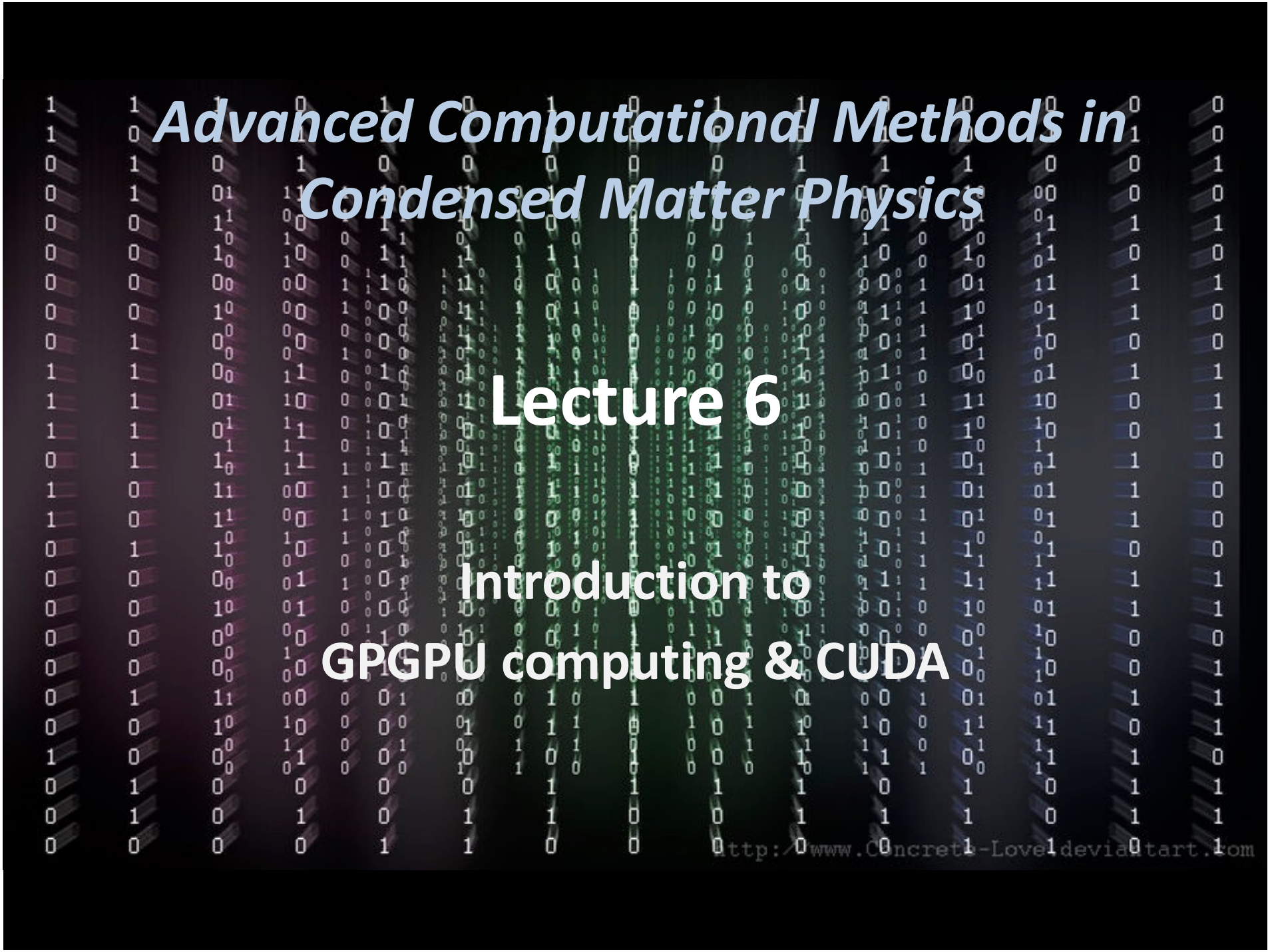

Advanced Computational Methods in Condensed Matter Physics
Lecture 6
Introduction to
GPGPU computing & CUDA

Terminology

- What is GPGPU?
  - General-Purpose computing on a Graphics Processing Unit
  - Using graphic hardware for non-graphic computations
- What is CUDA?
  - Compute Unified Device Architecture
  - Software architecture for managing data-parallel programming

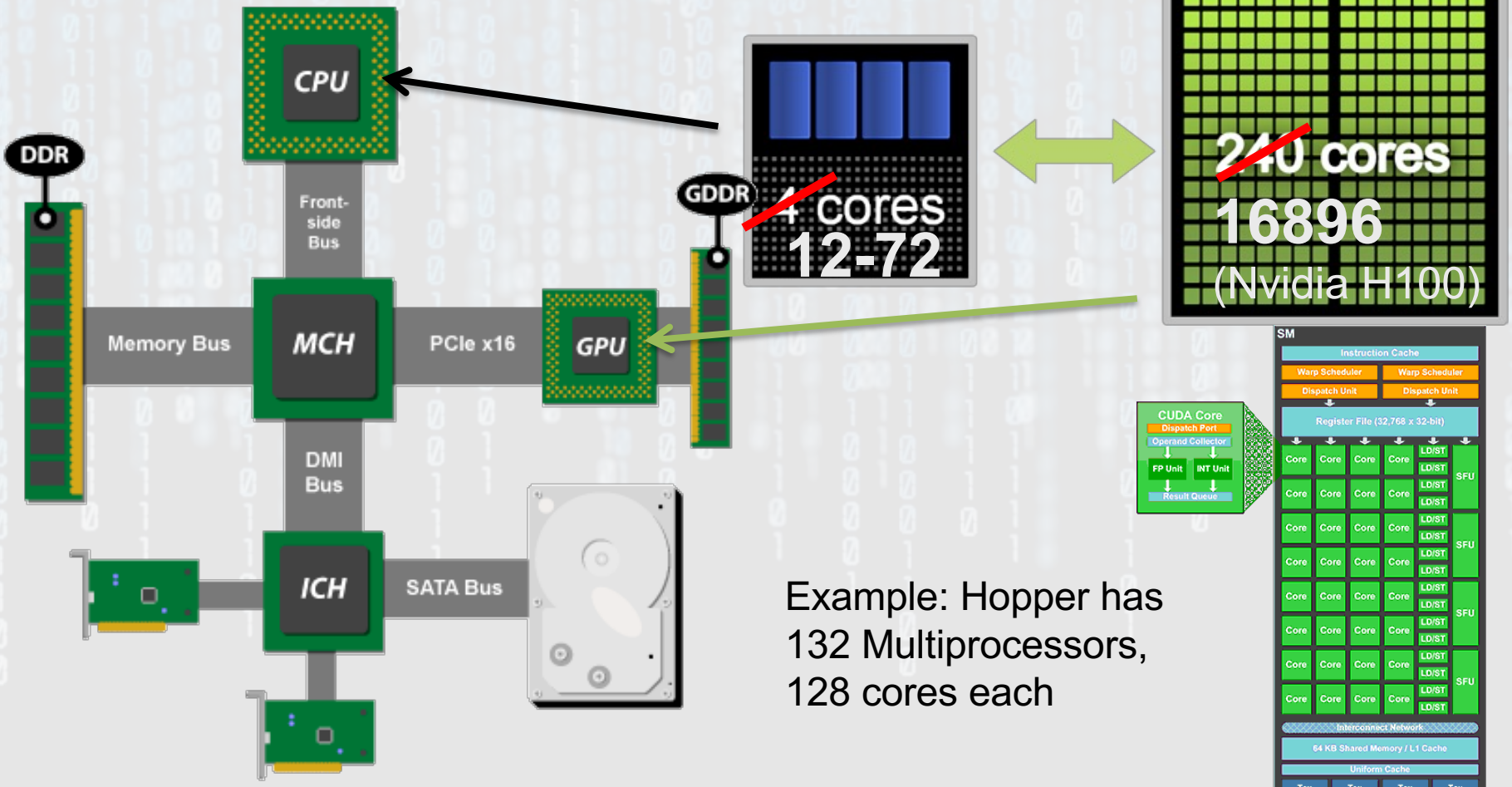
GPU computing

Scientific computing on General Purpose Graphics Processing Units (GPUs): a new paradigm

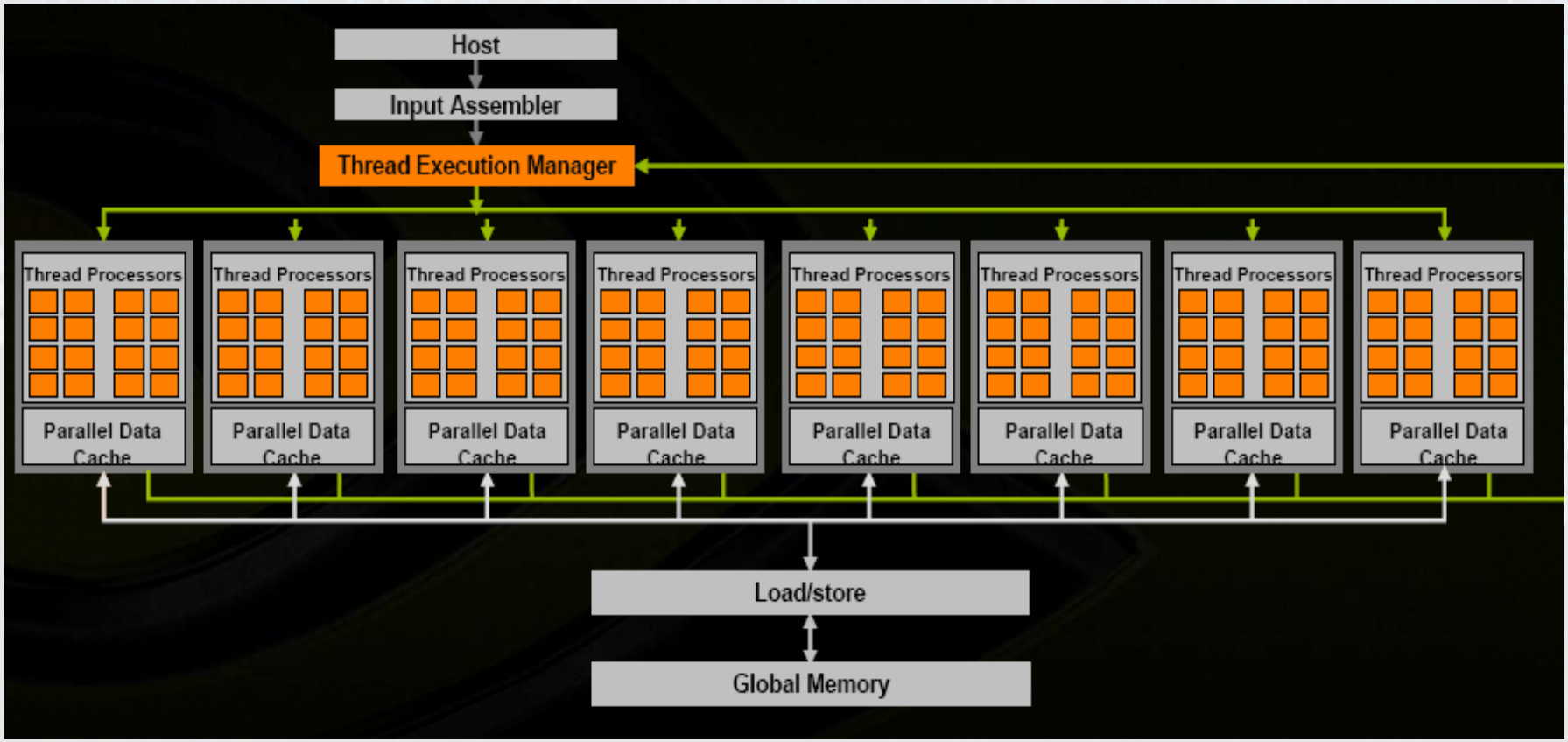
- Massive parallel computing on one chip
- Shared memory, no data latency
- Easy to use development toolkit: CUDA or OpenCL
- Minimal changes to existing codes needed
- Low power consumption (up to a factor 30 less than comparable CPU system)
- Low cost per performance (up to 50x improvement)

- speed-up of our simulations: **typically, more than *100x***

GPU architecture

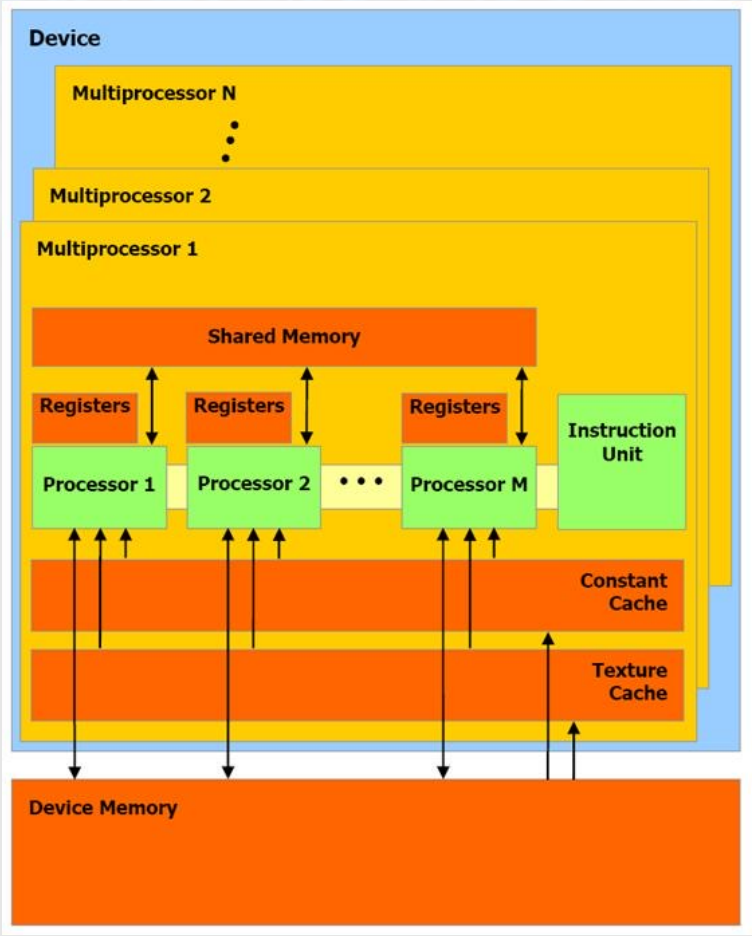


Example: Hopper has
132 Multiprocessors,
128 cores each


Host
Input Assembler
Thread Execution Manager
Thread Processors
Parallel Data Cache
Load/store
Global Memory

Memory architecture

- Constant Memory
- Texture Memory
- Device Memory



Bill Dally (Nvidia) in “Life After Moore's Law”:

“We have reached the limit of what is possible with one or more traditional, serial central processing units, or CPUs. It is past time for the computing industry - and everyone who relies on it for continued improvements in productivity, economic growth and social progress - to take the leap into parallel processing.”… “the CPU scaling predicted by Moore's Law is now dead.”

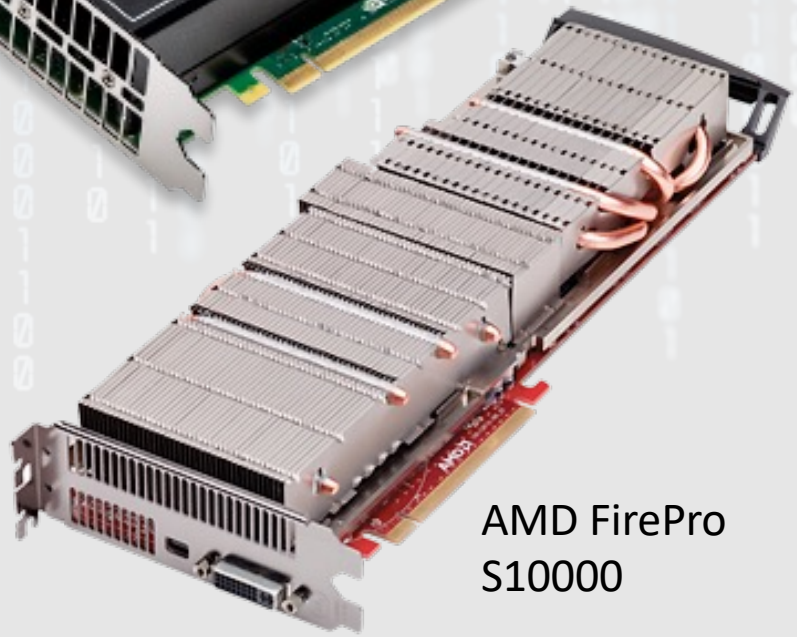
Performance

Comparison of single node performances:

- board with 8 Nvidia M2090: **10.48 TFlops**
- 8x AMD HD 6990: **10.16 Tflops**
- board with 8 Intel Knights Corner: **7.4 TFlops** (prototype)

- Blue Gene/P node card: **435 GFlops**

Nvidia Tesla K40



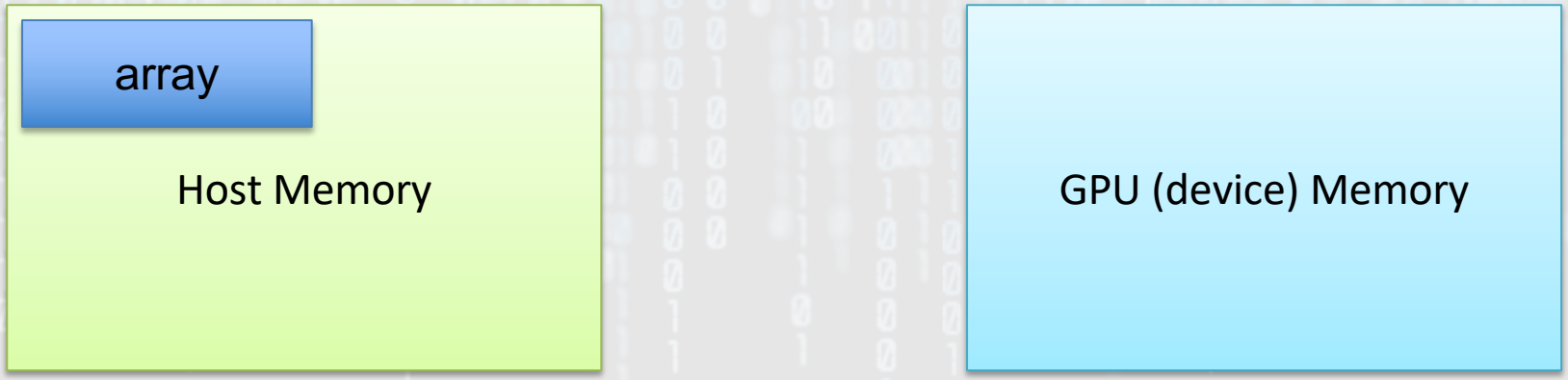
AMD FirePro S10000

A GPU is a specialized computer

- We need to allocate space in the video card's memory for the variables.
- The video card does not have I/O devices, hence we need to copy the input data from the memory in the host computer into the memory in the video card, using the variable allocated in the previous step.
- We need to specify code to execute.
- Copy the results back to the memory in the host computer.

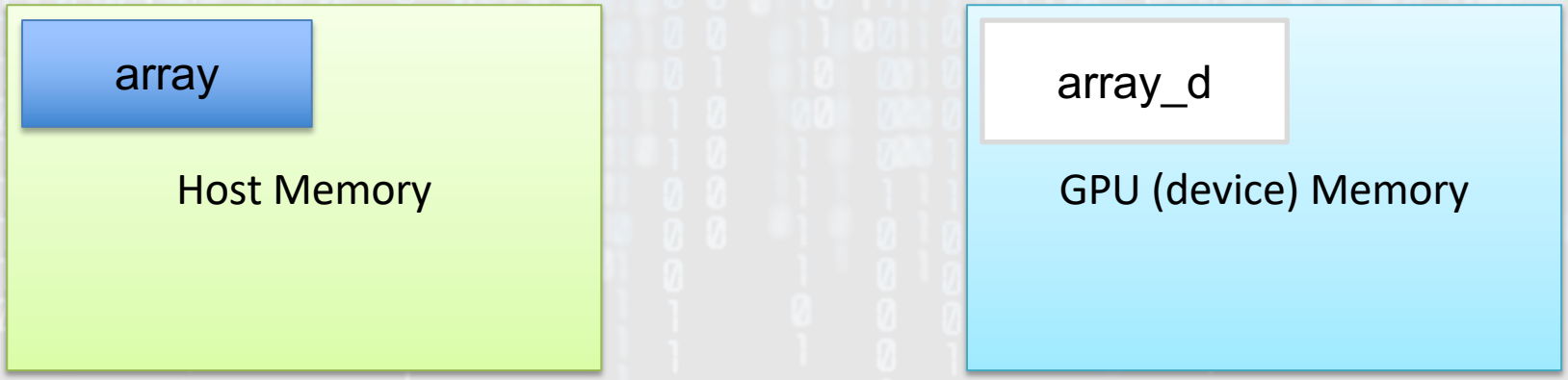
A typical scenario

Initial state



A typical scenario

Memory allocation on the GPU

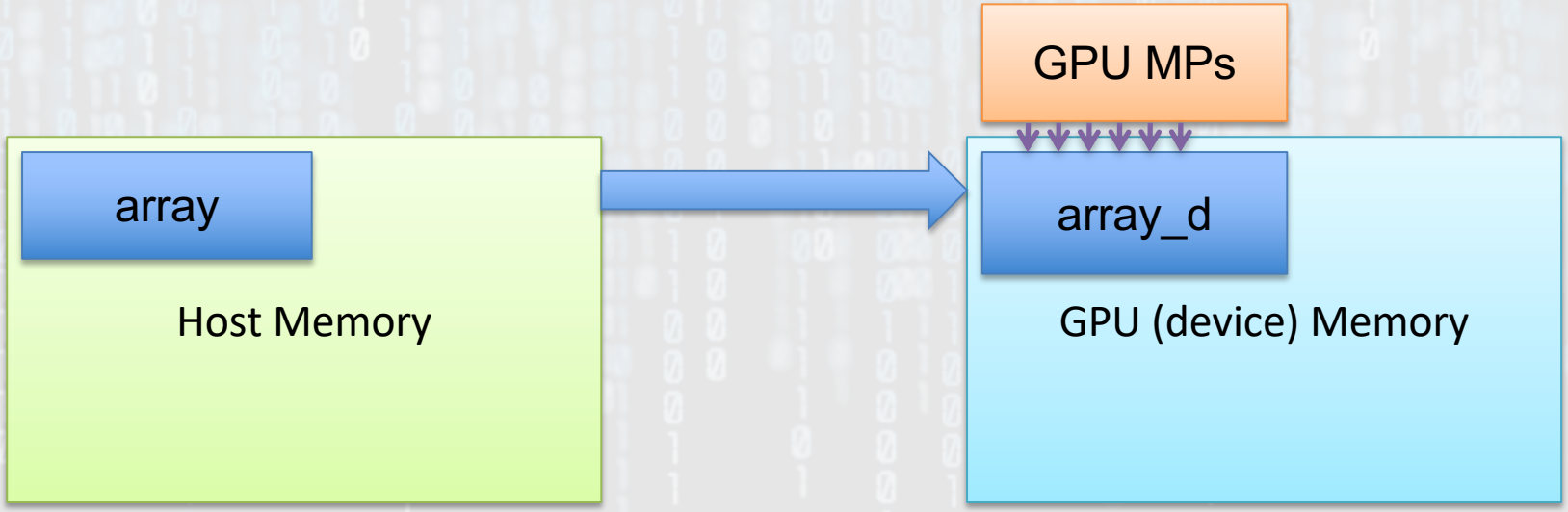


A typical scenario

Memory allocation on the GPU

A typical scenario

Execute code on the GPU



A typical scenario

Copy results back to the host memory

Introduction to CUDA

- *CUDA* – Compute Unified Device Architecture
- Data parallel problems
- Great for 1D, 2D, 3D regular grids
- Not true SIMD
  - SPMT – Single Program Multiple Threads
  - SIMT – Single Instruction Multiple Threads

CUDA Programming Model: A Highly Multithreaded Coprocessor

- The GPU is viewed as a compute device that:
  - Is a coprocessor to the CPU or host
  - Has its own DRAM (device memory)
  - Runs many threads in parallel
- Data-parallel portions of an application are executed on the device as kernels which run in parallel on many threads
- Differences between GPU and CPU threads
  - GPU threads are extremely lightweight
    - Very little creation overhead
  - GPU needs 1000s of threads for full efficiency
    - Multi-core CPU needs only a few

Getting started

- *Host* – CPU, main memory
- *Device* – GPU, GPU memory

- CUDA is like C/C++ with extensions
- *Kernel* – GPU function written in CUDA
- Source code placed in .cu files
- Nvidia compiler – **nvcc** – must be used
  - Link in libraries **–lcudart, -lcublas, -lcufft**

nvcc: Partition code and compile for device

mycode.cu

```
int main_data;
__shared__  int sdata;

Main() { }
__host__  hfunc () {
  int hdata;
<<<gfunc(g,b,m)>>>();
}

__global__ gfunc() {
  int gdata;
}

__device__ dfunc() {
  int ddata;
}
```

Host Only

Interface

Compiled by native compiler: gcc, icc, cc

```
int main_data;

Main() {}
__host__  hfunc () {
  int hdata;
<<<gfunc(g,b,m)>>>();
}
```

Compiled by nvcc compiler

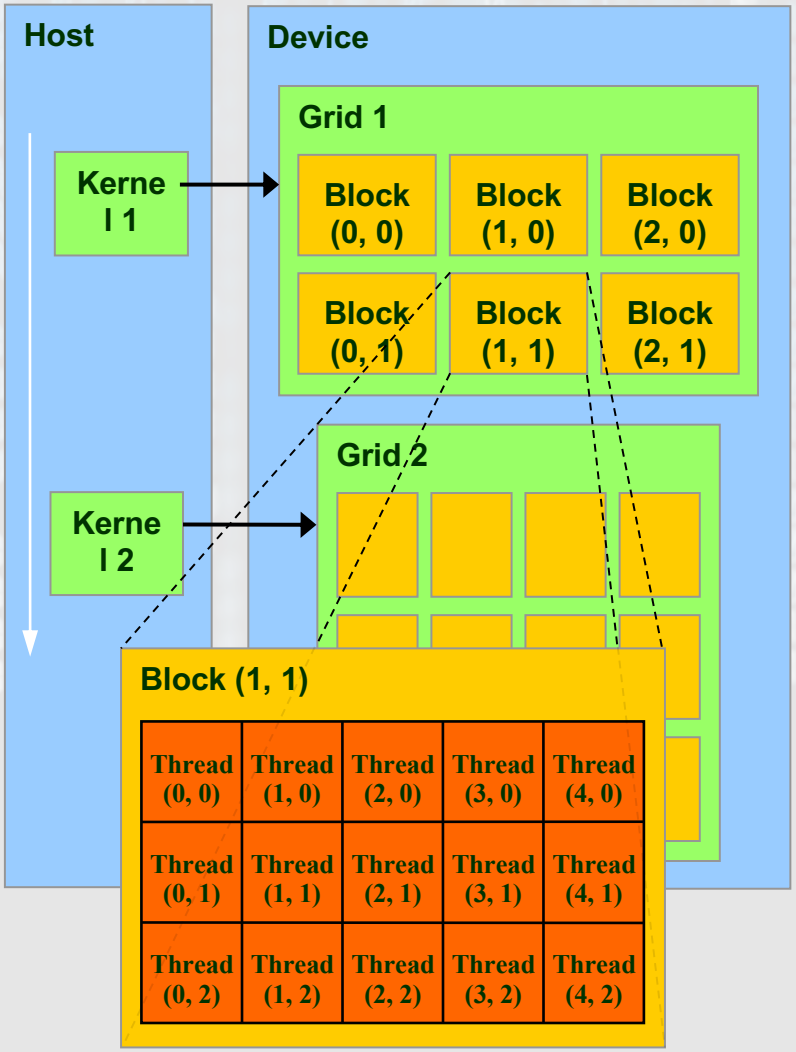
```
__shared__ sdata;

__global__ gfunc() {
  int gdata;
}

__device__ dfunc() {
  int ddata;
}
```

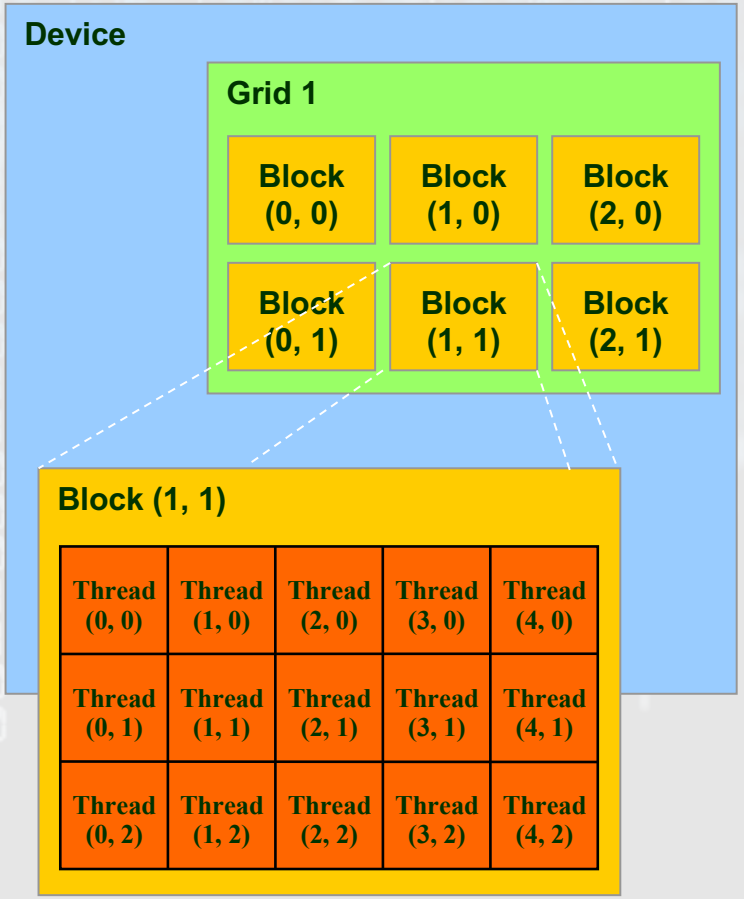
Threat batching: grids and blocks

- A kernel is executed as a grid of thread blocks
  - All threads share data memory space
- A thread block is a batch of threads that can cooperate with each other by:
  - Synchronizing their execution
    - For hazard-free shared memory accesses
  - Efficiently sharing data through a low latency shared memory
- Two threads from two different blocks cannot cooperate



Block and Thread IDs

- Threads and blocks have IDs
  - So each thread can decide what data to work on
  - Block ID: 1D or 2D (blockIdx.x, blockIdx.y)
  - Thread ID: 1D, 2D, or 3D (threadIdx.{x,y,z})
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …



CUDA functions

- **cudaError_t** returned by most functions
  - hopefully: **cudaSuccess**
- camelCase function names

Error checking:

```
bool cuErr(const char *s="n/a")
{
 cudaError_t err=cudaGetLastError();
 if(err==cudaSuccess)
  return false;
 printf("CUDA error [%s]: %s\n",s,cudaGetErrorString(err));
 return true;
}
```

Device Information

- **cudaGetDeviceCount(int * device_count)**
- Return values
  - **cudaErrorNoDevice**
  - **cudaErrorInsufficientDriver**
- **cudaGetDeviceProperties(cudaDeviceProp *, int device)**
  - **cudaDeviceProp** – holds device information

Device Management

- **cudaGetDevice(int * device)**
- **cudaSetDevice(int device)**
  - All memory management, kernels, will be on this device

Memory Management

- **cudaMalloc(void ** devPtr, size_t nbytes)**
  - Pass in a memory pointer which is altered
  - **CudaErrorMemoryAllocation** returned if malloc fails
- **cudaFree(void * devPtr)**
  - **cudaErrorInvalidDevicePointer** returned on failure
- **cudaMemcpy(void * dst, const void * src, size_t count, enum cudaMemcpyKind kind)**
  - **cudaMemcpyHostToHost**
  - **cudaMemcpyHostToDevice**
  - **cudaMemcpyDeviceToHost**
  - **CudaMemcpyDeviceToDevice**

CUDA Kernel functions

CUDA Example

```
1.  #include <stdio.h>
2.  #include <cuda.h>
3.
4.  // Kernel that executes on the CUDA device
5.  __global__ void square_array(float *a, int N)
6.  {
7.    int idx = blockIdx.x * blockDim.x + threadIdx.x;
8.    if (idx<N) a[idx] = a[idx] * a[idx];
9.  }
10.
11. // main routine that executes on the host
12. int main(void)
13. {float *a_h, *a_d;  // Pointer to host & device arrays
14.   const int N = 10;  // Number of elements in arrays
15.   size_t size = N * sizeof(float);
16.   a_h = (float *)malloc(size);       // Allocate array on host
17.   cudaMalloc((void **) &a_d, size);   // Allocate array on device
18.   // Initialize host array and copy it to CUDA device
19.   for (int i=0; i<N; i++) a_h[i] = (float)i;
20.   cudaMemcpy(a_d, a_h, size, cudaMemcpyHostToDevice);
21.   // Do calculation on device:
22.   int block_size = 4;
23.   int n_blocks = N/block_size + (N%block_size == 0 ? 0:1);
24.   square_array <<< n_blocks, block_size >>> (a_d, N);
25.   // Retrieve result from device and store it in host array
26.   cudaMemcpy(a_h, a_d, sizeof(float)*N, cudaMemcpyDeviceToHost);
27.   // Print results
28.   for (int i=0; i<N; i++) printf("%d %f\n", i, a_h[i]);
29.   // Cleanup
30.   free(a_h); cudaFree(a_d);}
```

Built-in variables in kernel function

- **threadIdx** – index number of thread within thread block
- **blockIdx** – index number of block within grid
- **blockDim** – size of block dimensions (threads)
- **gridDim** – size of grid dimensions (blocks)
- Each variable has x, y, and z fields.
  - For 1D partitions use .x
  - For 2D partitions use .x and .y
  - For 3D partitions use .x, .y, and .z

Computing limitations

- Compute capabilities (hardware)

1.0	1.1	1.2	1.3	2.0	2.1	3.0	3.5
G80			G200	Fermi		Kepler	

- Different hardware has different capabilities
- Max threads per block (for single precision)

1.0	1.1	1.2	1.3	2.0	2.1	3.0	3.5
512				1024			

- The maximum dimensions of each block are limited to [512,512,64]/[1024,1024,64] (Compute 1.x/2.x)
- Each block cannot consume more than 8k/16k/32k registers total (Compute 1.0,1.1/1.2,1.3/2.x)
- Each block cannot consume more than 16kb/48kb of shared memory (Compute 1.x/2.x)
- For large 1D arrays, e.g.:

```
int idx=(blockIdx.y*gridDim.x+blockIdx.x)*blockDim.x+threadIdx.x;
```

Newer compute capabilities

- Max threads per block 1024 on newer hardware
- Block size very large: up to 2^{31} per direction
- Compute capabilities of some NV GPUs:

12.1
NVIDIA GB10 (DGX Spark)

12.0
NVIDIA RTX PRO x000 Blackwell
GeForce RTX 50xx

10.3
NVIDIA GB300
NVIDIA B300

10.0
NVIDIA GB200
NVIDIA B200

9.0
NVIDIA GH200
NVIDIA H200
NVIDIA H100

8.9
NVIDIA L4
NVIDIA L40
NVIDIA L40S
NVIDIA RTX x000 Ada
GeForce RTX 40xx

8.6
NVIDIA Axx
NVIDIA RTX Ax000
GeForce RTX 30xx

8.0
NVIDIA A100
NVIDIA A30

7.5
NVIDIA T4
QUADRO RTX x000
GeForce RTX 20xx

Further Reading

- Nvidia:
  - https://developer.nvidia.com/cuda-zone
  - http://docs.nvidia.com/cuda
  - http://developer.download.nvidia.com/books/cuda-by-example/cuda-by-example-sample.pdf (excerpt, see amazon or google for a pdf)
- David Kirk and Wen-mei Hwu manuscript
  - http://www.toodoc.com/CUDA-textbook-by-David-Kirk-from-NVIDIA-and-Prof-Wen-mei-Hwu-pdf.html
  - Book: “Programming Massively Parallel Processors”
- Nice series from Dr. Dobbs Journal by Rob Farber
  - http://www.drdobbs.com/parallel/cuda-supercomputing-for-the-masses-part/207200659

- http://www.youtube.com/watch?v=ZrJeYFxpUyQ