Introduction to Multi-GPU Computing

Sebastian Kuckuk

Zentrum für Nationales Hochleistungsrechnen Erlangen (NHR@FAU)

Enroll in the Course

- Go to

  courses.nvidia.com/dli-event

- Enter the code

  [event code]

- All your courses are available at

  courses.nvidia.com/dashboard

Enroll in the Course

- Set your Zoom name as [first name] [last name] ([affiliation])

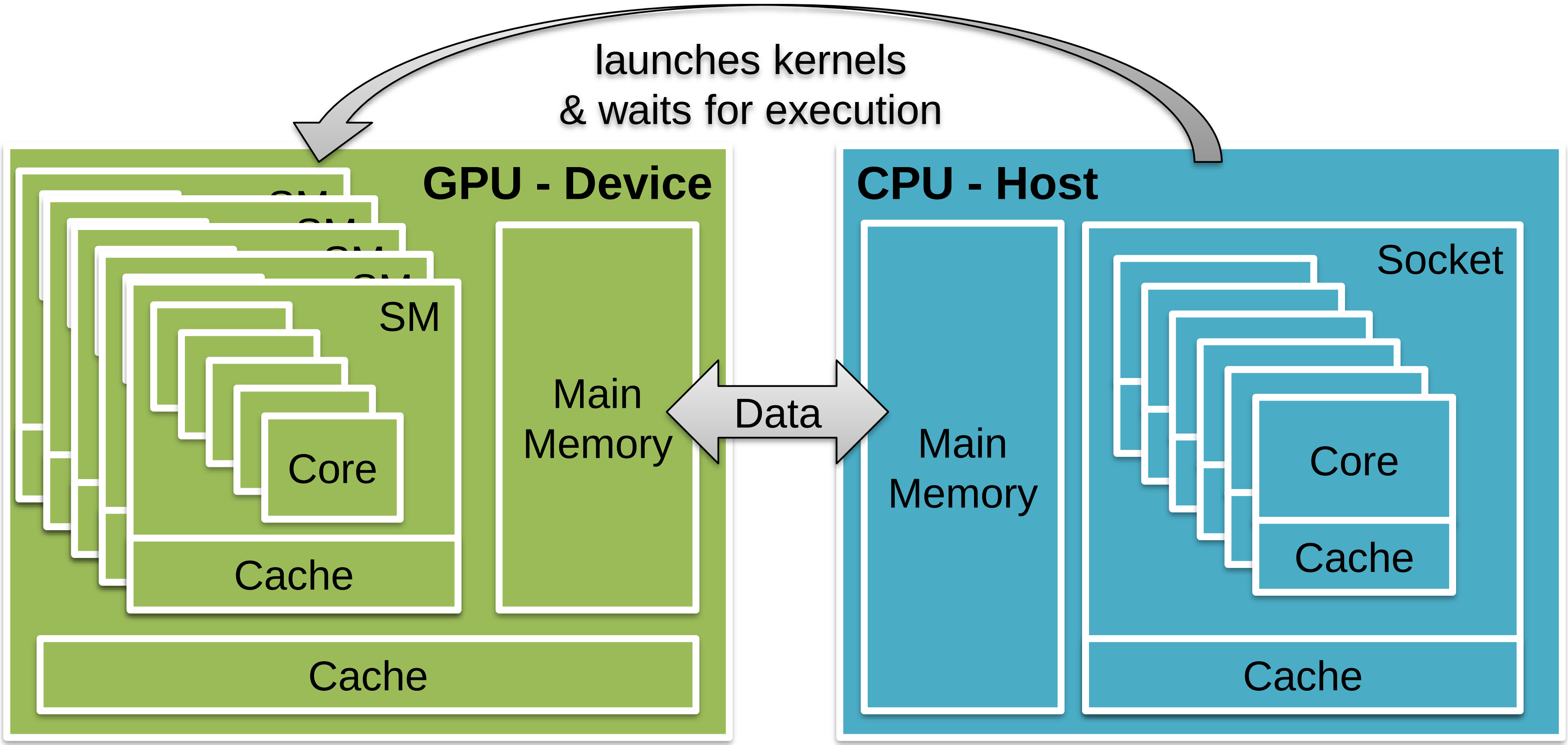
- The DLI part of the course is composed of multiple modules (one IPython notebook each) and augmented with additional material available at https://github.com/SebastianKuckuk/accelerated-programming

- Pass the assessment(s) to get a certificate from NVIDIA

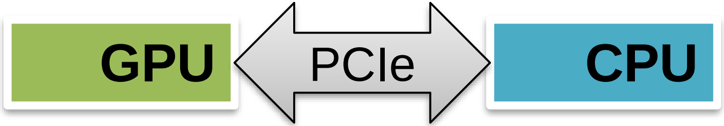
- You will have access to the course material at least six months

- Feel free to interrupt and ask questions

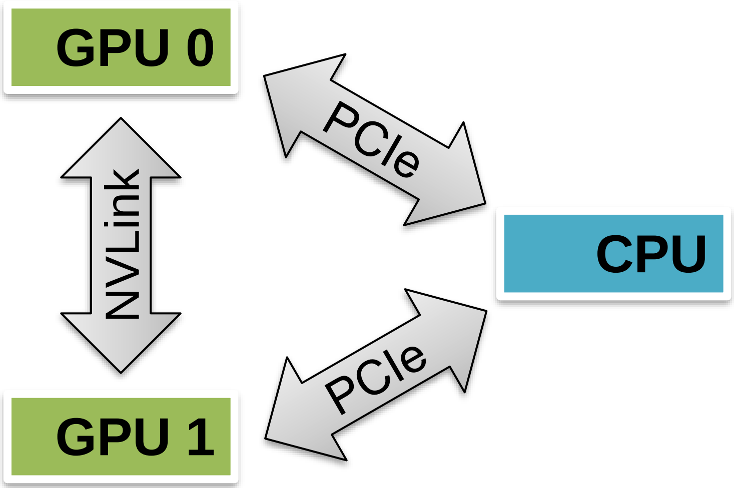
Simplified Architecture



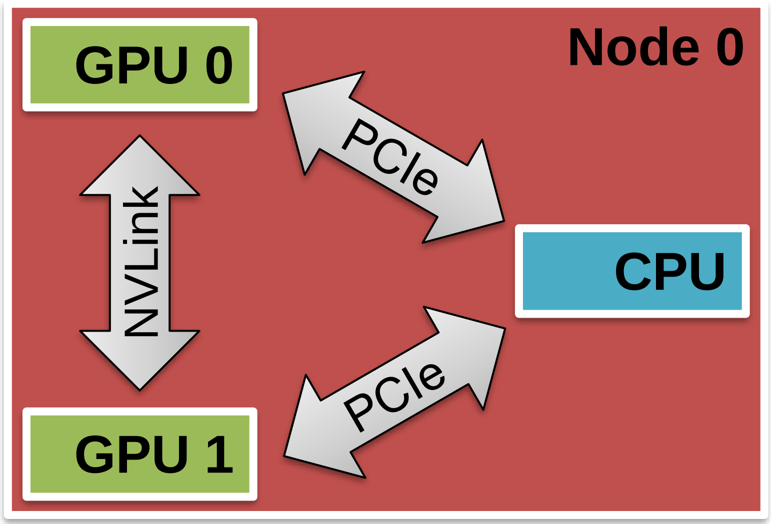
Simplified Architecture



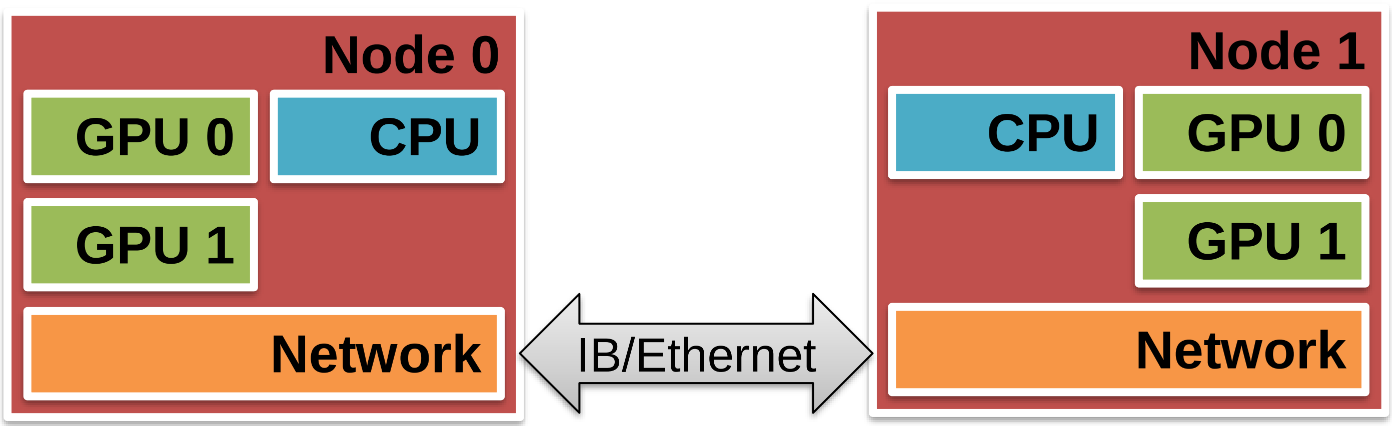
Simplified Architecture



Simplified Architecture



Simplified Architecture



Workflow (single GPU)

1. Allocate data
2. Initialize data on CPU

3. Copy data from CPU to GPU
4. Launch GPU kernels
5. Do independent work on CPU
6. Synchronize GPU
7. Copy data from GPU to CPU

Repeat as required

8. Post-process data on CPU
9. De-Allocate data

Workflow Example

- Goal: repetition of basic CUDA C++ programming elements

- Sample application: copy array and increase each element by 1

- Full code available at
  - https://github.com/SebastianKuckuk/accelerated-programming

Workflow

- 1. Allocate Data (managed)

```
int main(int argc, char *argv[]) {
    size_t nx   = atoi(argv[1]);
    size_t size = sizeof(double) * nx;

    double *src, *dest;
    cudaMallocManaged(&src,  size);
    cudaMallocManaged(&dest, size);




    // ...
}
```

(explicit)

```
double *src, *dest;
cudaMallocHost(&src,  size);
cudaMallocHost(&dest, size);

double *d_src, *d_dest;
cudaMalloc(&d_src,  size);
cudaMalloc(&d_dest, size);
```

Workflow

- 2. Initialize data on CPU

```
void initOnCPU(double *src, size_t nx) {
    for (size_t i = 0; i < nx; ++i)
        src[i] = 1337.;
}

int main(/* ... */) {
    // allocate

    initOnCPU(src, nx);

    // ...
}
```

Workflow

- 3. Copy data from CPU to GPU

```
// allocate & init

cudaMemPrefetchAsync(src,  size, 0);

cudaMemPrefetchAsync(dest, size, 0);

// ...
```

```
cudaMemcpy(d_src, src, size,
           cudaMemcpyHostToDevice);
```

In practice: use the actual device id instead of `0`

Workflow

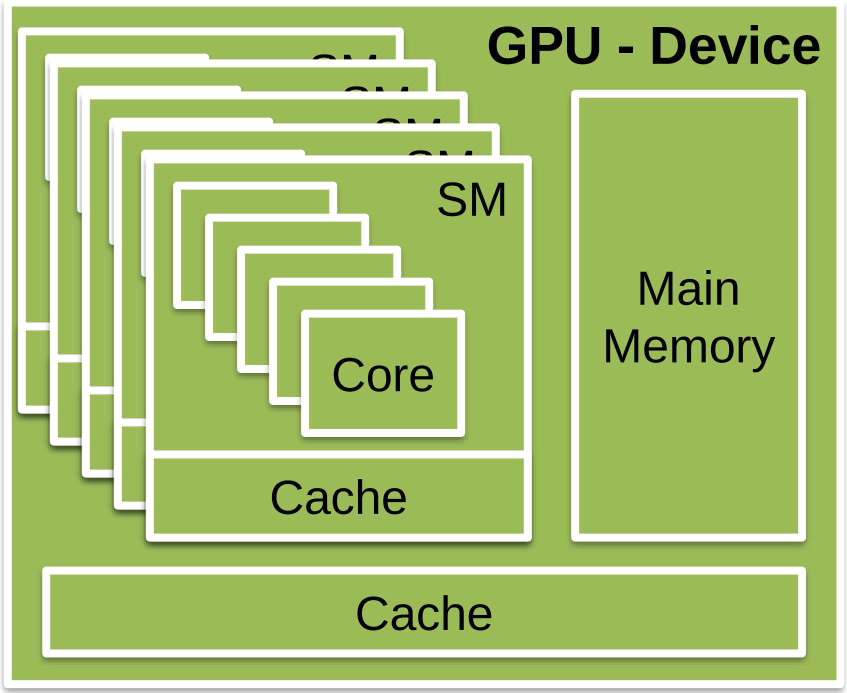
- 4. Launch GPU kernels

```
__global__ void copyOnGPU(double *src, double *dest, size_t nx) {
    size_t i = blockIdx.x * blockDim.x + threadIdx.x;


    if (i < nx)
        dest[i] = src[i] + 1;
}

// ... in main
copyOnGPU<<<(nx + 255) / 256, 256>>>(  src,   dest, nx);
// ... for managed, or for explicit
copyOnGPU<<<(nx + 255) / 256, 256>>>(d_src, d_dest, nx);
```

CUDA Mapping



- **Grids** are mapped to **devices**
- **Blocks** are mapped to **SMs**
- **Threads** are mapped to **cores**
- **Threads of a block** are executed in **warps** (groups of 32 threads)

Workflow

- 4. Launch GPU kernels (grid-stride loop)

```
__global__ void copyOnGPU(double *src, double *dest, size_t nx) {
    size_t start  = blockIdx.x * blockDim.x + threadIdx.x;
    size_t stride = gridDim.x * blockDim.x;

    for (size_t i = start; i < nx; i += stride)
        dest[i] = src[i] + 1;
}

// ... in main
copyOnGPU<<<1280, 256>>>(  src,   dest, nx);
// for managed or for explicit
copyOnGPU<<<1280, 256>>>(d_src, d_dest, nx);
```

Workflow

- 6. Synchronize GPU

```
cudaDeviceSynchronize();
```

Workflow

- 7. Copy data from GPU to CPU

```
// computation

cudaMemPrefetchAsync(dest, size,
                     cudaCpuDeviceId);

// ...
```

```
cudaMemcpy(dest, d_dest, size,
           cudaMemcpyDeviceToHost);
```

Workflow

- 8. Post-process data on CPU

```
void checkOnCPU(double *dest, size_t nx) {
    for (size_t i = 0; i < nx; ++i)
        assert(1338. == dest[i]);
}
```

Workflow

- 9. De-Allocate Data

```
// post-processing




cudaFree(src);
cudaFree(dest);
```

```
cudaFree(d_src);
cudaFree(d_dest);

cudaFreeHost(src);
cudaFreeHost(dest);
```