Programming OpenMP

(GPU) Offloading

Christian Terboven
Jannis Klinkenberg

RWTH AACHEN UNIVERSITY

Motivation

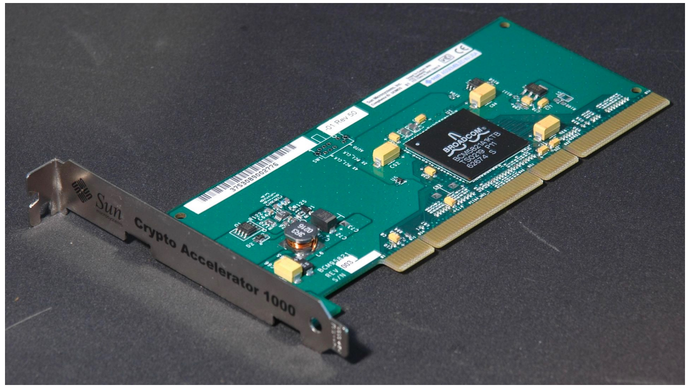
Hardware Accelerators

- Definition: A hardware component to speed up some aspect of the computing workload.

Computation: Intel 80386DX CPU with 80387DX Math Coprocessor

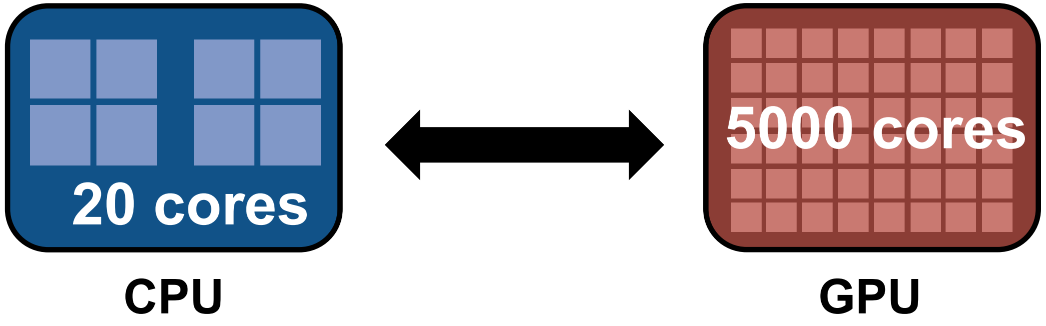
Generic FPGA: A Stratix IV FPGA from Altera

Digital signal processor (DSP), e.g. in music instruments



Encryption: PCI-X Crypto Accelerator

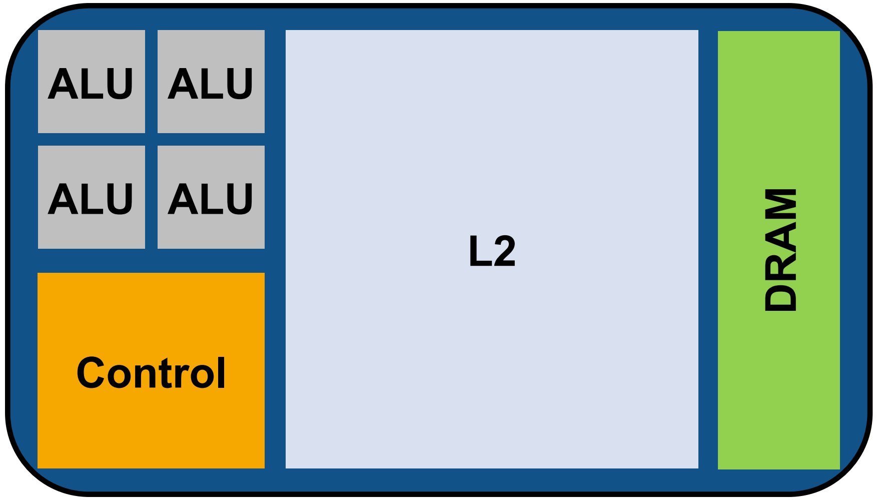
Comparison CPU ⇔ GPU



- GPU-Threads
  - Scheduled chain of instructions running on a CUDA core (basically a pipeline)
  - Light-weight, little creation overhead, fast context switching
  - SMT on CPU: few thread share core to better utilize execution units
  - GPU threads: up to 32 threads per core to hide memory latencies

- Lots of parallelism needed on GPU to get good performance!

Comparison CPU ⇔ GPU – Hardware Design



CPU

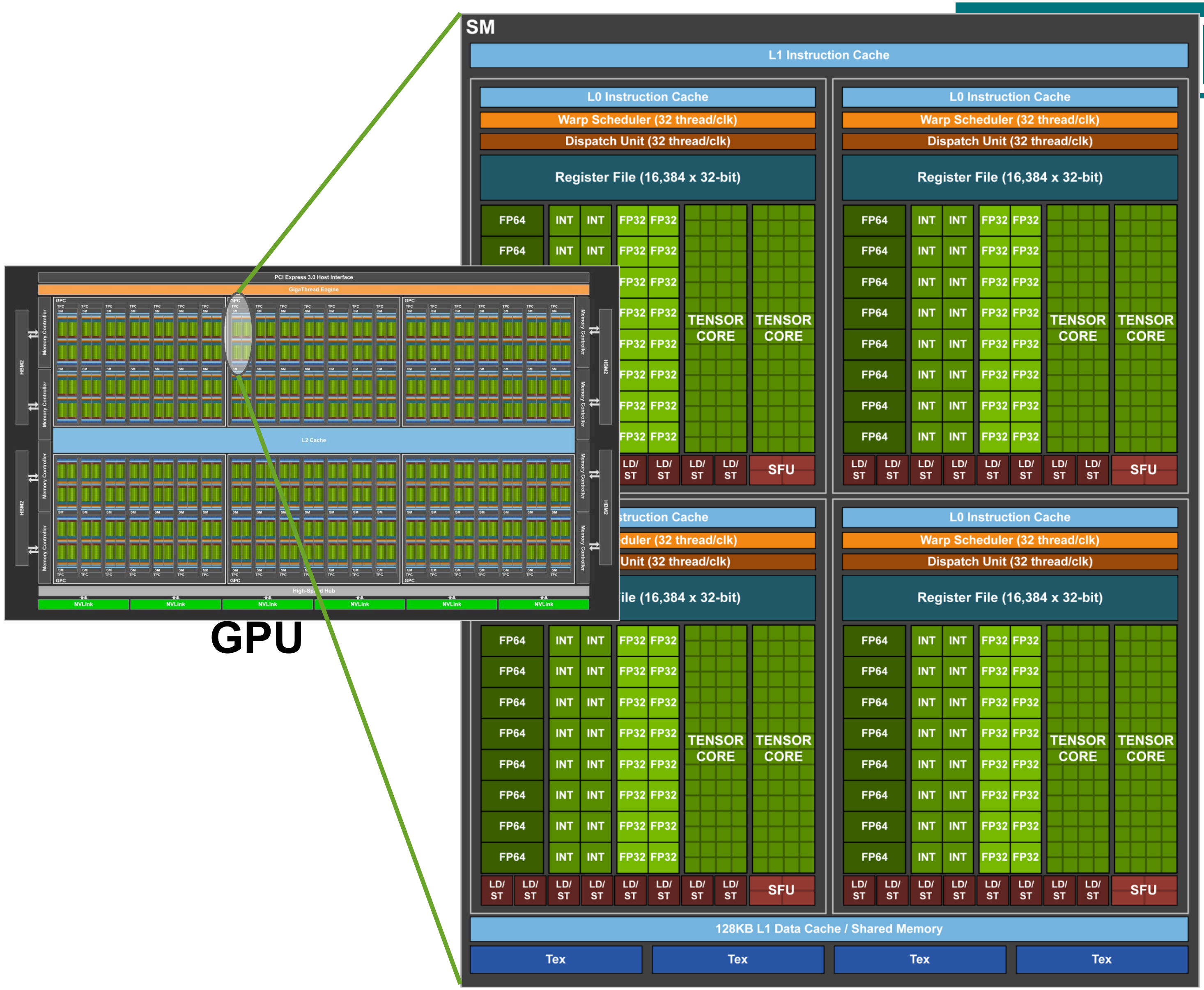
- Optimized for low latencies
- Huge caches
- Control logic for out-of-order and speculative execution
- **Targets on general-purpose applications**

GPU

- Optimized for data-parallel throughput
- Architecture tolerant of memory latency
- More transistors dedicated to computation
- **Suited for special kind of apps**

GPU architecture: Volta (V100)

- 21.1 billion transistors
- 80 streaming multiprocessors (SM)
  - Each: 64 (SP) cores, 32 (DP) cores, 8 Tensor cores
- Peak performance
  - SP: 15.7 Tflops
  - DP: 7.8 Tflops
  - Tensor: 125 Tflops
- 32 GB / 16 GB HBM2 memory
  - 900 GB/s bandwidth
- 300W thermal design power



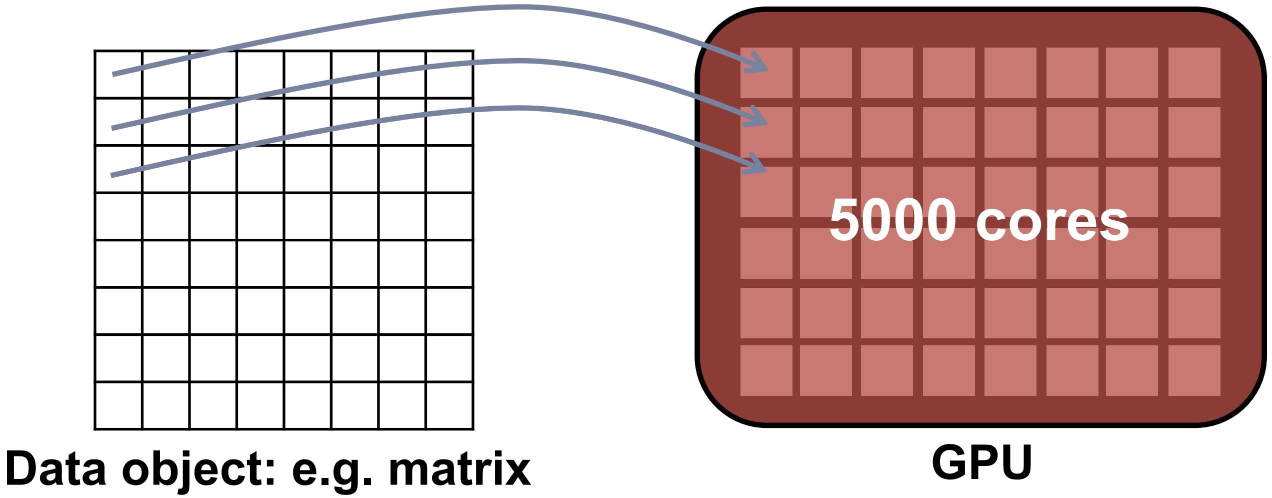
Source: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

(GPU) Offloading Concepts

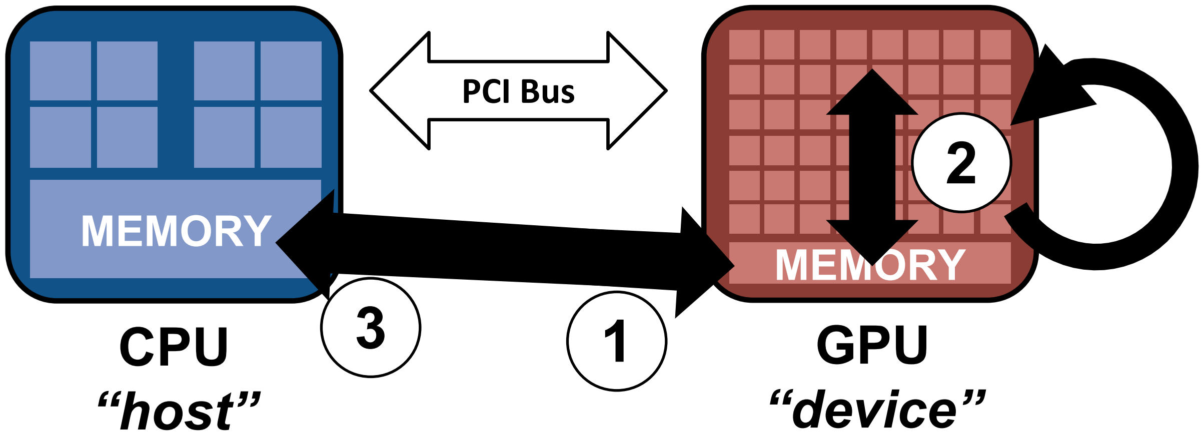
Data-Parallel Computing

- "If you were plowing a field, which would you rather use: Two strong oxen or 1024 chickens?“ Seymour Cray
  - Latency vs. throughput-oriented hardware

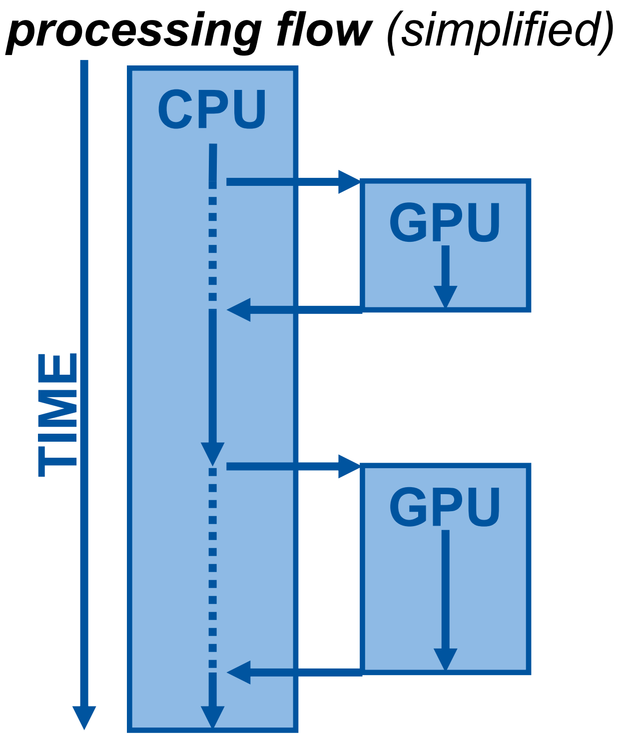
- GPU design goal: maximize throughput
  - A single thread is executed on each processing element simultaneously
  - Threads are logically organized like data



Offloading



We refer to "discrete GPUs" here.

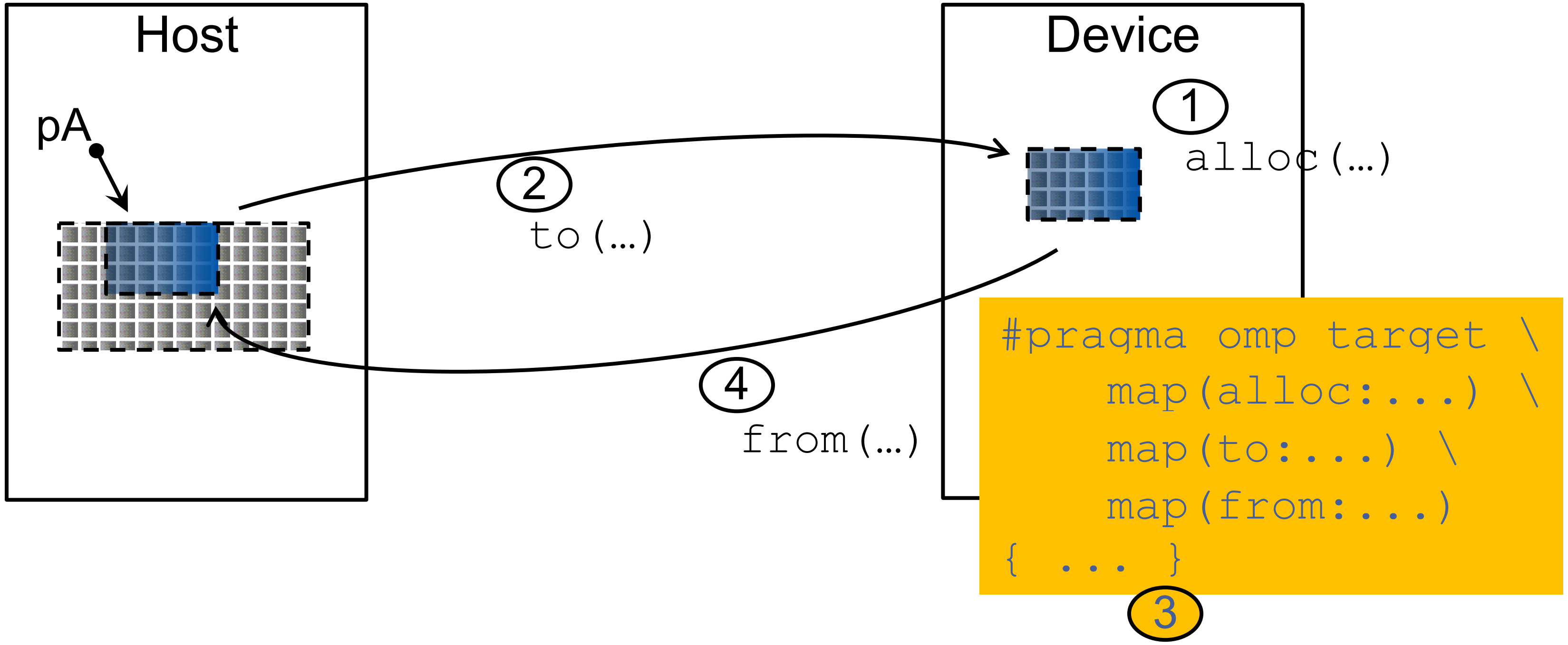


- Separate host and device memory
  - No coherence between host + device
    - Data transfers needed
- Host-directed execution model
  - Copy input data from CPU mem. to device mem.
  - Execute the device program
  - Copy results from device mem. to CPU mem.

Offloading in OpenMP

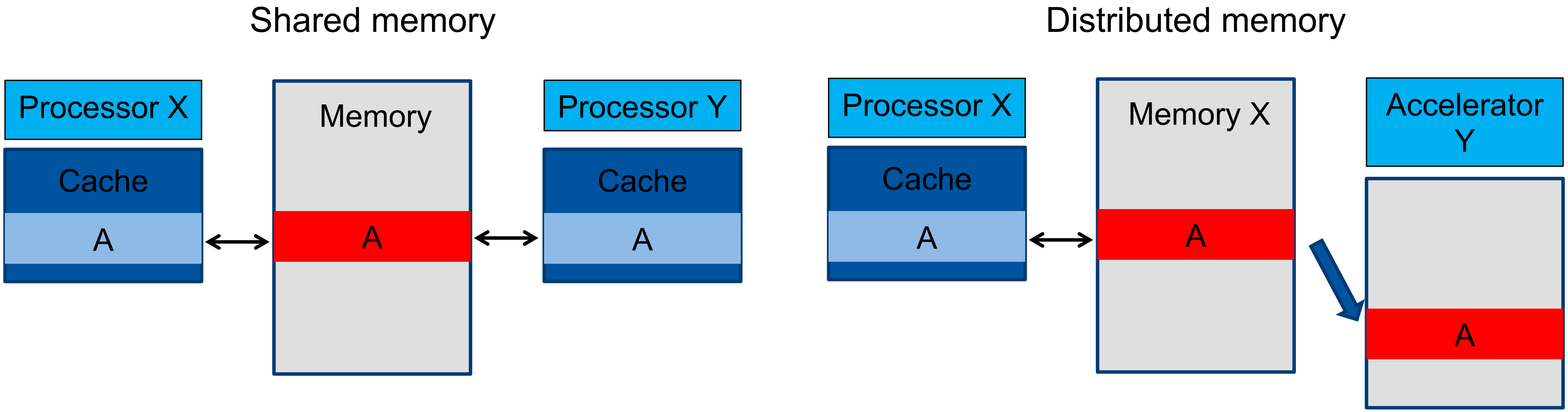
Device Data Environment

- The `map` clauses determine how an *original variable* in a data environment is mapped to a *corresponding variable* in a device data environment.



MAP is not necessarily a copy

- The corresponding variable in the device data environment may share storage with the original variable.
- Writes to the corresponding variable may alter the value of the original variable.



Data Management Directives

- Mapping Data (can directly be applied to **`target`** construct)
  - **`map(to:variable)`**: Copy input variable to device before executing the code region
  - **`map(from:variable)`**: Copy output variable from device after executing the code region
  - **`map(tofrom:variable)`**: Copy variable to device before executing the code region and copy variable back to the host after executing the code region
  - **`map(alloc:variable)`**: Allocate uninitialized variable on the device

- Construct: **`target data`**
  - maps data to device without offloading code
  - Useful for defining large areas of code that share device data
  - Helps reduce the required data transfers

- Construct: **`target update`**
  - Updates data on the device from the host

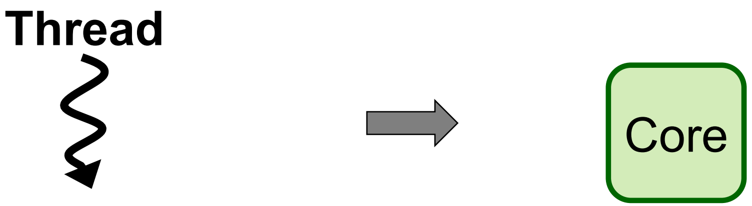
Example: DAXPY

Example DAXPY: Data Management

```
void daxpy(int n, double a, double *x, double *y) {
  #pragma omp target map(tofrom:y[0:n]) map(to:a,x[0:n])
  for (int i = 0; i < n; ++i)
    y[i] = a * x[i] + y[i];
}

int main(int argc, const char* argv[]) {
  static int n = 100000000; static double a = 2.0;
  double *x = (double *) malloc(n * sizeof(double));
  double *y = (double *) malloc(n * sizeof(double));

  // Initialize x, y
  for(int i = 0; i < n; ++i){
    x[i] = 1.0;
    y[i] = 2.0;
  }
  daxpy(n, a, x, y); // Invoke daxpy kernel
  // Check if all values are 4.0

  free(x); free(y);
  return 0;
}
```

Output:
$ $CC $FLAGS_OFFLOAD_OPENMP daxpy.c
$ a.out
Max error: 0.00000
Total runtime: 102.50s

Mapping to Hardware



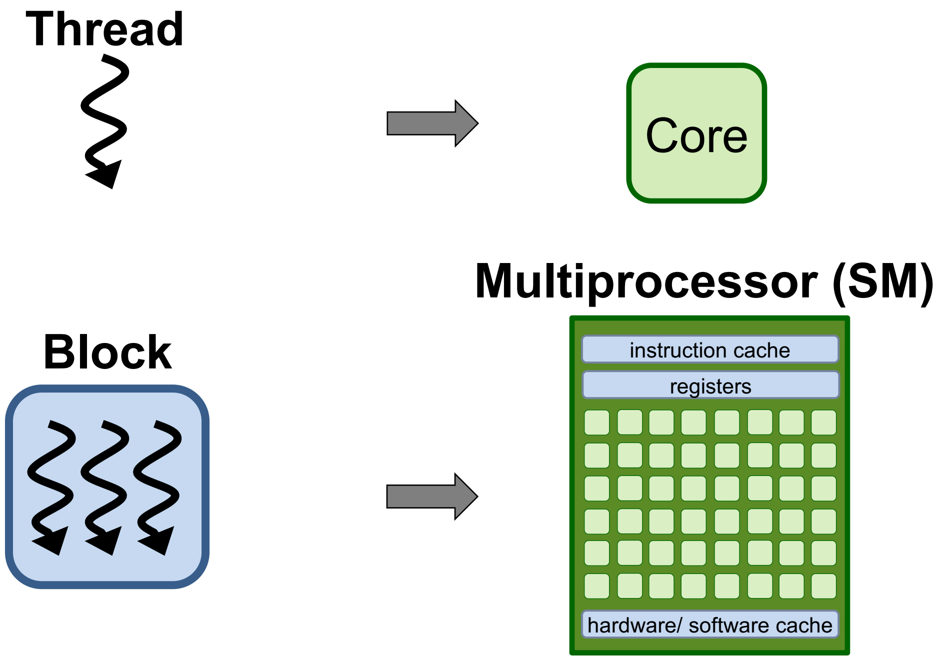
- Each thread is executed by a core

Example DAXPY: Thread Parallelism

```
void daxpy(int n, double a, double *x, double *y) {
  #pragma omp target parallel for map(tofrom:y[0:n]) map(to:a,x[0:n])
  for (int i = 0; i < n; ++i)
    y[i] = a * x[i] + y[i];
}

int main(int argc, const char* argv[]) {
  static int n = 100000000; static double a = 2.0;
  double *x = (double *) malloc(n * sizeof(double));
  double *y = (double *) malloc(n * sizeof(double));

  // Initialize x, y
  for(int i = 0; i < n; ++i){
    x[i] = 1.0;
    y[i] = 2.0;
  }
  daxpy(n, a, x, y); // Invoke daxpy kernel
  // Check if all values are 4.0

  free(x); free(y);
  return 0;
}
```

Output:
$ $CC $FLAGS_OFFLOAD_OPENMP daxpy.c
$ a.out
Max error: 0.00000
Total runtime: 9.65s

Mapping to Hardware



- Each thread is executed by a core

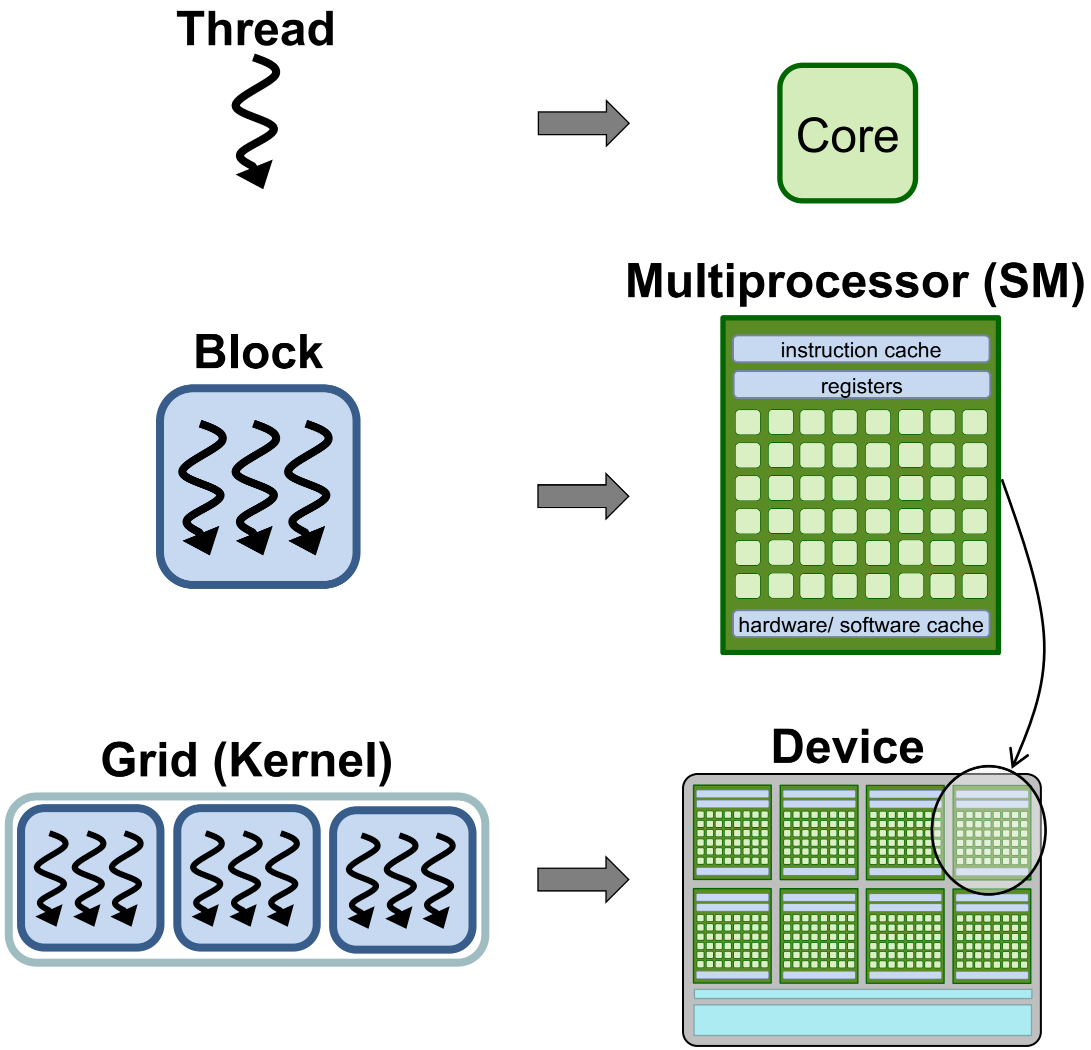
- Each block is executed on a SM
- Several concurrent blocks can reside on a SM depending on shared resources

Example DAXPY: Thread Parallelism

```
void daxpy(int n, double a, double *x, double *y) {
  #pragma omp target teams distribute parallel for map(tofrom:y[0:n]) map(to:a,x[0:n])
for (int i = 0; i < n; ++i)
    y[i] = a * x[i] + y[i];
}

int main(int argc, const char* argv[]) {
  static int n = 100000000; static double a = 2.0;
  double *x = (double *) malloc(n * sizeof(double));
  double *y = (double *) malloc(n * sizeof(double));

  // Initialize x, y
  for(int i = 0; i < n; ++i){
    x[i] = 1.0;
    y[i] = 2.0;
  }
  daxpy(n, a, x, y); // Invoke daxpy kernel
  // Check if all values are 4.0

  free(x); free(y);
  return 0;
}
```

```
Output:
$ $CC $FLAGS_OFFLOAD_OPENMP  daxpy.c
$ a.out
Max error: 0.00000
Total runtime: 0.80s
```

Mapping to Hardware



- Each thread is executed by a core
- Each block is executed on a SM
- Several concurrent blocks can reside on a SM depending on shared resources
- Each kernel is executed on a device