <thrust/device_vector.h>
#include <thrust/sort.h>

double* d_v;
cudaMalloc(&d_v, 100*sizeof(double));

thrust::sort(thrust::device,
             thrust::device_pointer_cast(d_v),
             thrust::device_pointer_cast(d_v + 100));
```



# Exercise: Sorting with Thrust

1. How does the performance of `std::sort` on the host compare against `thrust::sort` on the device?
2. What if the data transfer times to and from device are included?