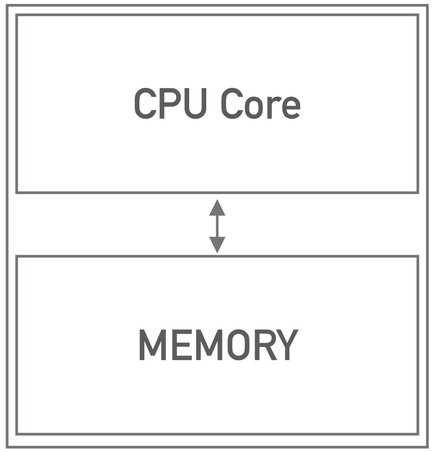
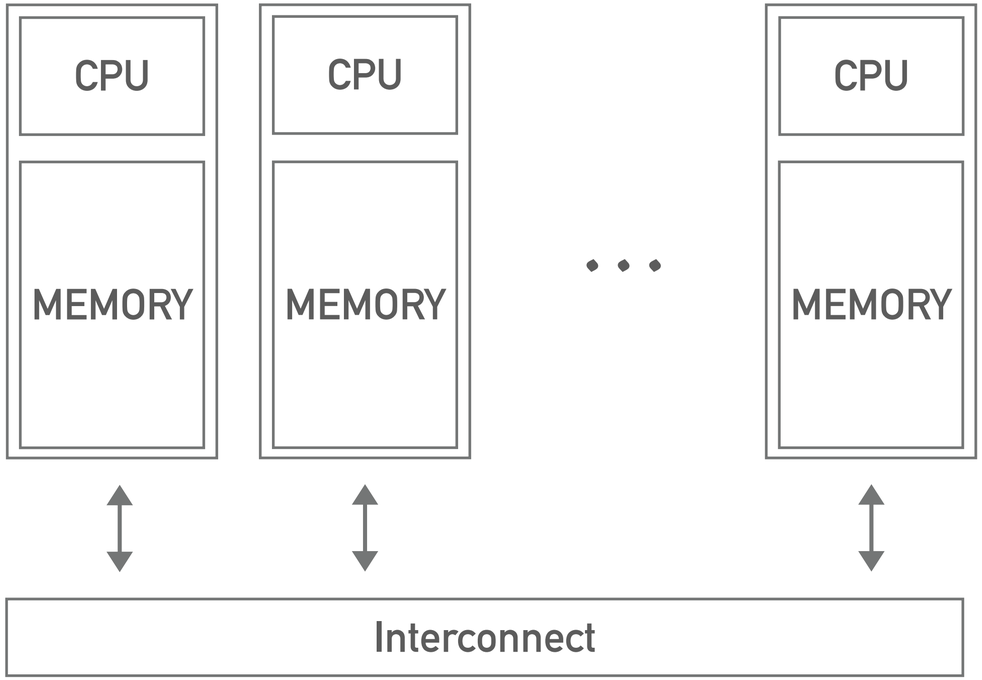
ME 471/571

Introduction to CUDA C

TOWARDS HETEROGENOUS COMPUTING



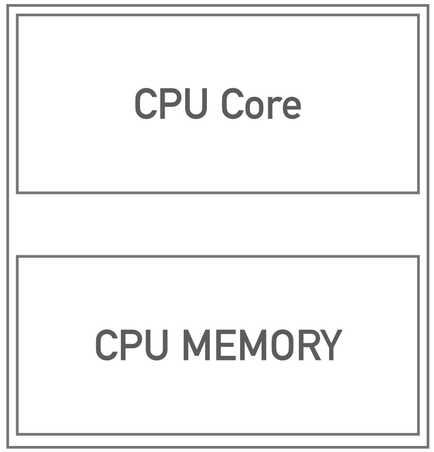
SERIAL COMPUTING



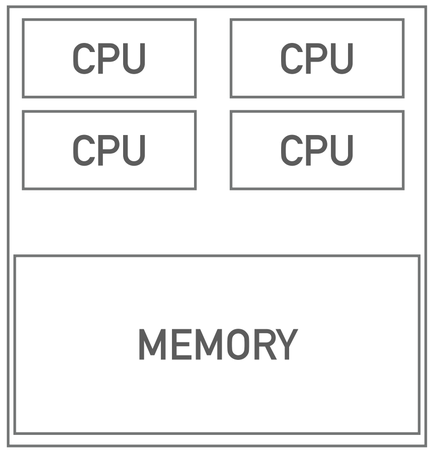
HOMOGENEOUS
PARALLEL COMPUTING

TOWARDS HETEROGENOUS COMPUTING

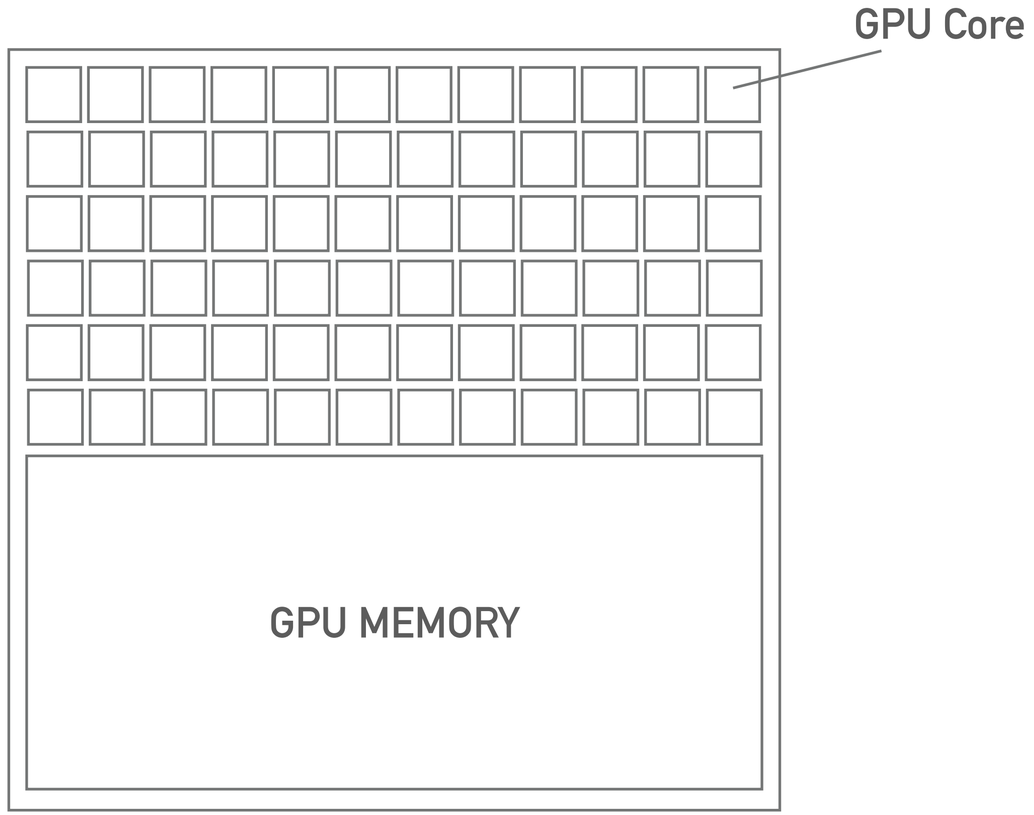
SINGLE CORE (CPU)



MULTI CORE (CPU)



MANY CORE (GPU)



TOWARDS HETEROGENOUS COMPUTING

CPU

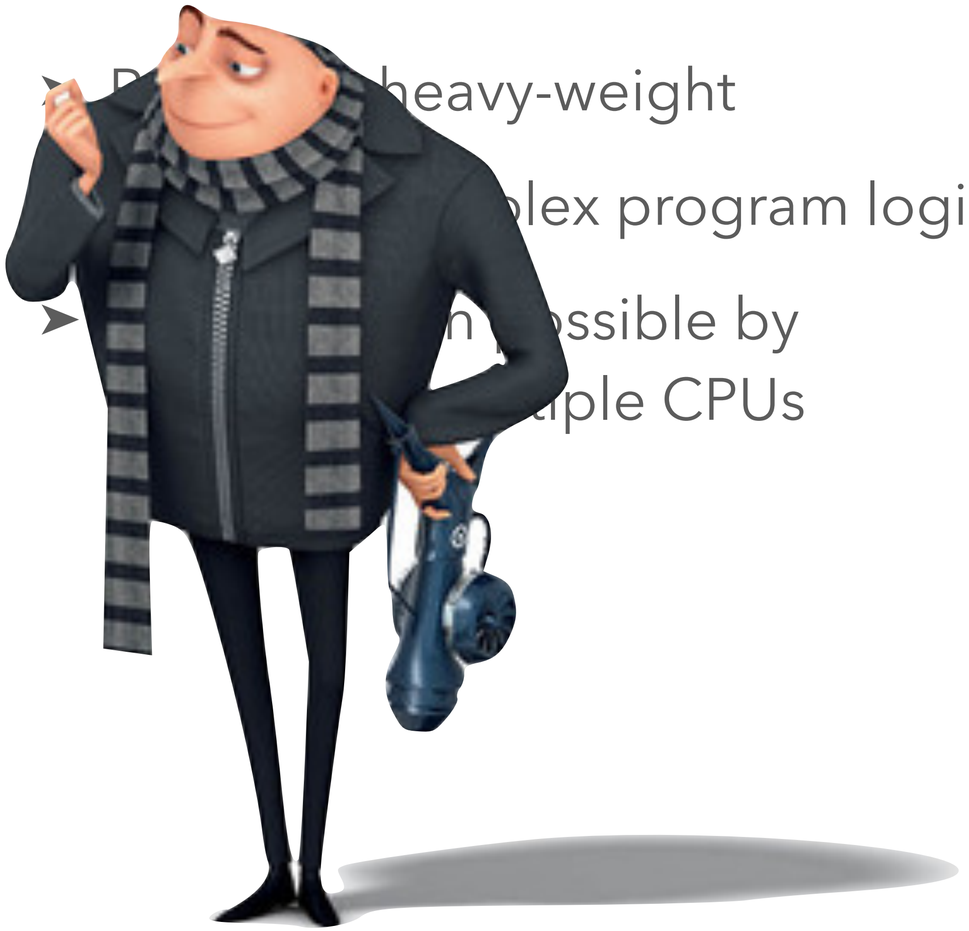
- Relatively heavy-weight
- Good for complex program logic
- Parallelization possible by connecting multiple CPUs

ACCELERATOR (GPU)

- One core is light-weight
- Simple program logic
- Built-in massive shared data parallelism

TOWARDS HETEROGENOUS COMPUTING

CPU



- R… heavy-weight
- …plex program logic
- …ssible by …iple CPUs

ACCELERATOR (GPU)

- One core is light-weight
- Simple program logic
- Built-in massive shared data parallelism

R2 COMPUTING CAPABILITY

DEVICE	# cores	PEAK PERFORMANCE
1 CPU core	1	~40 GFlops
1 CPU (Intel Xeon E5-2680)	14	~570 GFlops

R2 COMPUTING CAPABILITY

DEVICE	# cores	PEAK PERFORMANCE
1 CPU core	1	~40 GFlops
1 CPU (Intel Xeon E5-2680)	14	~570 GFlops
1 compute node (2x14 cores)	28	~1.1 TFlops
All 31 compute nodes (868 cores)	868	~35.4 TFlops

R2 COMPUTING CAPABILITY

DEVICE	# cores	PEAK PERFORMANCE
1 CPU core	1	~40 GFlops
1 CPU (Intel Xeon E5-2680)	14	~570 GFlops
1 compute node (2x14 cores)	28	~1.1 TFlops
All 31 compute nodes (868 cores)	868	~35.4 TFlops
1 GPU device (NVIDIA Tesla P100)	3584	~10.6 TFlops

R2 COMPUTING CAPABILITY

DEVICE	# cores	PEAK PERFORMANCE
1 CPU core	1	~40 GFlops
1 CPU (Intel Xeon E5-2680)	14	~570 GFlops
1 compute node (2x14 cores)	28	~1.1 TFlops
All 31 compute nodes (868 cores)	868	~35.4 TFlops
1 GPU device (NVIDIA Tesla P100)	3584	~10.6 TFlops
All 5 GPU nodes (2 GPUs per node)	35840	~106 TFlops

Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442,010.0	537,212.0	29,899
2	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148,600.0	200,794.9	10,096
3	**Sierra** - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94,640.0	125,712.0	7,438
4	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway, NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371
5	**Selene** - NVIDIA DGX A100, AMD EPYC 7742 64C 2.25GHz, NVIDIA A100, Mellanox HDR Infiniband, Nvidia NVIDIA Corporation United States	555,520	63,460.0	79,215.0	2,646

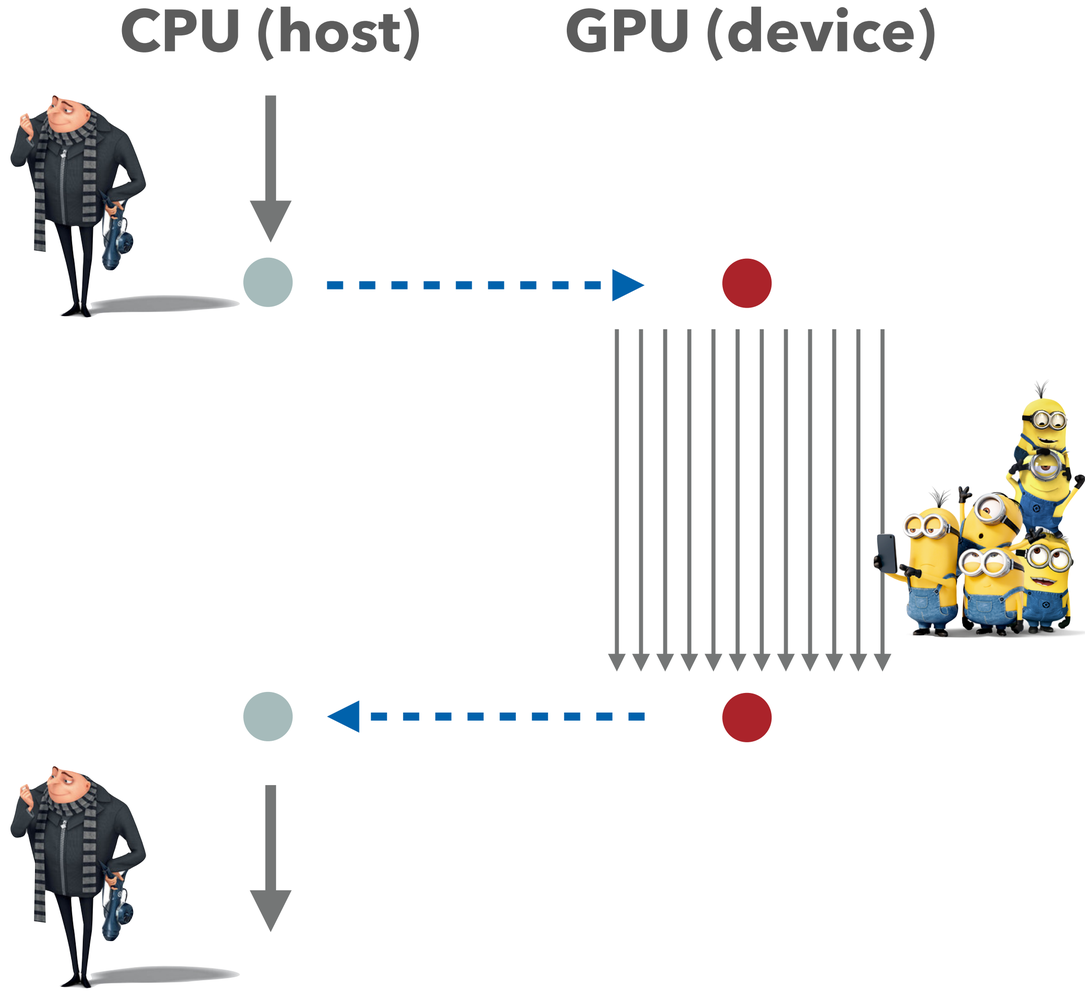
HETEROGENOUS COMPUTING

CUDA C

- based on standard C/C++
- small set of extensions to enable heterogeneous programming
- GPU programs are called **kernels**
- **kernels** are executed in parallel by threads (roughly corresponding to ranks in MPI)
- CUDA allows for
  - writing device kernels
  - moving data between device and host
  - organizing and synchronizing threads

SIMPLE WORKFLOW IN CUDA C

- host executes serial part of the program
- copy data from host to device
- multiple threads execute kernel on parts of data
- copy result from device to host
- host continues the execution of serial program



CUDA ON R2

`module avail`	- *list available modules*
`module load gcc/7.2.0`	- *load gcc compiler compatible with CUDA*
`module load cuda10.0`	- *load CUDA toolkit*
`module unload <module_name>`	- *unload <module_name> if needed*
`gpu-session`	- *get an interactive session on a GPU node*
`#BATCH -p gpuq`	- *put this in the run script to get gpu node access*
`nvcc`	- *CUDA compiler (instead of gcc or mpicc)*

HELLO WORLD!

*this kernel is **global**:*

- *runs on the **device***
- *can be called from **host***

*execute **kernel***

on 1 block of 10 threads

reset device

```
#include <stdio.h>


__global__ void helloFromGPU(void) {
  printf("Hello World from GPU);
}


int main(void) {
  printf("Hello World from CPU!\n");
  helloFromGPU<<<1, 10>>>();
  cudaDeviceReset();
  return 0;
}
```

EXAMPLE 1 - ADDITION

Compute c = a+b

1) Allocate space on GPU for a, b, c
2) Copy values of a, b from host to device
3) Launch kernel to add numbers
4) Copy value of c from device to host
5) Free GPU memory

EXAMPLE 1 - ADDITION

Compute c = a+b

1) Allocate space on GPU for a, b, c
2) Copy values of a, b from host to device
3) Launch kernel to add numbers
4) Copy value of c from device to host
5) Free GPU memory

cudaMalloc

```
cudaMalloc((float**) &d_a, size)
```

pointer to device copy of a

size in bytes

EXAMPLE 1 - ADDITION

Compute c = a+b

1) Allocate space on GPU for a, b, c — `cudaMalloc`
2) **Copy values of a, b from host to device** — **`cudaMemcpy`**
3) Launch kernel to add numbers
4) Copy value of c from device to host
5) Free GPU memory

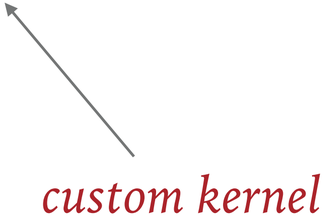
```
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice)
```

device copy of a — *host copy of a* — *size in bytes* — *direction of copy*

EXAMPLE 1 - ADDITION

Compute c = a+b

1) Allocate space on GPU for a, b, c — `cudaMalloc`
2) Copy values of a, b from host to device — `cudaMemcpy`
3) **Launch kernel to add numbers** — **`add_on_GPU`**
4) Copy value of c from device to host
5) Free GPU memory




```
__global__ add_on_GPU(float *a, float *b, float *c){
    *c = *a + *b;
    }
```

EXAMPLE 1 - ADDITION

Compute c = a+b

1) Allocate space on GPU for a, b, c — `cudaMalloc`
2) Copy values of a, b from host to device — `cudaMemcpy`
3) Launch kernel to add numbers — `add_on_GPU`
4) **Copy value of c from device to host** — **`cudaMemcpy`**
5) Free GPU memory

```
cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost)
```

host address of c — *device copy of c* — *size in bytes* — *direction of copy*

EXAMPLE 1 - ADDITION

Compute c = a+b

1) Allocate space on GPU for a, b, c — `cudaMalloc`
2) Copy values of a, b from host to device — `cudaMemcpy`
3) Launch kernel to add numbers — `add_on_GPU`
4) **Copy value of c from device to host** — `cudaMemcpy`
5) Free GPU memory — **`cudaFree`**

```
cudaFree(d_a);
```