Introduction to GPU Computing

Sebastian Kuckuk

Zentrum für Nationales Hochleistungsrechnen Erlangen (NHR@FAU)

Enroll in the Course

- Go to

learn.nvidia.com/dli-event

- Enter the code

[event code]

- All your courses are available at

learn.nvidia.com/my-learning

Enroll in the Course

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

- The course is composed of three modules (one IPython notebook each)
  - Slow start, more in depth as we go along
  - Advanced material at the end of most notebooks

- 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

Motivation

- Massive parallelism and performance
- Good performance in relation to energy (FLOP/s per Watt)
- More and more compute clusters are becoming heterogeneous
- 9 out of the top 10 supercomputers feature GPUs
  - 6 NVIDIA
  - 2 AMD
  - 1 Intel
  - c.f. https://www.top500.org/lists/top500/2024/06/

CPU-GPU Comparison

CPU

- Cores
  - Few but powerful
- Memory
  - Large capacity
  - Latency optimized

➢ Ideal for irregular workload

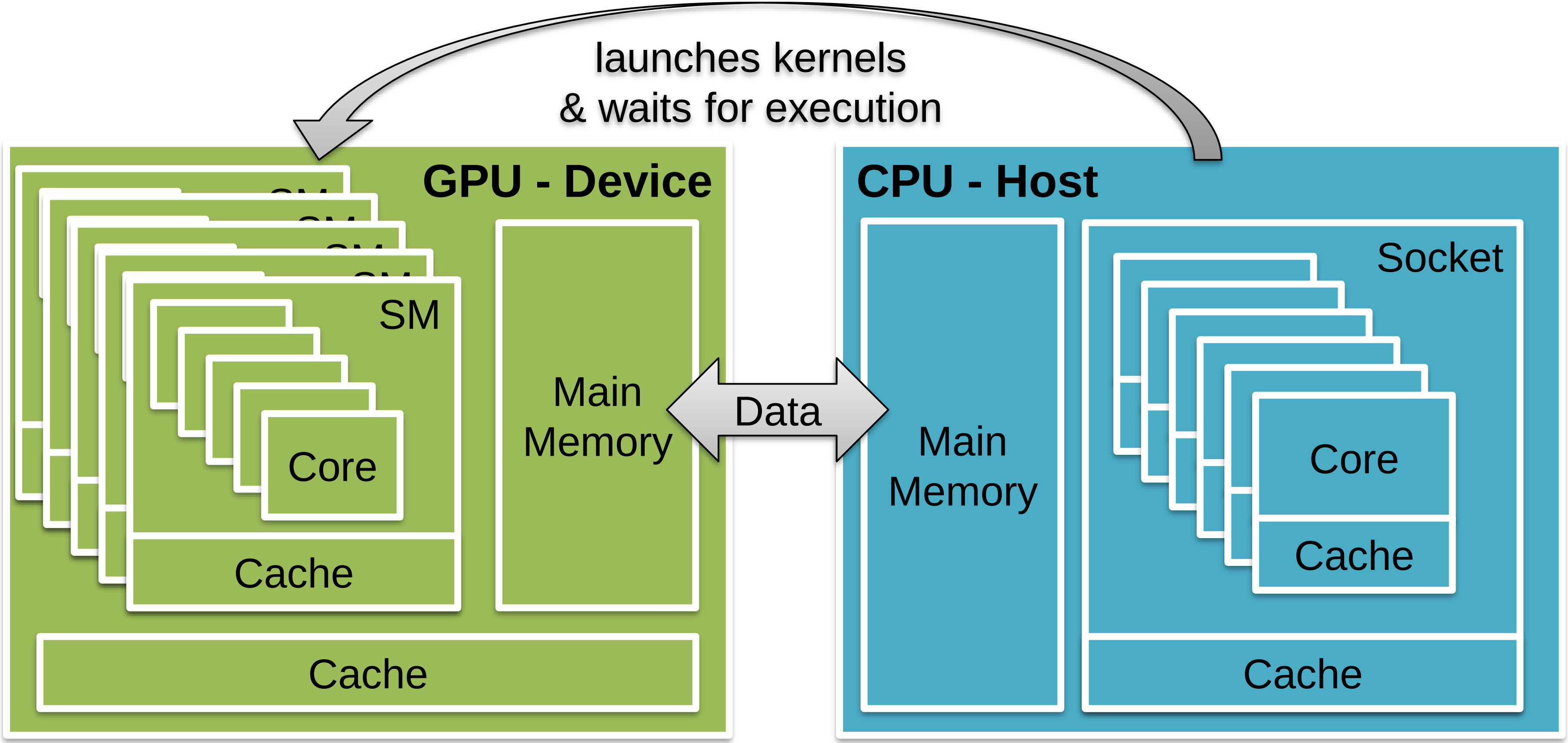
➢ A small number of fast threads

GPU

- Cores
  - Many but less powerful
- Memory
  - Small capacity
  - Bandwidth optimized

➢ Ideal for massively parallel structured computations

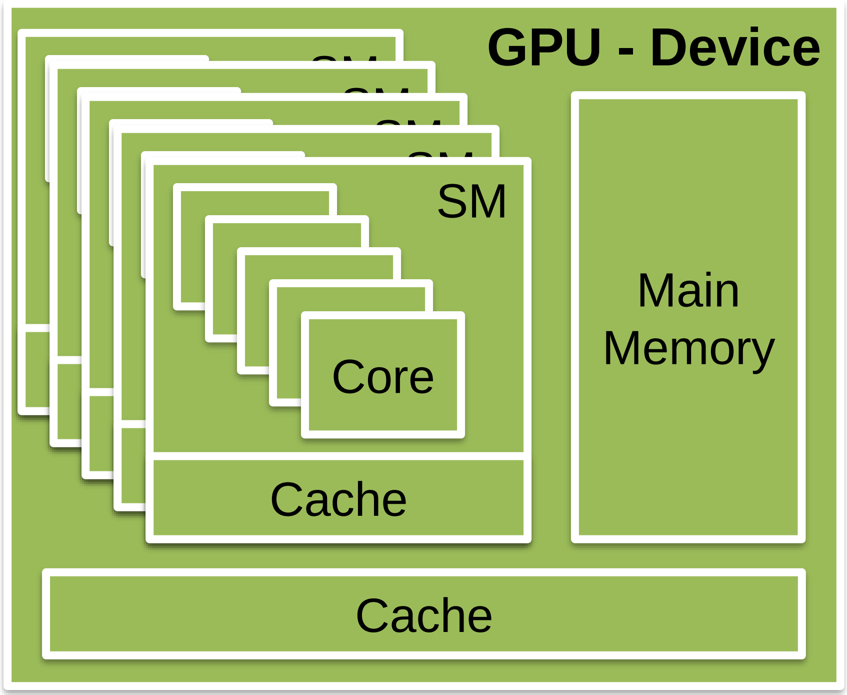
Simplified Architecture



Workflow

1. Initialize data on CPU
2. Copy data from CPU to GPU
3. Launch GPU kernels
4. Do independent work on CPU (optional)
5. Synchronize GPU
6. Copy data from GPU to CPU
7. Post-process data on CPU

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)

Programming models

- Dedicated programming languages
  - NVIDIA CUDA
  - AMD HIP
  - SYCL
  - …

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

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


int main(int argc, char *argv[]) {
  double *src, *dest;
  cudaMallocManaged((void **) &src,  sizeof(double) * nx);
  cudaMallocManaged((void **) &dest, sizeof(double) * nx);

  stream<<<(nx + 255) / 256, 256>>>(nx, src, dest);
  checkCudaError(cudaDeviceSynchronize());

  checkSolutionStream(src, nx);

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

Programming models

- Dedicated programming languages
- Pragma-based approaches
  - OpenACC
  - OpenMP

```
int main(int argc, char *argv[]) {
  auto src  = new double[nx];
  auto dest = new double[nx];

#pragma omp target teams distribute parallel for data
map(from : src[0 : nx], to dest[0 : nx]) schedule(static)
    for (int i = 0; i < nx; ++i)
        dest[i] = src[i] + 1;

  checkSolutionStream(src, nx);

  delete[] src;
  delete[] dest;
}
```

Programming models

- Dedicated programming languages
- Pragma-based approaches
- Software layers
  - Kokkos
  - …

```
int main(int argc, char *argv[]) {
  Kokkos::initialize(c, argv);
  {
    Kokkos::View<double *> src ("src",  nx);
    Kokkos::View<double *> dest("dest", nx);

    Kokkos::parallel_for(Kokkos::RangePolicy<>(0, nx),
                         KOKKOS_LAMBDA(int i) {
                             dest(i) = src(i) + 1;
                         });
    Kokkos::fence();

    checkSolutionStream(src.data(), nx);
  }
  Kokkos::finalize();
}
```

Programming models

- Dedicated programming languages

➢ Full control – more evolved code for maximized performance potential

- Pragma-based approaches

➢ Easy to integrate – if everything works as intended

- Software layers

➢ Performance portability – ideally