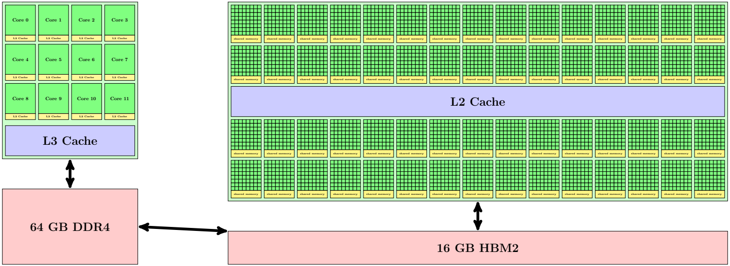
CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Working with GPU memory

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

Memory on a Piz Daint Node



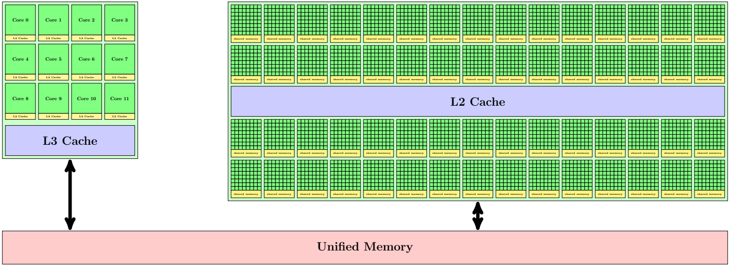
Host and Device Memory Spaces

- 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.

host $\leftrightarrow$ device	11×2 GB/s	PCIe gen3
host memory	45 GB/s	DDR4
device memory	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.

Unified Memory



CUDA unified memory presents a single memory space that can be accessed by both host and GPU code

Unified Memory

Unified memory presents a single memory space.

- Both CPU and GPU can access the same memory.
- First introduced with CUDA 6 and Keplar.
- Improved with CUDA 8 and Pascal:
  - All host and device memory can be addressed
  - The **page migration** engine transfers data between GPU and CPU memory as needed
  - API provides fine-grained control of page migration
- Simplifies memory management for GPU programming

Managed memory is useful for porting to the GPU.

- Not suitable as the default choice of memory management.
- Can lead to negative performance and subtle bugs.
- Host-device memory coherency will improve in future GPUs.

Accessing Memory

CUDA uses C pointers to address memory:

```
double* data = //address to either host, device or managed memory
```

- A pointer can hold an address in
  - **either** device or host memory
  - managed memory that can **migrate** between host and device
- The **CUDA runtime library** provides functions that can be used to allocate, free and copy managed and device memory.

Managing Managed Memory

Allocating managed memory

```
cudaMallocManaged(void** ptr, size_t size, unsigned flags)
```

- `size` number of **bytes** to allocate.
- `ptr` points to allocated memory on return.
- `flags` by default is set to `cudaMemAttachGlobal`.

Freeing managed memory

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

Allocate memory for 100 doubles in managed memory

```
double* v;
auto bytes = 100*sizeof(double);
cudaMallocManaged(&v, bytes); // allocate memory
cudaFree(v);                  // free memory
```

Exercise: Getting started

We have to set up the environment before compiling.

```
> module load daint-gpu
> module swap PrgEnv-cray PrgEnv-gnu
> module load cudatoolkit
> gcc --version # nvcc uses gcc:
gcc (GCC) 8.3.0 20190222 (Cray Inc.)
...
> nvcc --version
...
Cuda compilation tools, release 10.1, V10.1.105
```

Exercise: Managed Memory Example

1. open the files `api/managed.cu` and `api/util.hpp`.
2. what does `managed.cu` do?
   - you can use Google!
3. run it with 20 and 22

```
> cd topics/cuda/practicals/api
> make
> srun ./managed 20
```

4. does it work?
5. run the cuda profiler

```
> srun nvprof -o managed.nvvp --profile-from-start off -f
    ./managed 25
> nvvp managed.nvvp &
```

Concurrent Host-Device Memory Access

The CPU:

- launches the GPU code `gpu_call`
- executes CPU function `cpu_call`

The GPU:

- executes gpu code asynchronously

The problem:

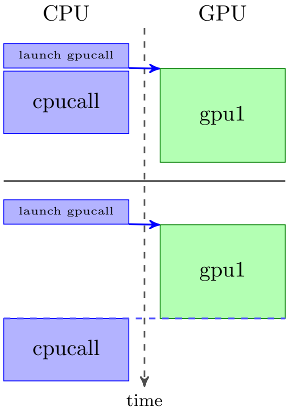
- both `cpu_call` and `gpu_call` may try to access the same memory

The solution:

- synchronize calls between host and device

application code

```
gpu_call<<<...>>>();
cpu_call();
```



Concurrent Host-Device Memory Access

The CPU can't access managed memory at the same time a GPU kernel is accessing it:

- Doing so causes a segmentation faults or undefined behavior.
- To test for synchronization issues run with an environment variable

```
> export CUDA_LAUNCH_BLOCKING=1
```

- The CUDA API function `cudaDeviceSynchronize` can be used to force synchronization.

```
gpu_call<<<...>>>();
cudaDeviceSynchronize();
cpu_call();
```

Exercise: Managed Memory Debugging

1. Test if concurrent host-device memory access caused the incorrect results in the `api/managed.cu` example.
2. Can you fix the issue by adding one `cudaDeviceSynchronize()` call?
3. does the profile from nvprof look different?

Allocating Device Memory

It is possible to allocate host and device memory directly

- Explicitly allocate memory on device.
  - can't be read from host.
- Manually copy data to and from host.
- For memory that should always reside on device.
- The programmer can optimize memory transfers by hand.
  - with effort, you can get the best performance this way.

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
```

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

1. How does performance compare with the managed memory version?
2. What does the nvprof profile look like?
   - contrast with managed memory profile.

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