Programming Techniques for Supercomputers

Friedrich-Alexander-Universität
Erlangen-Nürnberg

GPU Programming – CUDA & OpenMP

Sebastian Kuckuk

Erlangen National High Performance Computing Center (NHR@FAU)

NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

GPU Programming

Motivation, GPU Architecture and Theoretical Performance

GPUs in HPC – Motivation

- Promises
  - Massive parallelism and performance
  - Good performance in relation to energy (FLOPs per Watt)

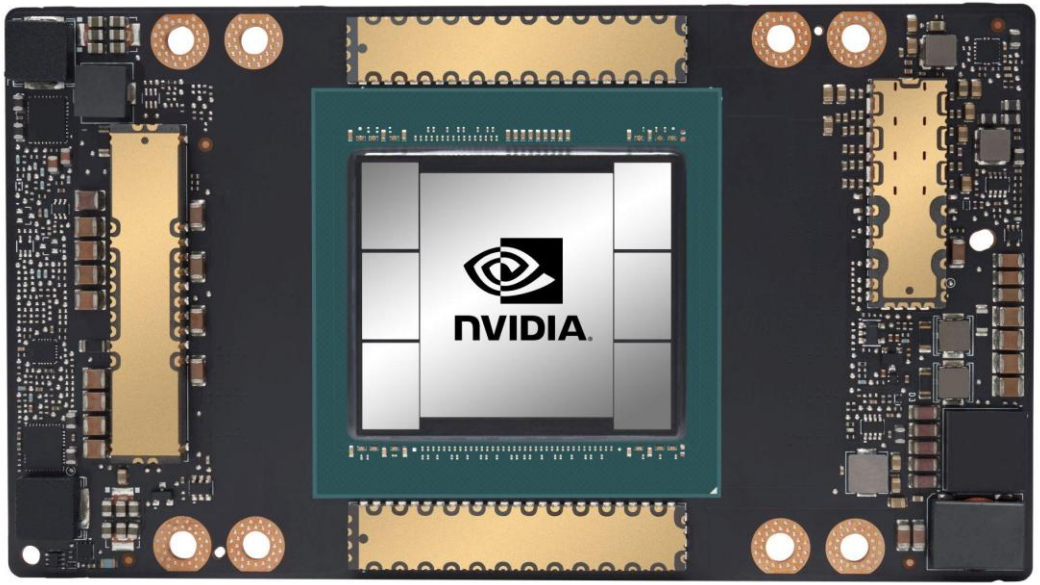
- Already widespread in the HPC landscape
  - c.f. Top500 list (https://www.top500.org/lists/top500/2025/11/)
    - 9 out of the top 10 supercomputers are equipped with GPUs
    - 4 NVIDIA (2 x GH200, H100, A100)
    - 4 AMD (MI300A, 3 x MI250X)
    - 1 Intel (GPU Max Series)

Helma is #58
Alex is #322
Fritz is #350

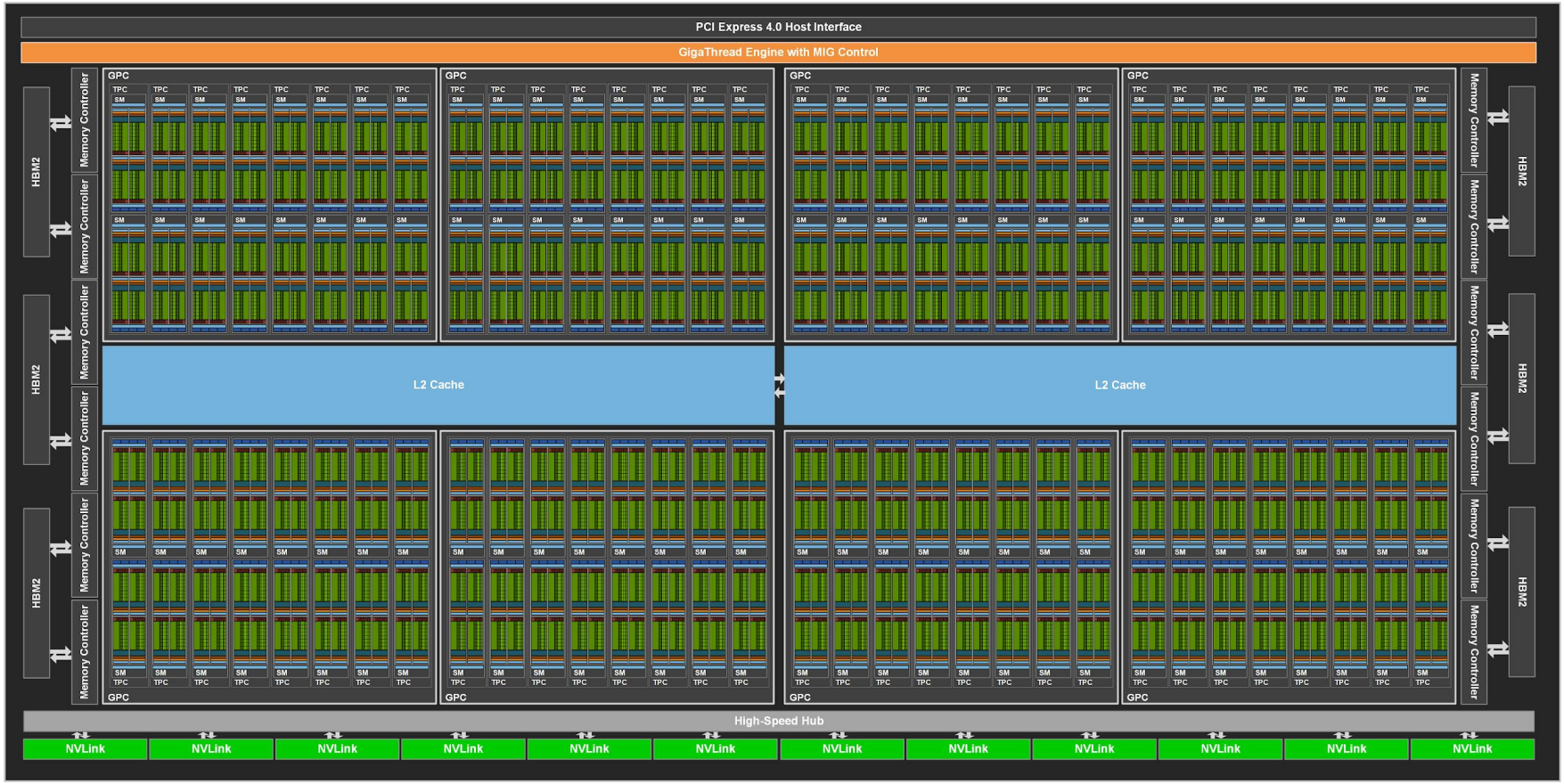
- Where does the performance come from?

GPU Architecture – Example: A100 in Alex

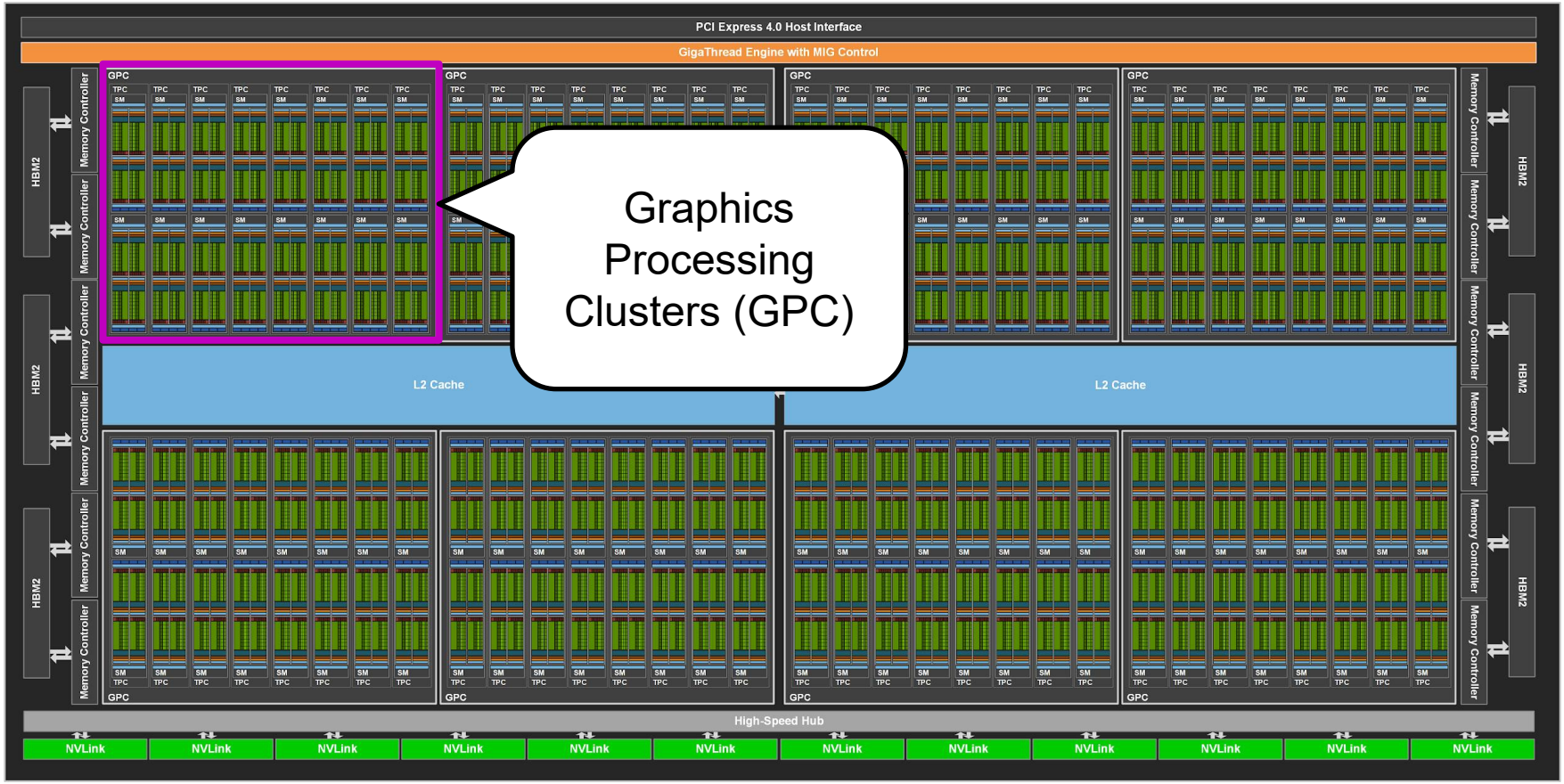
- Detailed documentation as whitepaper
  - https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf



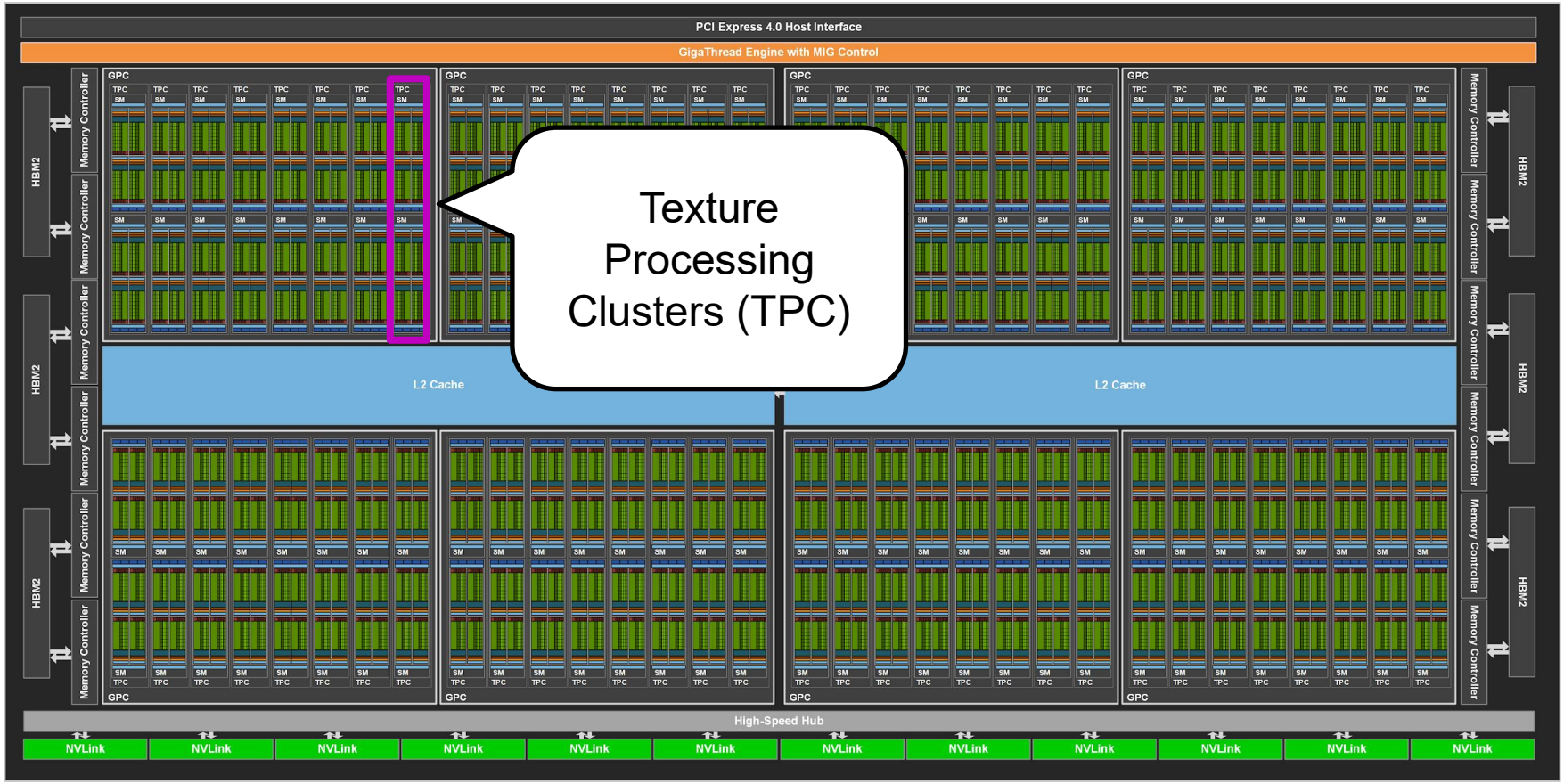
GPU Architecture – Example: A100 in Alex



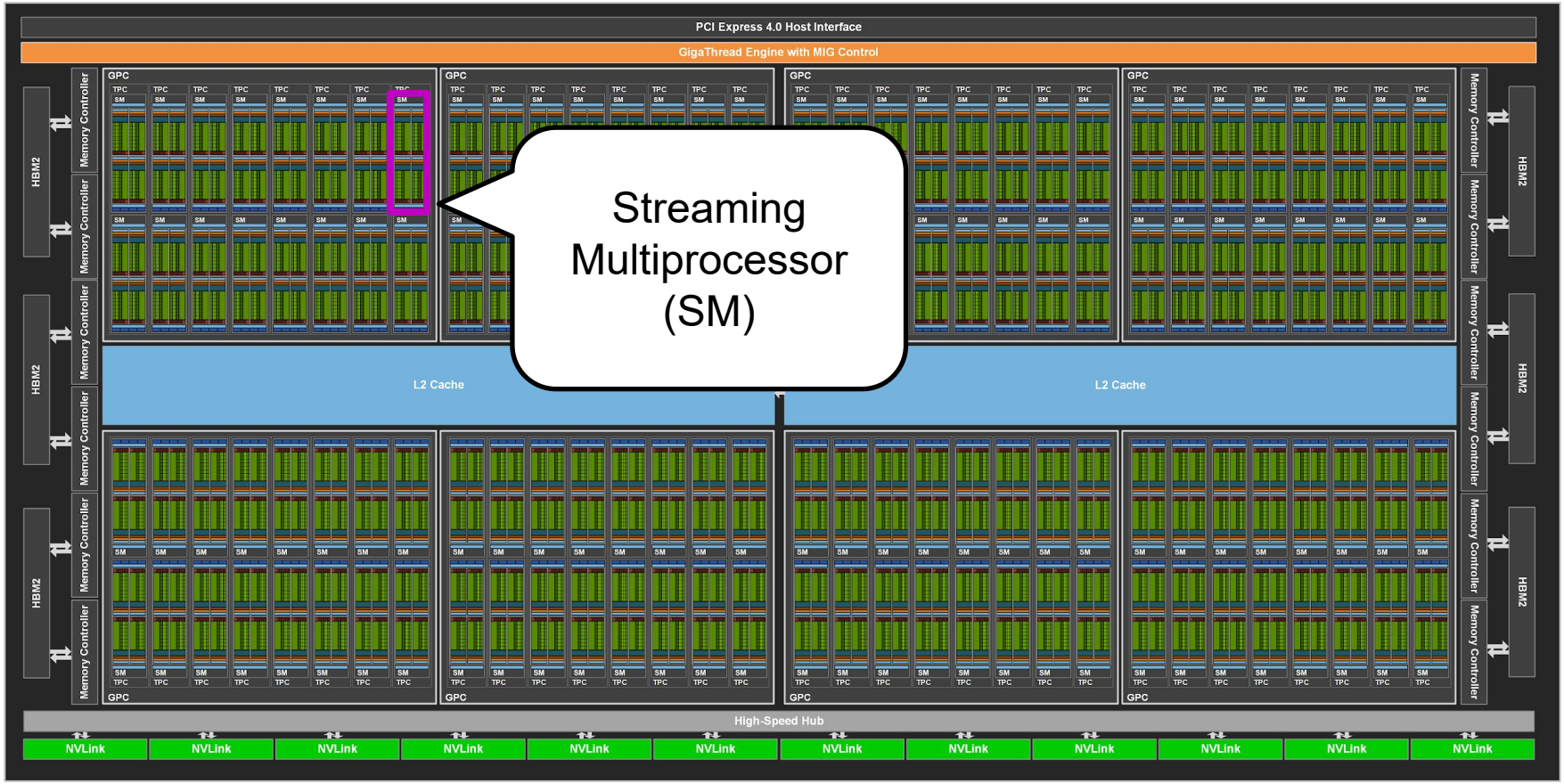
GPU Architecture – Example: A100 in Alex



GPU Architecture – Example: A100 in Alex



GPU Architecture – Example: A100 in Alex



GPU Architecture – Example: A100 in Alex

- Previous slides show ‘full configuration’

- One A100 GPU features
  - Seven Graphics Processing Clusters (GPCs), each with
  - Seven or eight Texture Processing Clusters (TPCs), each with
  - Two Streaming Multiprocessors (SMs)

➢ Total of 108 SMs

Some are deactivated as part of the manufacturing process

GPU Architecture – Example: A100 in Alex

- Total of 108 SMs, each with
- Four sub partitions, each with
- 16 INT32 units
  - Total: 108 * 4 * 16 = 6912
- 16 FP32 units
  - Total: 108 * 4 * 16 = 6912
- 8 FP64 units
  - Total: 108 * 4 * 8 = 3456
- One tensor core
  - Total: 108 * 4 = 432
  - Each: 256 FP16/FP32-mixed-prec. FMAs

GPU Architecture – Example: A100 in Alex

Performance: $P_{chip} = n_{core} \cdot n_{super}^{FP} \cdot n_{FMA} \cdot n_{SIMD} \cdot f$

- For FP32
  - $P_{chip} = 6912 \cdot 1 \cdot 2 \cdot 1 \cdot 1.41\ GF/s = 19.5\ TF/s$

Model each execution unit as core; each can do one scalar FMA per cycle

GPU Architecture – Example: A100 in Alex

Performance: $P_{chip} = n_{core} \cdot n_{super}^{FP} \cdot n_{FMA} \cdot n_{SIMD} \cdot f$

- For FP32
  - $P_{chip} = 6912 \cdot 1 \cdot 2 \cdot 1 \cdot 1.41\ GF/s = 19.5\ TF/s$
    $= 432 \cdot 1 \cdot 2 \cdot 16 \cdot 1.41\ GF/s = 19.5\ TF/s$

Model each SMSP as core; each can do one 16-wide FMAs per cycle

GPU Architecture – Example: A100 in Alex

Performance: $P_{chip} = n_{core} \cdot n_{super}^{FP} \cdot n_{FMA} \cdot n_{SIMD} \cdot f$

- For FP32
  - $P_{chip} = 6912 \cdot 1 \cdot 2 \cdot 1 \cdot 1.41\ GF/s = 19.5\ TF/s$
    $= 432 \cdot 1 \cdot 2 \cdot 16 \cdot 1.41\ GF/s = 19.5\ TF/s$
    $= 108 \cdot 4 \cdot 2 \cdot 16 \cdot 1.41\ GF/s = 19.5\ TF/s$

Model each SM as core; each can do four 16-wide FMAs per cycle

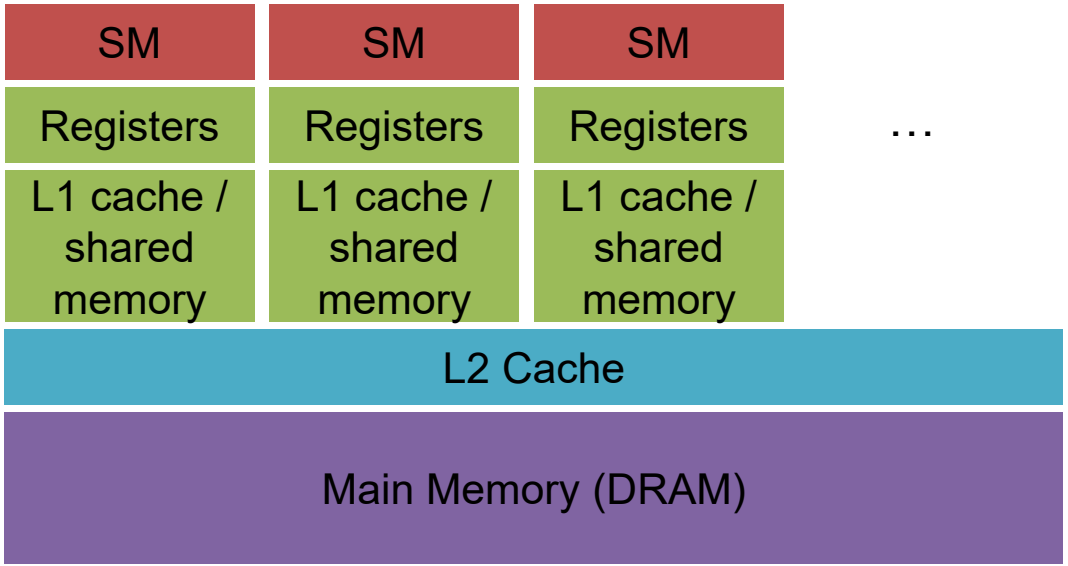
GPU Architecture – Example: A100 in Alex

Performance: $P_{chip} = n_{core} \cdot n_{super}^{FP} \cdot n_{FMA} \cdot n_{SIMD} \cdot f$

- For FP32
  - $P_{chip} = 6912 \cdot 1 \cdot 2 \cdot 1 \cdot 1.41\ GF/s = 19.5\ TF/s$
    $= 432 \cdot 1 \cdot 2 \cdot 16 \cdot 1.41\ GF/s = 19.5\ TF/s$
    $= 108 \cdot 4 \cdot 2 \cdot 16 \cdot 1.41\ GF/s = 19.5\ TF/s$
- For FP64: $9.7\ TF/s$
- For INT32: $19.5\ TO/s$

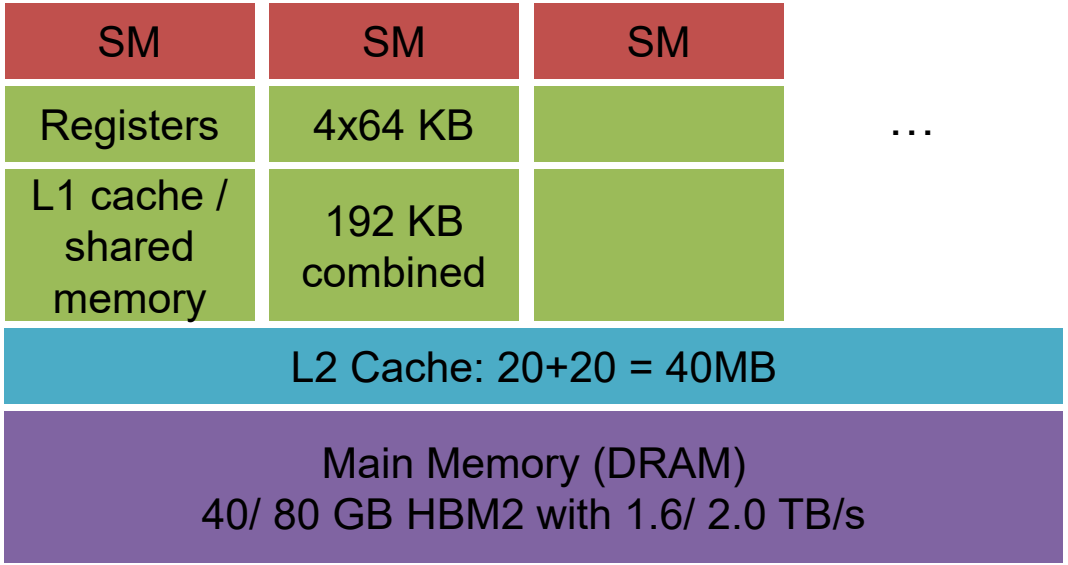
GPU Architecture – Example: A100 in Alex

- Memory also follows a specific hierarchy
- Not shown: constant memory, texture memory, …



GPU Architecture – Example: A100 in Alex

- Memory also follows a specific hierarchy
- Not shown: constant memory, texture memory, …



GPU Architecture – Example: A100 in Alex

Memory

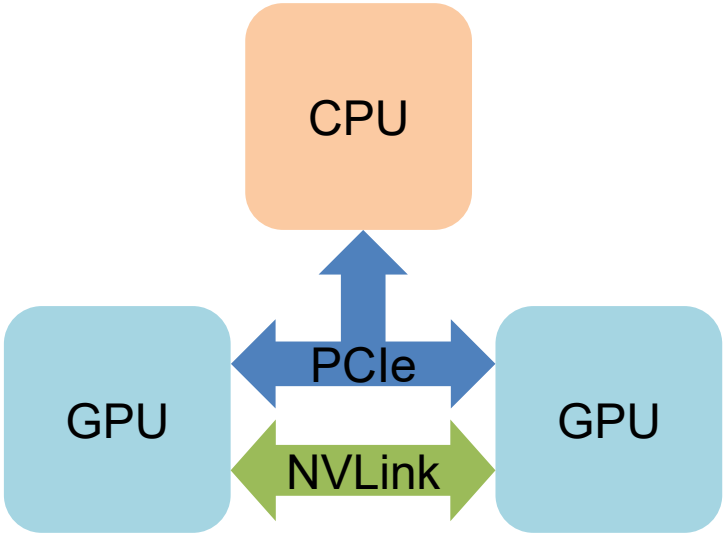
- Per GPU
  - 40/ 80 GB HBM2 main memory with 1555/ 2039 GB/s
  - 20+20 = 40MB L2 cache
- Per SM
  - 192 KB of combined shared memory and L1 cache
  - 4x64 KB register file

Different versions of the same base architecture with different performance characteristics are common

GPU Architecture – Example: A100 in Alex

Interconnect

- Reference: main memory with 1555/ 2039 GB/s
- Connection to CPU with PCIe 4.0 x16 with 31.5 GB/s *per direction*
- Connection to other GPUs in the same node via NVLink with 300 GB/s *per direction*



CPU-GPU Comparison

CPU

- Cores
  - Few but powerful
  - Branch prediction & speculative execution
  - Prefetchers
  - Low latency instructions
  - Out-of-Order Execution

GPU

- 'Cores'
  - Many but less powerful
  - Extreme SMT

CPU-GPU Comparison

CPU

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

GPU

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

CPU-GPU Comparison

CPU

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

➢ Ideal for irregular workload/ task parallelism

GPU

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

➢ Ideal for massively parallel structured computations

Aside: Latency

- Connecting parallelism, latency and bandwidth – Little's Law
  - $\mathrm{L} \cdot \mathrm{b_s} = \mathrm{N}$, where
  - L is the latency, $\mathrm{b_s}$ is the performance, i.e. bandwidth, and N the number of customers, i.e. the total data volume requested

- Example: Latency in absolute time ($\mathrm{L_t}$) or cycles ($\mathrm{L_{cyc}}$) on an A100 at low occupancy
  - $\mathrm{L_t} = \frac{\mathrm{N}}{\mathrm{b_s}} = \frac{64\,\frac{threads}{SM} \cdot 108\,SM \cdot 16\,\frac{B}{thread}}{157\,GB/s} = 704\,ns$
  - $\mathrm{L_{cyc}} = \mathrm{L_t} \cdot \mathrm{F} = 704\,ns \cdot 1.38\,GHz = 972\,cyc$

Aside: Latency

- Connecting parallelism, latency and bandwidth – Little's Law
  - $\mathrm{L} \cdot \mathrm{b_S} = \mathrm{N}$, where
  - L is the latency, $\mathrm{b_S}$ is the performance, i.e. bandwidth, and N the number of customers, i.e. the total data volume requested

- Example: Bandwidth on an A100 at low occupancy
  - $\mathrm{b_S} = \frac{\mathrm{N}}{\mathrm{L}_t} = \frac{128\,\frac{threads}{SM} \cdot 108\,SM \cdot 16\,\frac{B}{thread}}{500\,cyc \cdot (1.38\,GHz)^{-1}} = 610\,GB/s$

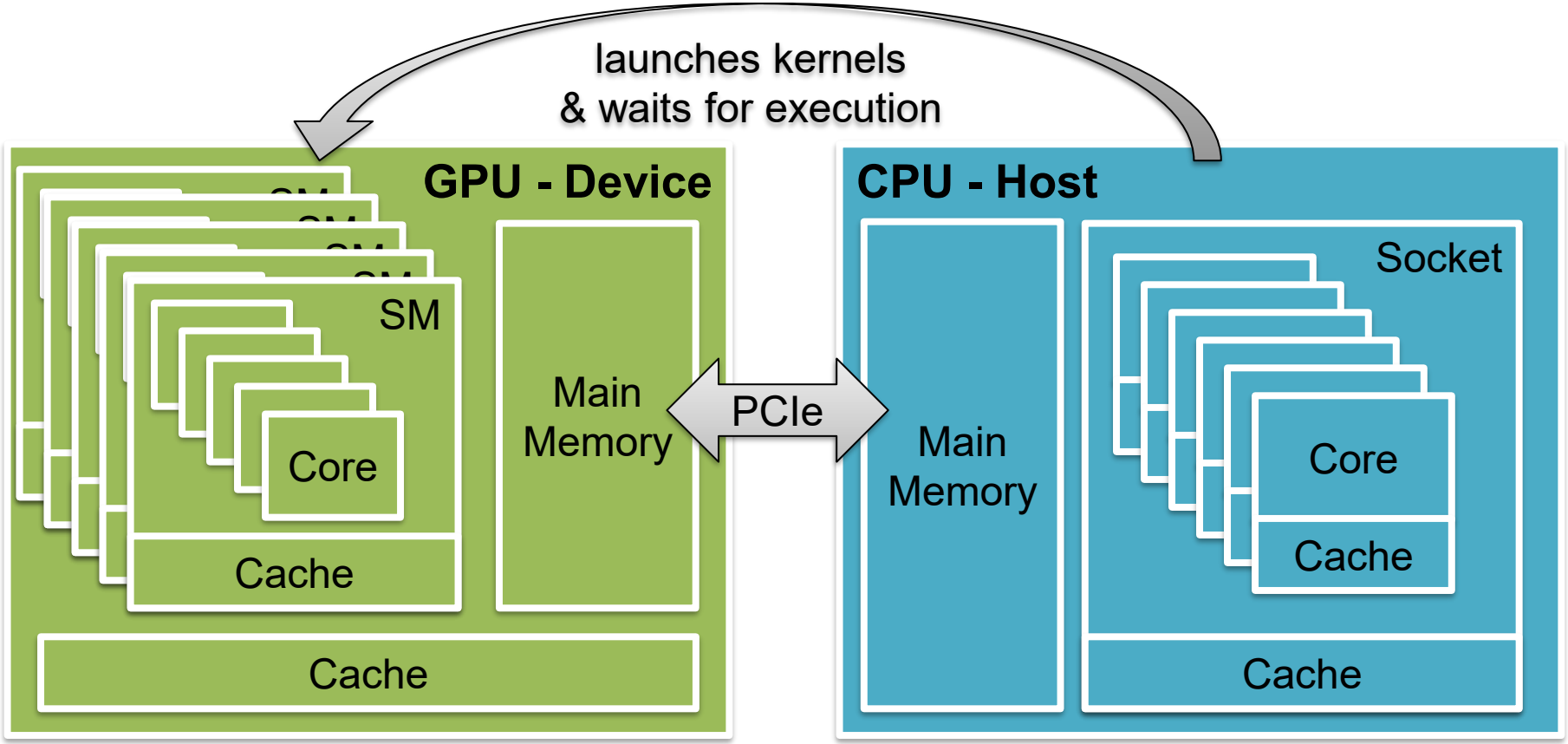
NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

GPU Programming

Abstract Workflow

Simplified Architecture



GPU Programming Approaches – Requirements

- Data management
  - (De-)Allocating memory on host and device
  - Transferring data between host and device
- Control flow management
  - Start execution on device
  - Wait for completion of execution
- Parallelism management
  - Express parallel operations
  - Mapping threads to work items

GPU Programming Approaches – Overview

- Dedicated programming language (extensions)
  - E.g. NVIDIA CUDA, AMD HIP, SYCL/ Intel OneAPI
  - ➢ Full control – more evolved code for maximized performance potential

GPU Programming Approaches – Overview

- Dedicated programming language (extensions)
  - E.g. NVIDIA CUDA, AMD HIP, SYCL/ Intel OneAPI

- Pragma-based approaches
  - E.g. OpenMP target offloading, OpenACC
  - ➢ Easy to integrate – let the compiler make critical choices (for better or worse)

GPU Programming Approaches – Overview

- Dedicated programming language (extensions)
  - E.g. NVIDIA CUDA, AMD HIP, SYCL/ Intel OneAPI
- Pragma-based approaches
  - E.g. OpenMP target offloading, OpenACC
- Software layers
  - E.g. Kokkos
    - ➢ Performance portability – if design & implementation of the framework are ideal

GPU Programming Approaches – Overview

- Dedicated programming language (extensions)
  - E.g. NVIDIA CUDA, AMD HIP, SYCL/ Intel OneAPI
- Pragma-based approaches
  - E.g. OpenMP target offloading, OpenACC
- Software layers
  - E.g. Kokkos
- Modern C++
  - E.g. std::par, thrust
  - ➢ Look and feel of modern C++ – miles of template errors included :)

Workflow (CPU)

1. Allocate data
2. Initialize data
3. Do work
4. Post-process data
5. De-Allocate data

Workflow (GPU)

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

Workflow (GPU)

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

Most complex part

NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

GPU Programming

Introduction to CUDA

GPU Kernel example

CPU code

```
void workOnCPU(
    int *data, size_t nx) {

  for (size_t i = 0; i < nx; ++i)
    data[i] = 0;
}

// in main
workOnCPU(data, nx);
```

GPU Kernel example

CPU code

```
void workOnCPU(
    int *data, size_t nx) {

  for (size_t i = 0; i < nx; ++i)
    data[i] = 0;
}

// in main
workOnCPU(data, nx);
```

Serial kernel (only for porting)

```
__global__ void workSerial(
    int *data, size_t nx) {

  for (size_t i = 0; i < nx; ++i)
    data[i] = 0;
}

// in main
workSerial<<<1, 1>>>(d_data, nx);
```

GPU Kernel example

CPU code

```
void workOnCPU(
    int *data, size_t nx) {

  for (size_t i = 0; i < nx; ++i)
    data[i] = 0;
}

// in main
workOnCPU(data, nx);
```

Serial kernel (only for porting)

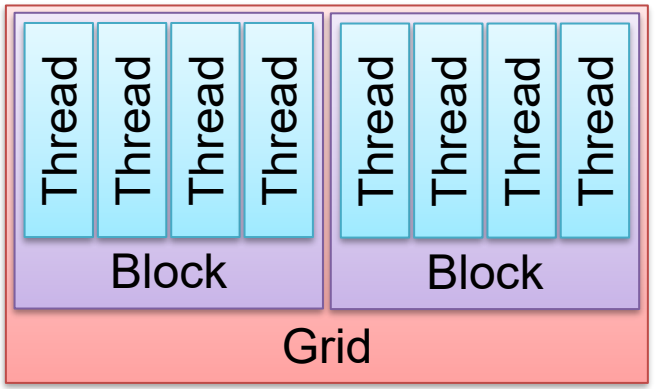
```
__global__ void workSerial(
    int *data, size_t nx) {

  for (size_t i = 0; i < nx; ++i)
    data[i] = 0;
}

// in main
workSerial<<<1, 1>>>(d_data, nx);
```

GPU data – see later slides

Execution configuration: one block with one thread each

GPU Kernel Concepts

- CPU code *launches kernels* that then execute *asynchronously* on the GPU
- *Kernels* are executed by many *threads*
- The number of threads is specified by the *execution configuration*
- Threads are organized hierarchically
  - Grids > blocks > threads



GPU Kernel Concepts

- CPU code *launches kernels* that then execute *asynchronously* on the GPU
- *Kernels* are executed by many *threads*
- The number of threads is specified by the *execution configuration*
- Threads are organized hierarchically (Grids > blocks > threads)
- Each *thread* executes the whole *kernel body*
- *Built-in variables* allow identifying *thread ids* and mapping to work items

GPU Kernel example

Serial kernel (only for porting)

```
__global__ void workSerial(
    int *data, size_t nx) {

  for (size_t i = 0; i < nx; ++i)


    data[i] = 0;
}

// in main
workSerial<<<1, 1>>>(d_data, nx);
```

GPU Kernel example

Serial kernel (only for porting)

```
__global__ void workSerial(
    int *data, size_t nx) {

  for (size_t i = 0; i < nx; ++i)


    data[i] = 0;
}

// in main
workSerial<<<1, 1>>>(d_data, nx);
```

Parallel kernel (without checks)

```
__global__ void work(
    int *data, size_t nx) {

  size_t i = blockIdx.x * blockDim.x
           + threadIdx.x;

  data[i] = 0;
}

// in main
work<<<nx / 256, 256>>>(d_data, nx);
```

GPU Kernel example

Serial kernel (only for porting)

```
__global__ void workSerial(
    int *data, size_t nx) {

  for (size_t i = 0; i < nx; ++i)

    data[i] = 0;
}

// in main
workSerial<<<1, 1>>>(d_data, nx);
```

Parallel kernel (without checks)

```
__global__ void work(
    int *data, size_t nx) {

  size_t i = blockIdx.x * blockDim.x
           + threadIdx.x;

  data[i] = 0;
}

// in main
work<<<nx / 256, 256>>>(d_data, nx);
```

Unique *data index* (or *global thread index*)

Each thread updates a single item

#blocks, #threads per block (assumes evenly divisible nx)

GPU Kernel example

Parallel kernel (without checks)

```
__global__ void work(
    int *data, size_t nx) {

  size_t i = blockIdx.x * blockDim.x
           + threadIdx.x;


  data[i] = 0;
}

// in main
work<<<nx / 256, 256>>>(d_data, nx);
```

GPU Kernel example

Parallel kernel (without checks)

```
__global__ void work(
    int *data, size_t nx) {

  size_t i = blockIdx.x * blockDim.x
           + threadIdx.x;


  data[i] = 0;
}

// in main
work<<<nx / 256, 256>>>(d_data, nx);
```

Parallel kernel

```
__global__ void work(
    int *data, size_t nx) {

  size_t i = blockIdx.x * blockDim.x
           + threadIdx.x;

  if (i < nx)
    data[i] = 0;
}

// in main
work<<<(nx+255)/256, 256>>>(d_data, nx);
```

GPU Kernel example

Parallel kernel (without checks)

```
__global__ void work(
    int *data, size_t nx) {

  size_t i = blockIdx.x * blockDim.x
           + threadIdx.x;



  data[i] = 0;
}

// in main
work<<<nx / 256, 256>>>(d_data, nx);
```

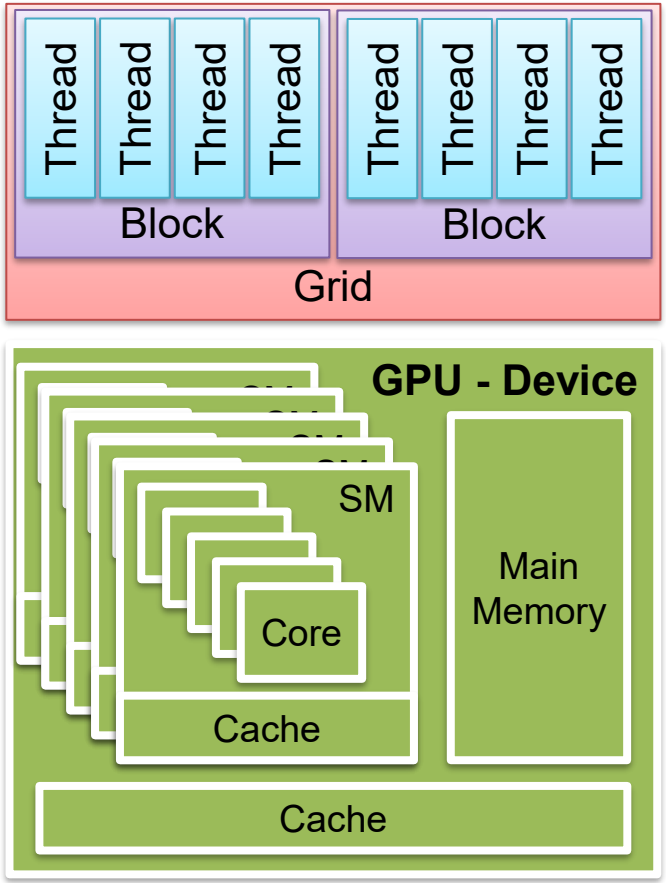
Parallel kernel

```
__global__ void work(
    int *data, size_t nx) {

  size_t i = blockIdx.x * blockDim.x
           + threadIdx.x;

  if (i < nx)
    data[i] = 0;
}

// in main
work<<<(nx+255)/256, 256>>>(d_data, nx);
```

Out-of-bounds check

Round-up division

CUDA Mapping



- **Grids** are mapped to **devices** (one grid per kernel)
- **Blocks** are mapped to **SMs** (A100: 108 SMs)
- **Threads** are mapped to **'cores'** (A100: 64 'cores'/ SM for FP32)
- **Threads of a block** are executed in **warps** (groups of 32 threads)

GPU Kernel example

Parallel kernel

```
__global__ void work(
    int *data, size_t nx) {

  size_t i = blockIdx.x * blockDim.x
           + threadIdx.x;


  if (i < nx)
    data[i] = 0;
}

// in main
work<<<(nx+255)/256, 256>>>(d_data, nx);
```

GPU Kernel example

Parallel kernel

```
__global__ void work(
    int *data, size_t nx) {

  size_t i = blockIdx.x * blockDim.x
           + threadIdx.x;


  if (i < nx)
    data[i] = 0;
}

// in main
work<<<(nx+255)/256, 256>>>(d_data, nx);
```

Parallel kernel w/ grid-stride loop

```
__global__ void work(
    int *data, size_t nx) {

  size_t i = blockIdx.x * blockDim.x
           + threadIdx.x;
  size_t stride = gridDim.x * blockDim.x;

  for ( ; i < nx; i += stride)
    data[i] = 0;
}

// in main
work<<<32 * 108, 256>>>(d_data, nx);
```

GPU Kernel example

Parallel kernel

```
__global__ void work(
    int *data, size_t nx) {

  size_t i = blockIdx.x * blockDim.x
           + threadIdx.x;


  if (i < nx)
    data[i] = 0;
}

// in main
work<<<(nx+255)/256, 256>>>(d_data, nx);
```

Parallel kernel w/ grid-stride loop

```
__global__ void work(
    int *data, size_t nx) {

  size_t i = blockIdx.x * blockDim.x
           + threadIdx.x;
  size_t stride = gridDim.x * blockDim.x;

  for ( ; i < nx; i += stride)
    data[i] = 0;
}

// in main
work<<<32 * 108, 256>>>(d_data, nx);
```

Each thread updates zero, one, or more items

Arbitrary number of blocks chosen to be a multiple of #SM

NHR FAU

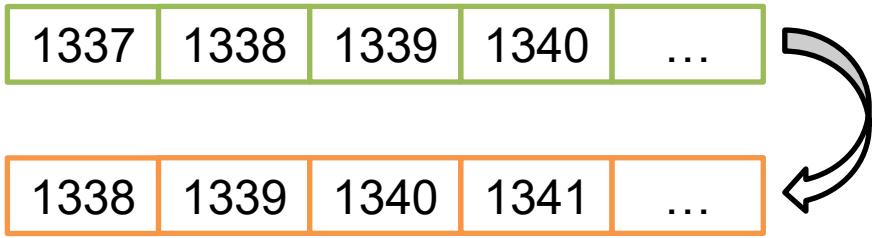
FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

GPU Programming

GPU Programming Example – Stream Benchmark

Example Application

- Copy array and increase each element by 1 (on the GPU)

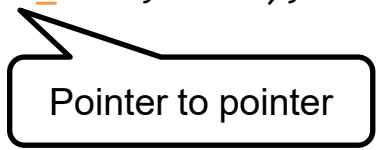


1. Allocate Data

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

1. Allocate Data

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

  // allocate _host_ arrays
  double *src, *dest;
  cudaMallocHost(&src,  size);
  cudaMallocHost(&dest, size);

  // allocate _device_ arrays
  double *d_src, *d_dest;
  cudaMalloc(&d_src,  size);
  cudaMalloc(&d_dest, size);
```



2. Initialize data on CPU

```
int main(int argc, char *argv[]) {
  // allocate

  initOnCPU(src, nx);
```

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

3. Copy data from CPU to GPU

```
int main(int argc, char *argv[]) {
  // allocate

  initOnCPU(src, nx);

  //          to     from size  direction
  cudaMemcpy(d_src, src, size, cudaMemcpyHostToDevice);
```

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;
}
```

4. Launch GPU kernels

Denotes a kernel to be launched from the host and executed on the device

`src` and `dest` are device arrays

```
__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;
}
```

Build-in thread variables

Guard against out-of-bounds

Each thread performs one update

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;
}

int main(int argc, char *argv[]) {
  // allocate, init and copy

  auto numThreadsPerBlock = 256;
  auto numBlocks = (nx + numThreadsPerBlock - 1) / numThreadsPerBlock;

  copyOnGPU<<<numBlocks, numThreadsPerBlock>>>(d_src, d_dest, nx);
```

Ceiling/ round-up division

4. Launch GPU kernels – Alternative 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;
}

int main(int argc, char *argv[]) {
  // allocate, init and copy

  auto numThreadsPerBlock = 256;
  auto numBlocks = 108 * 32;

  copyOnGPU<<<numBlocks, numThreadsPerBlock>>>(d_src, d_dest, nx);
```

Number of blocks is now decoupled from nx; can be tuned for the #SM

5. Do independent work on CPU

```
int main(int argc, char *argv[]) {
  // allocate, init and copy

  copyOnGPU<<<numBlocks, numThreadsPerBlock>>>(d_src, d_dest, nx);

  // copyOnGPU runs asynchronously => additional CPU work could be done here
```

6. Synchronize GPU

```
int main(int argc, char *argv[]) {
  // allocate, init and copy

  copyOnGPU<<<numBlocks, numThreadsPerBlock>>>(d_src, d_dest, nx);

  // copyOnGPU runs asynchronously => additional CPU work could be done here

  cudaDeviceSynchronize();
```

7. Copy data from GPU to CPU

```
int main(int argc, char *argv[]) {
  // allocate, init and copy

  copyOnGPU<<<numBlocks, numThreadsPerBlock>>>(d_src, d_dest, nx);

  // copyOnGPU runs asynchronously => additional CPU work could be done here

  cudaDeviceSynchronize();

  //          to    from    size  direction
  cudaMemcpy(dest, d_dest, size, cudaMemcpyDeviceToHost);
```

7. Copy data from GPU to CPU

```
int main(int argc, char *argv[]) {
  // allocate, init and copy

  copyOnGPU<<<numBlocks, numThreadsPerBlock>>>(d_src, d_dest, nx);

  // copyOnGPU runs asynchronously => additional CPU work could be done here

  cudaDeviceSynchronize();

  //          to    from    size  direction
  cudaMemcpy(dest, d_dest, size, cudaMemcpyDeviceToHost);
```

Technically not necessary since cudaMemcpy is blocking

8. Post-process data on CPU

```
int main(int argc, char *argv[]) {
  // allocate, init and copy

  copyOnGPU<<<numBlocks, numThreadsPerBlock>>>(d_src, d_dest, nx);

  // copyOnGPU runs asynchronously => additional CPU work could be done here

  cudaDeviceSynchronize();

  //          to    from    size  direction
  cudaMemcpy(dest, d_dest, size, cudaMemcpyDeviceToHost);

  checkOnCPU(dest, nx);
```

8. Post-process data on CPU

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


int main(int argc, char *argv[]) {
  // steps 1 through 6

  //          to    from    size  direction
  cudaMemcpy(dest, d_dest, size, cudaMemcpyDeviceToHost);

  checkOnCPU(dest, nx);
```

8. Post-process data on CPU

```
int main(int argc, char *argv[]) {
  // steps 1 through 7

  checkOnCPU(dest, nx);

  // de-allocate _device_ arrays
  cudaFree(d_src);
  cudaFree(d_dest);

  // de-allocate _host_ arrays
  cudaFreeHost(src);
  cudaFreeHost(dest);

  return 0;
}
```

Running Applications

- Compilation with either
  - NVCC
    - CUDA compiler, relays CPU code to host compiler (e.g. GCC)
    - `nvcc -arch=sm_80 -O3 -o my-cuda-app my-cuda-app.cu`
    - `-arch` sets the GPU to optimize for (https://en.wikipedia.org/wiki/CUDA#GPUs_supported)
  - or NVC++
    - C++ compiler that supports GPU programming via OpenACC, OpenMP, std::par, ...
    - `nvc++ -O3 -o my-cuda-app my-cuda-app.cu`
    - Auto-detects current GPU – compile on the system you want to execute on
- Execution as usual: `./my-cuda-app`

Running Applications

- Compilation with either NVCC or NVC++
- Execution as usual: `./my-cuda-app`
- Note: On many HPC systems compilers are only available after loading the containing module (including TinyGPU and Alex)
  - Get information about available modules

    ```
    module avail
    ```

  - Load required modules

    ```
    module load nvhpc
    module load cuda
    ```

Profiling Applications

- Getting information about available hardware
  - `nvidia-smi`
- Profiling with results on command line
  - nsight systems
  - `nsys profile --stats=true \ ./my-cuda-app`
  - nsight compute
  - `ncu ./my-cuda-app`
- Profiling with result files
  - nsight systems
  - `nsys profile -o output-file \ ./my-cuda-app`
  - nsight compute
  - `ncu -o output-file \ ./my-cuda-app`
- Open generated profiles with the corresponding GUIs

Aside: Error Handling

- Most CUDA API function return an error code

```
cudaError_t code = cudaDeviceSynchronize();
if (cudaSuccess != code)
    std::cerr << cudaGetErrorString(code) << std::endl;
```

- Kernel launches return *no* error code
  - Query for launch errors with `cudaGetLastError`

- Implementing an inline function/ macro/ … can be helpful

Error Handling – Helper Function

```
#define checkCudaError(...) \
    checkCudaErrorImpl(__FILE__, __LINE__, __VA_ARGS__)

inline void checkCudaErrorImpl(const std::string &file, int line, cudaError_t code) {
    if (cudaSuccess != code) {
        std::cerr << "CUDA Error (" << file << " : " << line << ") --- "
                  << cudaGetErrorString(code) << std::endl;
        exit(1);
    }
}
```

Error Handling – Example

```
checkCudaError(cudaMalloc( /* ... */ ));

checkCudaError(cudaMemcpy( /* ... */ ));

copyOnGPU<<<numBlocks, numThreadsPerBlock>>>(d_src, d_dest, nx);
checkCudaError(cudaGetLastError());

checkCudaError(cudaDeviceSynchronize());
```

Error Handling – Example

```
checkCudaError(cudaMalloc( /* ... */ ));

checkCudaError(cudaMemcpy( /* ... */ ));

copyOnGPU<<<numBlocks, numThreadsPerBlock>>>(d_src, d_dest, nx);
checkCudaError(cudaGetLastError());

checkCudaError(cudaDeviceSynchronize());
```

May, e.g., report out of memory

Reports kernel launch errors

May catch asynchronous errors that happened during kernel execution

Aside: Managed Memory

- 'Classic'
  - Distinct allocations for host and device
  - Explicit copies from H-D/ D-H
- Managed
  - One unified allocation shared between host and device
  - Migration between host and device on access
  - Small migration granularity (e.g. page size)

Managed Memory Code Changes

- 1. Allocate Data (managed) (explicit)

```
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);

    // ...
}
```

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

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

Managed Memory Code Changes

- 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`

Managed Memory Code Changes

- 4. Launch GPU kernels

```
copyOnGPU<<<numBlocks, numThreads>>>(
        src, dest, nx);
```

```
copyOnGPU<<<numBlocks, numThreads>>>(
        d_src, d_dest, nx);
```

Managed Memory Code Changes

- 7. Copy data from GPU to CPU

```
// computation

cudaMemPrefetchAsync(dest, size,
                     cudaCpuDeviceId);

// ...
```

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

Managed Memory Code Changes

- 9. De-Allocate Data

```
// post-processing
```

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

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

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

NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

GPU Programming

Introduction to OpenMP Target Offloading

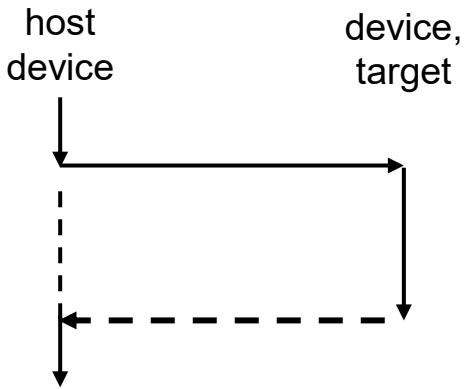
OpenMP Target Offloading

- OpenMP supports GPU offloading since version 4.0
- Remember: OpenMP standard != compiler implementation
- GPUs of all mayor vendors are supported
  - But performance may drastically vary from GPU to GPU and from compiler to compiler
- The following slides are based on *Introduction to OpenMP*, a course offered by NHR@FAU

➢ The full course covers concepts in more depth

Introduction

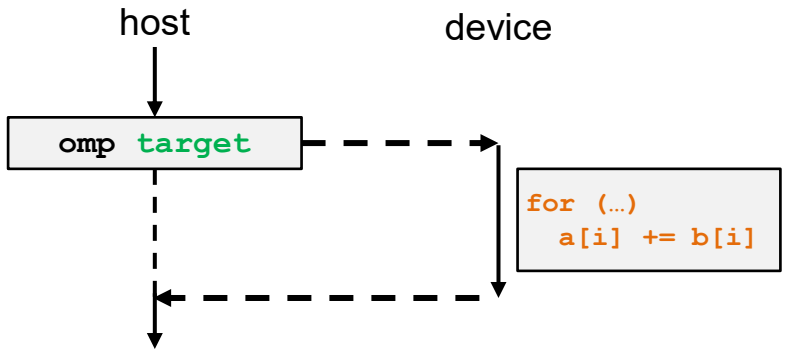
- Execute code on a device
  - Not necessarily a GPU, can also be an FPGA, DSP, …
- **Target**: device where code and data is offloaded to
- Execution always starts on the **host** device



Offloading Code to the Target

- target: execute associated structured block on the device
- Execution on the target is initially single threaded
- Host waits until offloaded code completes

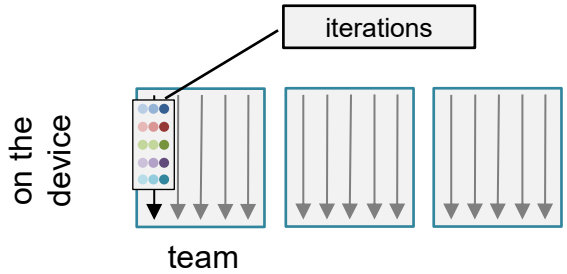
```
int a[1024], b[1024];
/* init a and b */
#pragma omp target
{
  for (int i = 0; i < 1024; ++i)
    a[i] += b[i];
} /* wait until complete */
```



Generating Parallelism on the Target

- Target construct alone does not generate parallelism

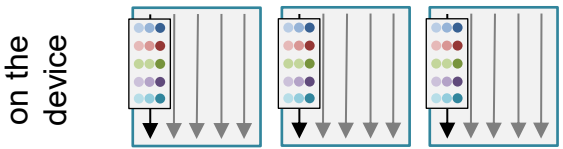
```
#pragma omp target
for (int i = 0; i < 1024; ++i)
  a[i] += b[i];
```



Generating Parallelism on the Target

- `teams` construct
  - generate **league of teams**
  - a team has only one initial thread
  - each team executes the same code

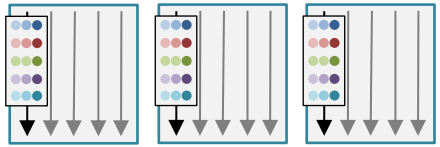
```
#pragma omp target teams
for (int i = 0; i < 1024; ++i)
  a[i] += b[i];
```



Generating Parallelism on the Target

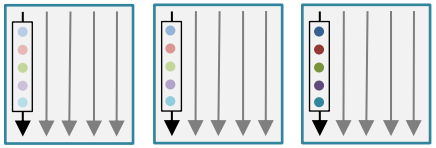
- `teams` construct
  - generate **league of teams**
  - a team has only one initial thread
  - each team executes the same code

```
#pragma omp target teams
for (int i = 0; i < 1024; ++i)
  a[i] += b[i];
```



- `distribute` construct
  - distributes iteration space of associated loop(s) over teams

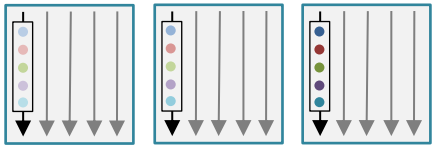
```
#pragma omp target teams distribute
for (int i = 0; i < 1024; ++i)
  a[i] += b[i];
```



Generating Parallelism on the Target

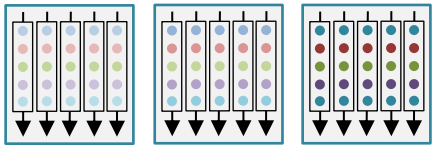
- `distribute` construct
  - Distributes iteration space of associated loop(s) over teams

```
#pragma omp target teams distribute
for (int i = 0; i < 1024; ++i)
  a[i] += b[i];
```



- `parallel` construct
  - Generate parallel region with multiple threads inside each team

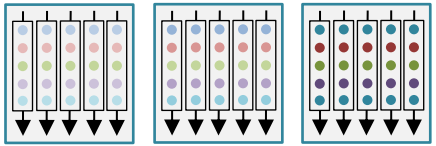
```
#pragma omp target teams distribute \
                    parallel
for (int i = 0; i < 1024; ++i)
  a[i] += b[i];
```



Generating Parallelism on the Target

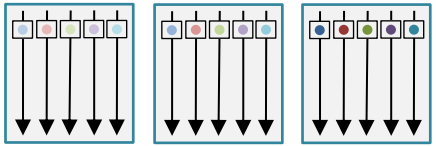
- `parallel` construct
  - Generate parallel region with multiple threads inside each team

```
#pragma omp target teams distribute \
                    parallel
for (int i = 0; i < 1024; ++i)
  a[i] += b[i];
```



- `for` construct
  - Worksharing loop
  - Distribute team's iteration space over all threads inside a team

```
#pragma omp target teams distribute \
                    parallel for
for (int i = 0; i < 1024; ++i)
  a[i] += b[i];
```



Generating Parallelism on the Target

- Additionally: `simd` construct – use SIMD lanes in each thread

➢ How each directive is mapped to GPU entities depends on the compiler

- Specializing the parallelization is possible (`num_teams`, `thread_limit`, …)

- Reductions are supported as usual

Data Mapping

- Mapping data consists of three phases

1. map-enter: on entry to `target` region
   - Storage allocation and copy of mapped variable to device
2. Compute: execution of `target` region, device threads access mapped variable
3. map-exit: on exist of `target` region, variable is unmapped from device
   - Copy of corresponding variable to host, deallocation

Data Mapping

- Mapping variables, etc. can be done implicit, or explicit, e.g.

```
#pragma omp target map(tofrom : a) map(to : b[0 : nx])
// block
```

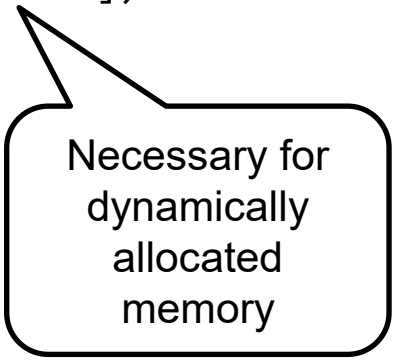
- Mapping causes a presence check
  - Copy to device only if not already present

Data Mapping

- Mapping variables, etc. can be done implicit, or explicit, e.g.

```
#pragma omp target map(tofrom : a) map(to : b[0 : nx])
// block
```

- Mapping causes a presence check
  - Copy to device only if not already present

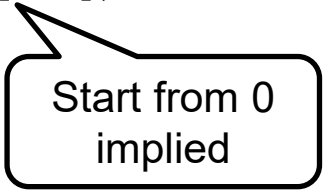


Data Mapping

```
// Three options
// map as part of target region
#pragma omp target            map(tofrom : a[:nx]) map(to : b[:nx])
{ /* ... */ }
```

Data Mapping

```
// Three options
// map as part of target region
#pragma omp target                map(tofrom : a[:nx]) map(to : b[:nx])
{ /* ... */ }
```



Data Mapping

```
// Three options
// map as part of target region
#pragma omp target            map(tofrom : a[:nx]) map(to : b[:nx])
{ /* ... */ }

// target data region
#pragma omp target data       map(tofrom : a[:nx]) map(to : b[:nx])
{
    #pragma omp target
    { /* ... */ }
}
```

Data Mapping

```
// Three options
// map as part of target region
#pragma omp target                map(tofrom : a[:nx]) map(to : b[:nx])
{ /* ... */ }

// target data region
#pragma omp target data           map(tofrom : a[:nx]) map(to : b[:nx])
{
    #pragma omp target
    { /* ... */ }
}
```

One or more target regions plus optional host code; may be in other functions

Data Mapping

```
// Three options
// map as part of target region
#pragma omp target            map(tofrom : a[:nx]) map(to : b[:nx])
{ /* ... */ }

// target data region
#pragma omp target data       map(tofrom : a[:nx]) map(to : b[:nx])
{ /* #pragma omp target ... */ }

// target data enter and exit
#pragma omp target data enter map(to   : a[:nx])   map(to      : b[:nx])
/* ... */
#pragma omp target data exit  map(from : a[:nx])   map(release : b[:nx])
```

Data Mapping

```
// Three options
// map as part of target region
#pragma omp target                map(tofrom : a[:nx]) map(to : b[:nx])
{ /* ... */ }

// target data region
#pragma omp target data           map(tofrom : a[:nx]) map(to : b[:nx])
{ /* #pragma omp target ... */ }

// target data enter and exit
#pragma omp target data enter map(to   : a[:nx])   map(to      : b[:nx])
/* ... */
#pragma omp target data exit  map(from : a[:nx])   map(release : b[:nx])
```

Useful if `enter` and `exit` are in different functions/ classes/ …

Prominent use cases:
class c'tor/d'tor;
class member functions

GPU Kernel example

CPU code

```
void workOnCPU(int *data, size_t nx) {


  for (size_t i = 0; i < nx; ++i)
    data[i] = 0;
}

// in main



workOnCPU(data, nx);
```

GPU Kernel example

CPU code

```
void workOnCPU(int *data, size_t nx) {


  for (size_t i = 0; i < nx; ++i)
    data[i] = 0;
}

// in main



workOnCPU(data, nx);
```

GPU code

```
void workOnGPU(*data, size_t nx) {
#pragma omp target \
    teams distribute parallel for
for (size_t i = 0; i < nx; ++i)
    data[i] = 0;
}

// in main
#pragma omp target data \
    map(from : data[:nx])
{
    workOnGPU(data, nx);
}
```

GPU Kernel example

CPU code

```
void workOnCPU(int *data, size_t nx) {


  for (size_t i = 0; i < nx; ++i)
    data[i] = 0;
}

// in main



workOnCPU(data, nx);
```

GPU code

```
void workOnGPU(*data, size_t nx) {
#pragma omp target \
    teams distribute parallel for
for (size_t i = 0; i < nx; ++i)
    data[i] = 0;
}

// in main
#pragma omp target data \
    map(from : data[:nx])
{
    workOnGPU(data, nx);
}
```

Implies allocation of data at the beginning of the region

Running Applications

- Compilation with NVC++
  - C++ compiler that also supports OpenACC, OpenMP, std::par, ...
  - `nvc++ -O3 -mp=gpu -target=gpu -o my-omp-app my-omp-app.cpp`
  - Auto-detects current GPU – compile on the system you want to execute on

- For using managed memory
  - ~~Add this pragma to your code:~~
    - ~~`#pragma omp requires unified_shared_memory`~~
  - And add `-gpu=mem:managed` to the compiler flags
    - replaces the deprecated `-gpu=managed`

- Execution as usual: `./my-omp-app`

GPU Kernel Example (Managed Memory)

CPU code

```
void workOnCPU(int *data, size_t nx) {


  for (size_t i = 0; i < nx; ++i)
    data[i] = 0;
}

// in main
workOnCPU(data, nx);
```

GPU code

```
#pragma omp requires unified_shared_memory

void workOnGPU(*data, size_t nx) {
#pragma omp target \
    teams distribute parallel for
for (size_t i = 0; i < nx; ++i)
    data[i] = 0;
}

// in main
workOnGPU(data, nx);
```

GPU Kernel Example (Managed Memory)

CPU code

```
void workOnCPU(int *data, size_t nx) {


  for (size_t i = 0; i < nx; ++i)
    data[i] = 0;
}

// in main
workOnCPU(data, nx);
```

GPU code

```
#pragma omp requires unified_shared_memory

void workOnGPU(*data, size_t nx) {
#pragma omp target \
    teams distribute parallel for
for (size_t i = 0; i < nx; ++i)
    data[i] = 0;
}

// in main
workOnGPU(data, nx);
```

No data region specification

NHR FAU

FAU Friedrich-Alexander-Universität Erlangen-Nürnberg

GPU Programming

Outlook

Recommended Implementation Process

1. Implement CPU-only application, check correctness

2. Revise memory management to allocate GPU and CPU memory

3. Transform CPU functions to GPU kernels with grid-stride loops
   1. Execute serially
   2. Execute parallel

4. Repeat step 3 as necessary, then optimize
   1. Remove unnecessary synchronization
   2. Remove unnecessary copies

Recommended Implementation Process

5. Profile and optimize
   - Nsight systems to get whole application view
   - Nsight compute to zero in on performance behavior of single kernels
   - Try to optimize for bottlenecks, e.g.
     - Long PCIe transfer periods – try to overlap computation and data transfers
     - Short kernel runtimes – try to fuse multiple kernels
     - Not enough parallelism – try running multiple kernels concurrently (c.f. CUDA streams)

Outlook

- HPC GPU systems with different memory integrations
  - NVIDIA GraceHopper super chip
  - AMD MI300A APU
  - Might require different ways of programming (for full performance)

- Additional resources
  - NHR courses (https://hpc.fau.de/teaching/tutorials-and-courses/)
  - Lectures
    - High End Simulation in Practice (HESP)
    - Programming Techniques for Supercomputers (PTfS)
  - GTC (https://www.nvidia.com/gtc/)