TECHNISCHE UNIVERSITÄT DRESDEN

NVIDIA GPU CENTER OF EXCELLENCE

HZDR HELMHOLTZ ZENTRUM DRESDEN ROSSENDORF

Highly Parallel Programming of GPUs

Introduction

Wolfgang E. Nagel | Guido Juckeland | Robert Schöne

DRESDEN concept

Content of the Course

- Basics of parallel programming
- Basics of computational science simulations
- Introduction to CUDA and related techniques for using GPUs
- Basic techniques for high performing GPU codes
- case studies and real use cases from scientific partners

Goal:

- You are able to program parallel GPU accelerated applications yourself

Material for the Course

- https://bildungsportal.sachsen.de/opal/auth/RepositoryEntry/41232924675
  Technische Universität Dresden => Fakultät Informatik => Institut für Technische Informatik

Contact person:

- Guido Juckeland, +49-351-260-3660, Helmholtz-Zentrum Dresden-Rossendorf,
  g.juckeland@hzdr.de / guido.juckeland@tu-dresden.de

Prerequisites

- skills in using algortihms to solve problems
- ability to code algorithms in a programming language
- basic knowledge of the C programming language
- Linux baiscs (ls, cd, rm, …)
- Experience with using and IDE for editing, compiling, testing and debugging of self-written programs
- For CUDA development you can use the nsight IDE
- Experience with parallel programming is a plus, but not required

How will this course work in an online version?

Lectures:

- held in person (every Monday 11:10-12:40, APB/065)
- Videos from previous lectures will be posted after each lecture

Labs:

- Two types of labs
  - In person lab (every Tuesday 11:10-12:40, APB/E065) with local GPUS
  - Work on your own labs
  - You can use the ZIH HPC-system “taurus” -> intro tomorrow during office hours (also video availalbe)

Office hours:

- consultation time to ask questions on Tuesday 14:50-16:20 (via Zoom)

Lecture Roadmap

Lecture:	Mon 3.DS	Lab	Tue 3. DS
09.10.	---	10.10.	---
16.10.	Introduction	11.10.	Data transfers, error handling
23.10.	Kernels, Blocks, Grids	24.10.	Vector operations
30.10.	GPU architecture	31.10.	<holiday>
06.11.	Memory management 1	07.11.	2D data structures
13.11.	Memory management 2	14.11.	Matrix-matrix multiplication
20.11.	Computer Graphics	21.11.	Computer Graphics
27.11.	OpenACC	28.11.	OpenACC
04.12.	Deep Learning	07.12.	Deep Learnung
11.12.	Handout final project	12.12.	<time to meet with mentors>
18.12.	Program optimization	19.12.	(work on final project)
8.1.	CUDA Advanced	09.1.	(work on final project)
15.1.	CUDA Alternatives	10.1.	(work on final project)
22.1.	Case Studies	23.1.	(work on final project)
29.1.	Project Presentations 1	30.1.	Project Presentations 2

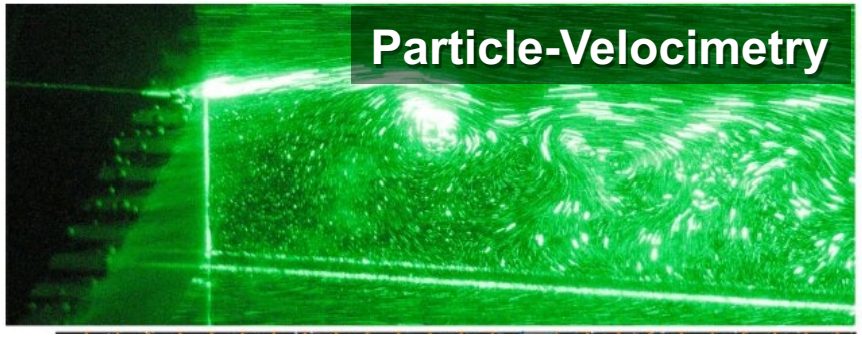
What is this “final project”?

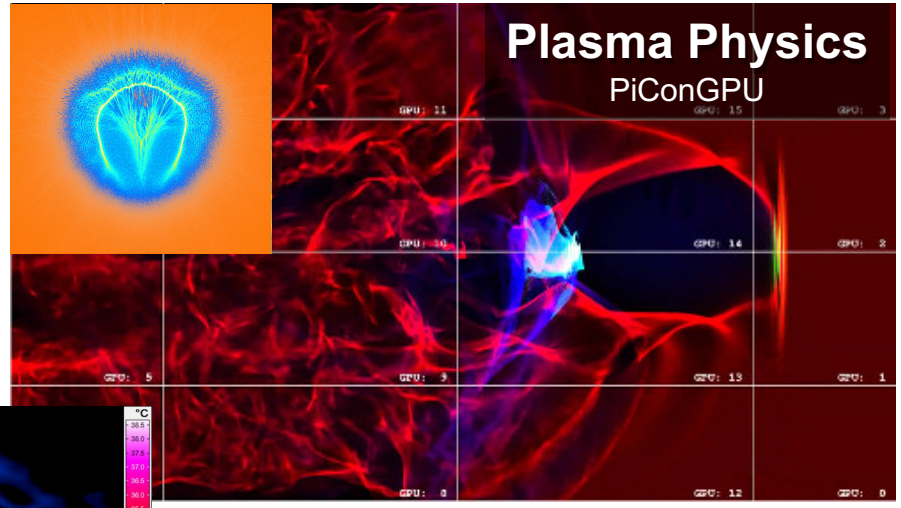
- offer from our partners at the *GPU Center of Excellence*
- small software projects instead of the usual labs
- team work (two to three persons)
- topics handed out on Dec 5
- presentation of your results (and software demo) on Jan 30/31

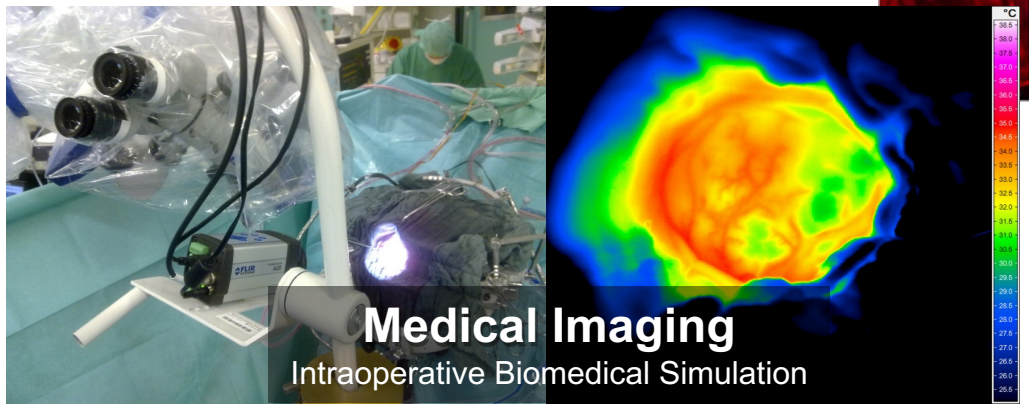
HELMHOLTZ ZENTRUM
DRESDEN ROSSENDORF

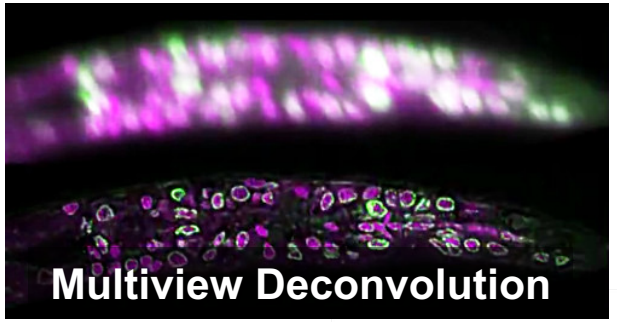
CAN I USE THE CONTENT OF THIS COURSE FOR ANYTHING IN REAL LIFE?

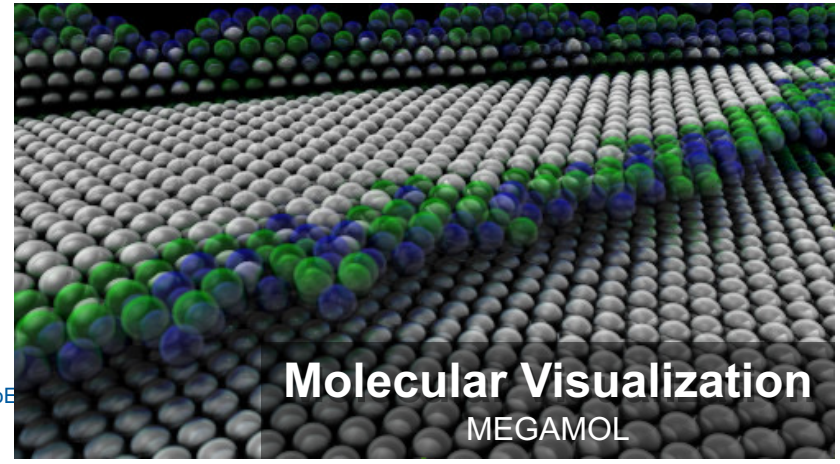
GCOE Projects



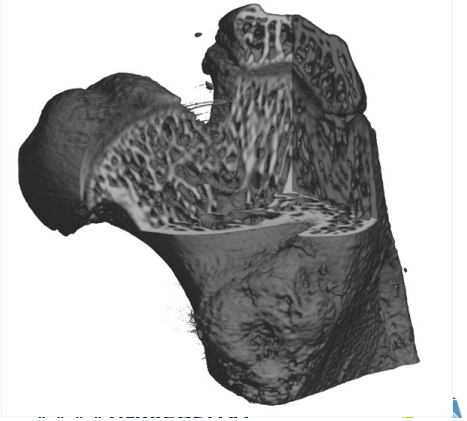






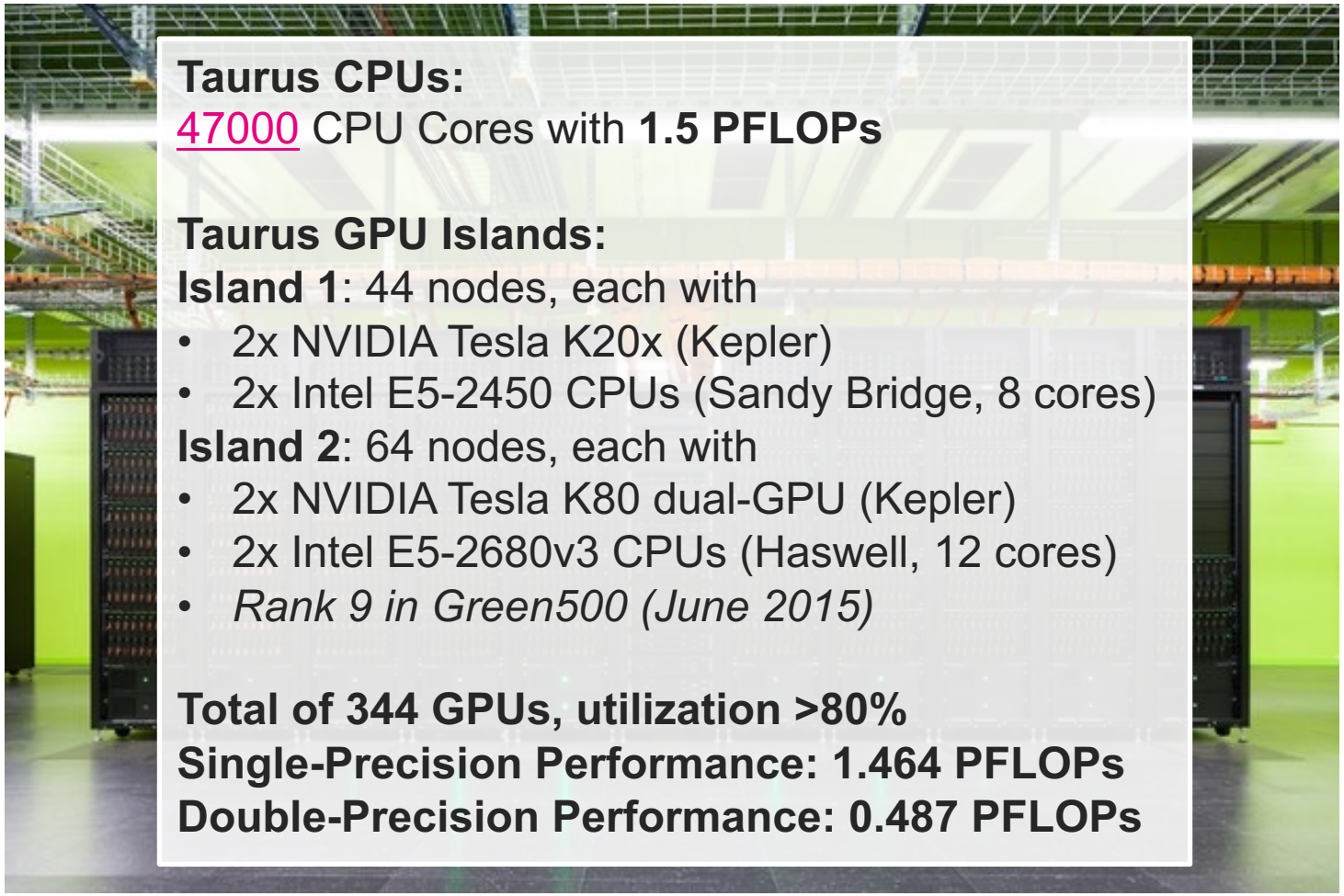


Geometry Processing



X-Ray Tomography

GPU Hardware at ZIH



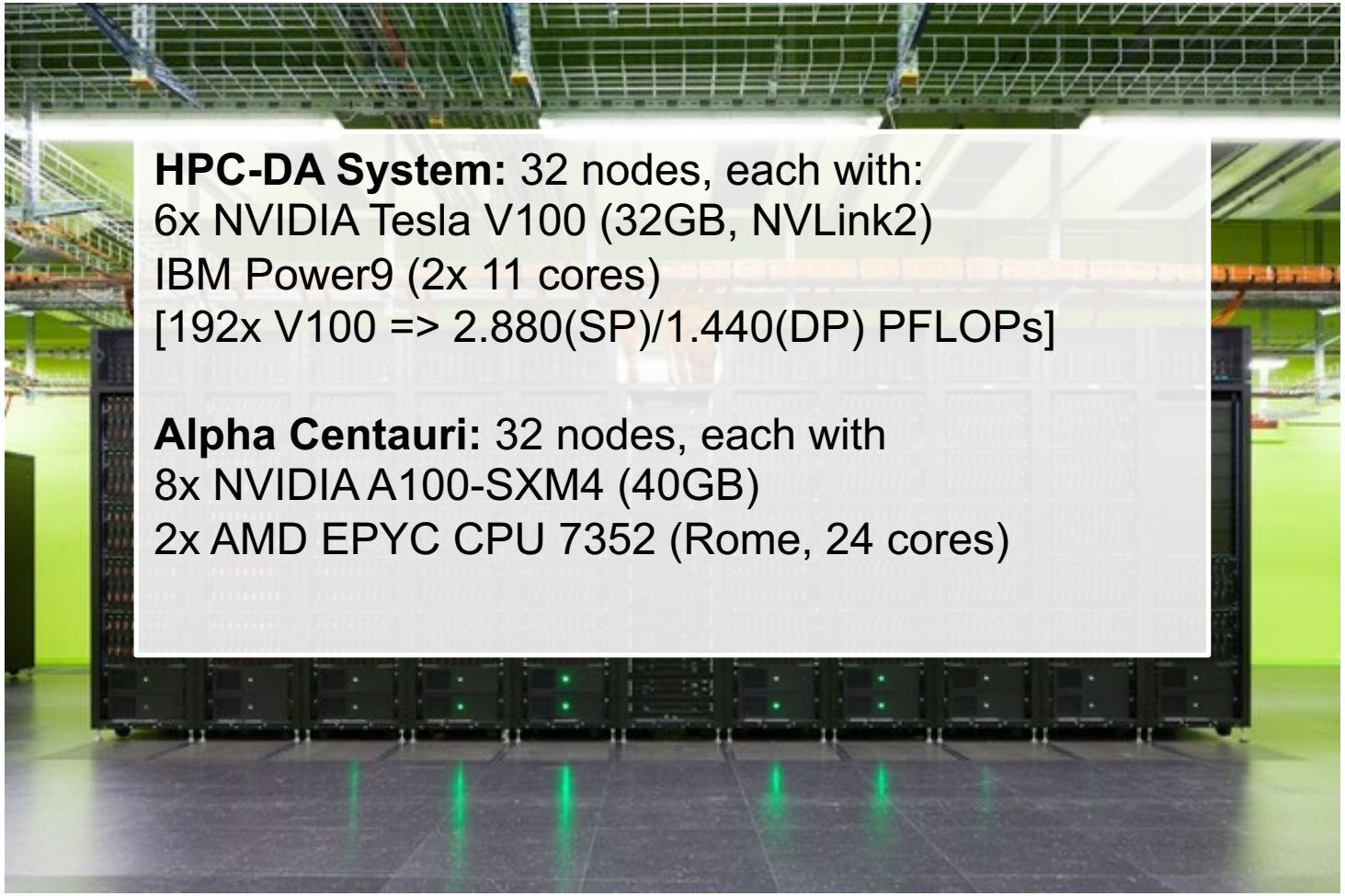
Taurus CPUs:
47000 CPU Cores with **1.5 PFLOPs**

Taurus GPU Islands:
Island 1: 44 nodes, each with
- 2x NVIDIA Tesla K20x (Kepler)
- 2x Intel E5-2450 CPUs (Sandy Bridge, 8 cores)

Island 2: 64 nodes, each with
- 2x NVIDIA Tesla K80 dual-GPU (Kepler)
- 2x Intel E5-2680v3 CPUs (Haswell, 12 cores)
- *Rank 9 in Green500 (June 2015)*

Total of 344 GPUs, utilization >80%
Single-Precision Performance: 1.464 PFLOPs
Double-Precision Performance: 0.487 PFLOPs

GPU Hardware at ZIH



HELMHOLTZ ZENTRUM
DRESDEN ROSSENDORF

Let‘s Get Started …

DRESDEN concept

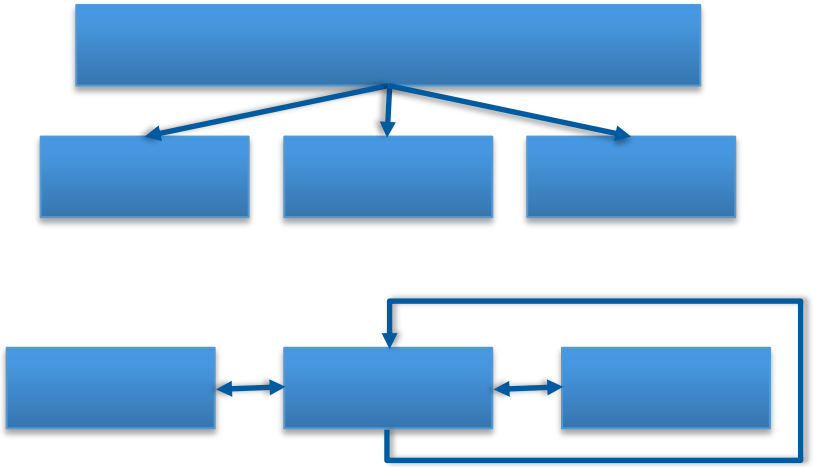
Parallelism is Everywhere

Parallelism occurs whenever multiple agents work together in solving a larger problem. You use it when one agent cannot solve the problem in time.

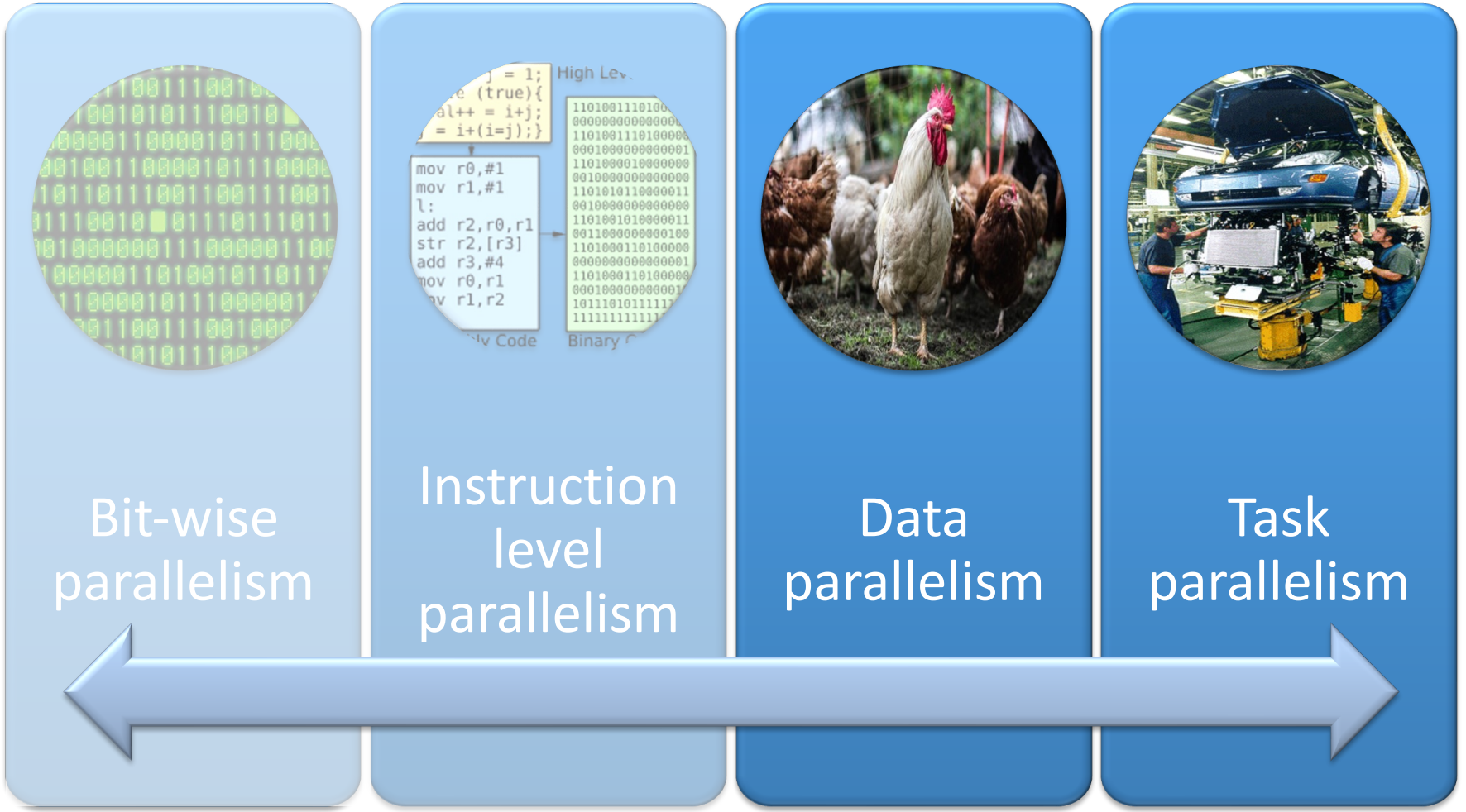
Parallel Computing

- Take a large computational problem
- Break it into smaller parts
- Solve the parts concurrently

- If necessary:
  - Communicate partial results
  - Repeat until solution is reached

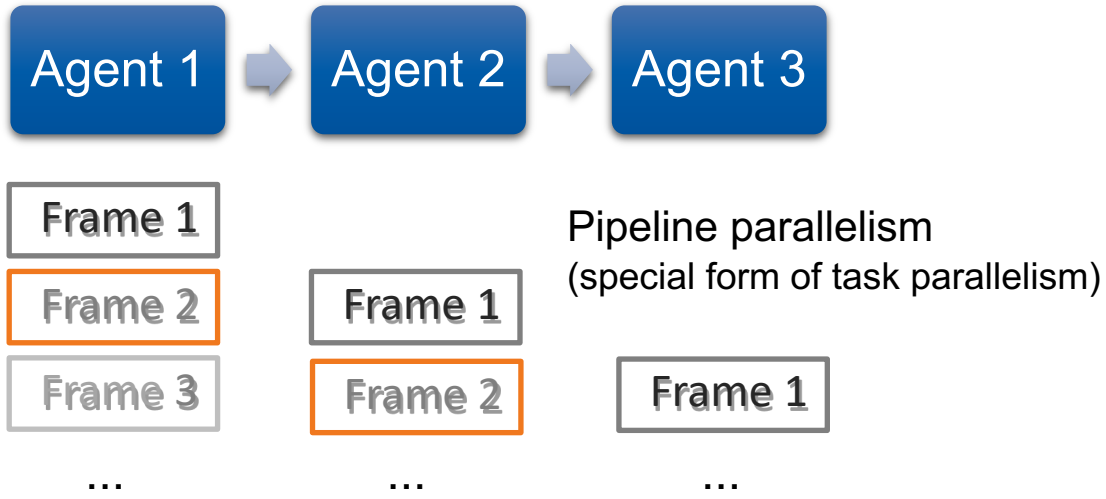


Types of Parallelism



Task Parallelism

- Performs (sub-)tasks on sets of data (concurrently = in parallel, if possible)
  - Example application: Video processing
  - Agent 1: Load frames
  - Agent 2: Remove blur
  - Agent 3: Adjust colors,…
  - …



- Agent = Thread or process
- Task parallelism usually does **not** scale with the problem size
- Load-balancing and synchronization are important
- Multi-core CPUs are often a good choice for task parallel applications
- On a coarse-grained level (low communication), GPUs also can handle it well, but their main talent is **data parallelism**

Data Parallelism

- Performs the same task on different data
  - Example: Video processing
  - Agent 1: works on top left corner
  - Agent 2: works on top right corner
  - Agent 3: works on bottom left corner
  - …

- Agent = Thread or process
- Works well on CPUs and GPUs
- Requires dividable data

Parallelization

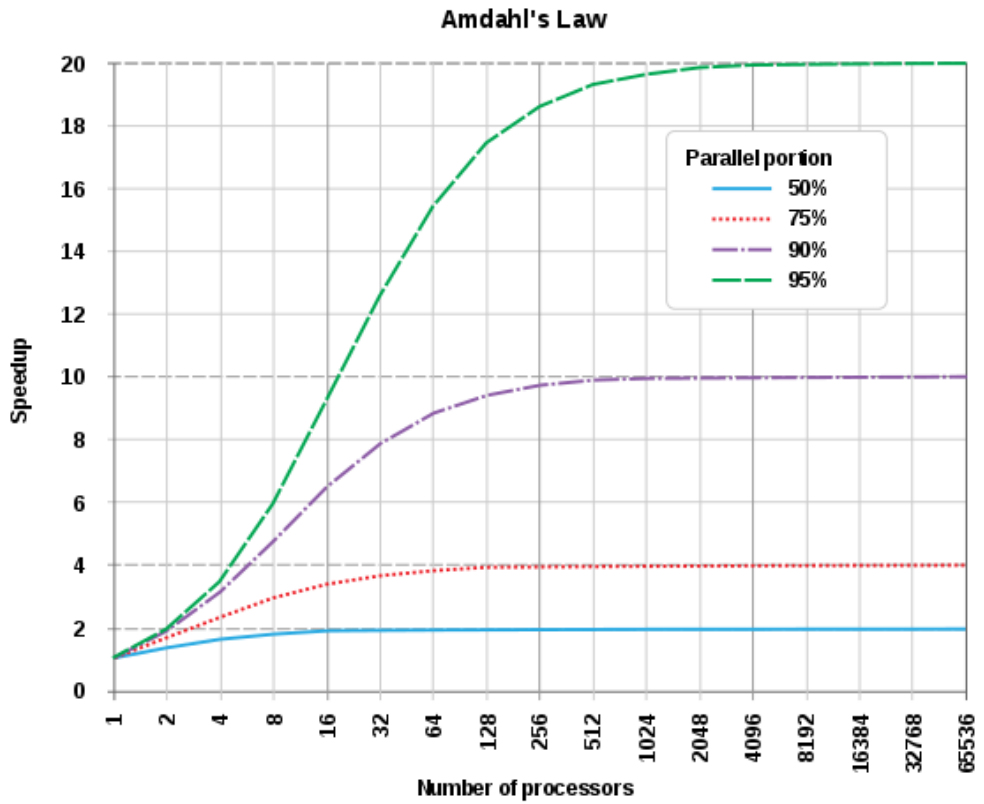
Porting or refactoring a code to run parallel on GPUs is usually no easy task

- Find **hotspots** and **embarassingly[1] parallel**-enabled portions
  - exploit data parallelism and SIMD programming model
- If serial part of the algorithm is too high, then parallelization does not help much
  - Amdahl‘s Law: $S = \frac{1}{(1-P)+\frac{P}{N}}$

S – theoretical speedup

N – number of processors

P – parallel portion



Daniels220 CC 3.0, wiki

Top 500 (June 2022)

Rank	Name	Computer	Site	Manufacturer	Country	Year	Total Cores	Accelerator/ Co-Processor Cores	Rmax [TFlop/s]	Rpeak [TFlop/s]	Power (kW)
1	Frontier	HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11	DOE/SC/Oak Ridge National Laboratory	HPE	United States	2021	8.699.904	8.138.240	1.194.000,00	1.679.818,75	22.703,00
2	Supercomputer Fugaku	Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D	RIKEN Center for Computational Science	Fujitsu	Japan	2020	7.630.848		442.010,00	537.212,00	29.899,23
3	LUMI	HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11	EuroHPC/CSC	HPE	Finland	2023	2.220.288	2.069.760	309.100,00	428.703,74	6.015,77
4	Leonardo	BullSequana XH2000, Xeon Platinum 8358 32C 2.6GHz, NVIDIA A100 SXM4 64 GB, Quad-rail NVIDIA HDR100 Infiniband	EuroHPC/CINECA	Atos	Italy	2023	1.824.768	1.714.176	238.700,00	304.465,92	7.404,40
5	Summit	IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband	DOE/SC/Oak Ridge National Laboratory	IBM	United States	2018	2.414.592	2.211.840	148.600,00	200.794,88	10.096,00
6	Sierra	IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband	DOE/NNSA/LLNL	IBM / NVIDIA / Mellanox	United States	2018	1.572.480	1.382.400	94.640,00	125.712,00	7.438,28
7	Sunway TaihuLight	Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway	National Supercomputing Center in Wuxi	NRCPC	China	2016	10.649.600		93.014,59	125.435,90	15.371,00
8	Perlmutter	HPE Cray EX235n, AMD EPYC 7763 64C 2.45GHz, NVIDIA A100 SXM4 40 GB, Slingshot-10	DOE/SC/LBNL/NERSC	HPE	United States	2021	761.856	663.552	70.870,00	93.750,00	2.589,00
9	Selene	NVIDIA DGX A100, AMD EPYC 7742 64C 2.25GHz, NVIDIA A100, Mellanox HDR Infiniband	NVIDIA Corporation	Nvidia	United States	2020	555.520	483.840	63.460,00	79.215,00	2.646,00
10	Tianhe-2A	TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000	National Super Computer Center in Guangzhou	NUDT	China	2018	4.981.760	4.554.752	61.444,50	100.678,66	18.482,00

Parallel C++ APIs (Incomplete)

	CPU	GPU (NVIDIA)	GPU (AMD)	FPGA
CUDA	-	x	-	-
HCC	x	-	x	-
OpenCL	x	x	x	x
HIP	x	x (via nvcc)	x (via hcc/clang)	-
SYCL	x	x (experimental)	x (prototype)	x
OpenGL Direct3D Vulkan	-	x	x	-
OpenMP4/5	x	x	x	x
OpenACC (GCC, NVIDIA)	x	x	x	o, o
MPI	x	GPU transfers if MPI is CUDA-aware	-	-

Parallel Hardware Examples (Incomplete)

Nvidia	**GTX**	**Tesla**	**SoC**
Kepler	680	K80	Tegra K1
Maxwell	980	M40	Tegra X1
Pascal	1080	P100	Tegra X2
Volta	(Titan)	V100	Xavier
Turing	2080	-	Orin?
Ampere	*3080*	*A100*	Orin?
Hopper	4080.	H100	?

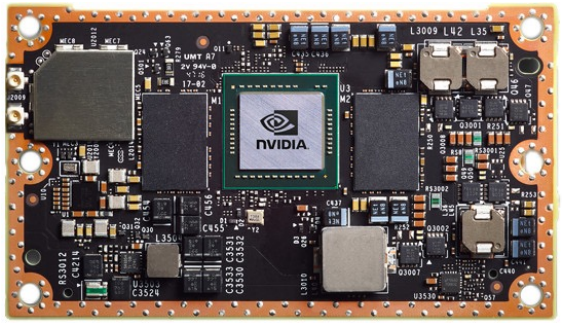
AMD	**Gaming**	**Pro (W+S)**	**MI**
GCN 3rd	R9 285	FirePro S7150	MI8
GCN 4th	RX 480	RadeonPro 580	MI6
GCN 5th	RX Vega64	RadeonPro SSG	MI25
RDNA	RX 5700	RadeonPro W5000	MI100
RDNA2	RX 6800	RadeonPro W6000	MI2xx
RDNA3	RX 7900	RadeonPro W7000	MI300

Some Accelerator Cards

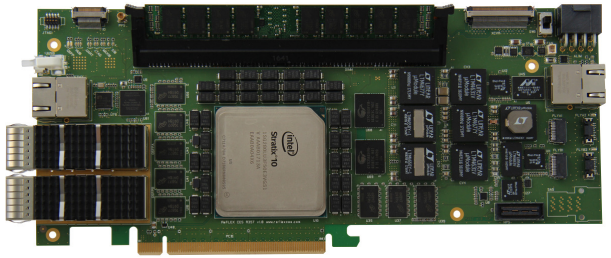
Nvidia Tesla V100 (Volta) (2017)

… semi-custom AMD GPUs (2017)

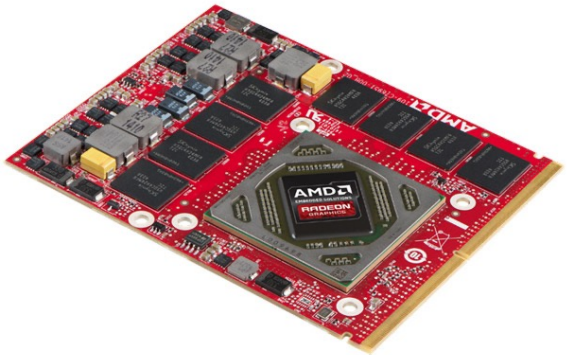
Radeon Pro SSG (Vega) (2017)



Nvidia Tegra X2 (Pascal) (2016)

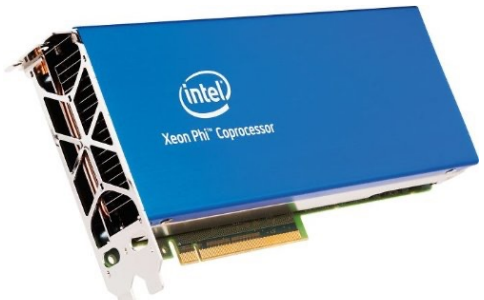


Altera Stratix 10 FPGA (2013)



Radeon Embedded E9550 MXM (Polaris) (2016)

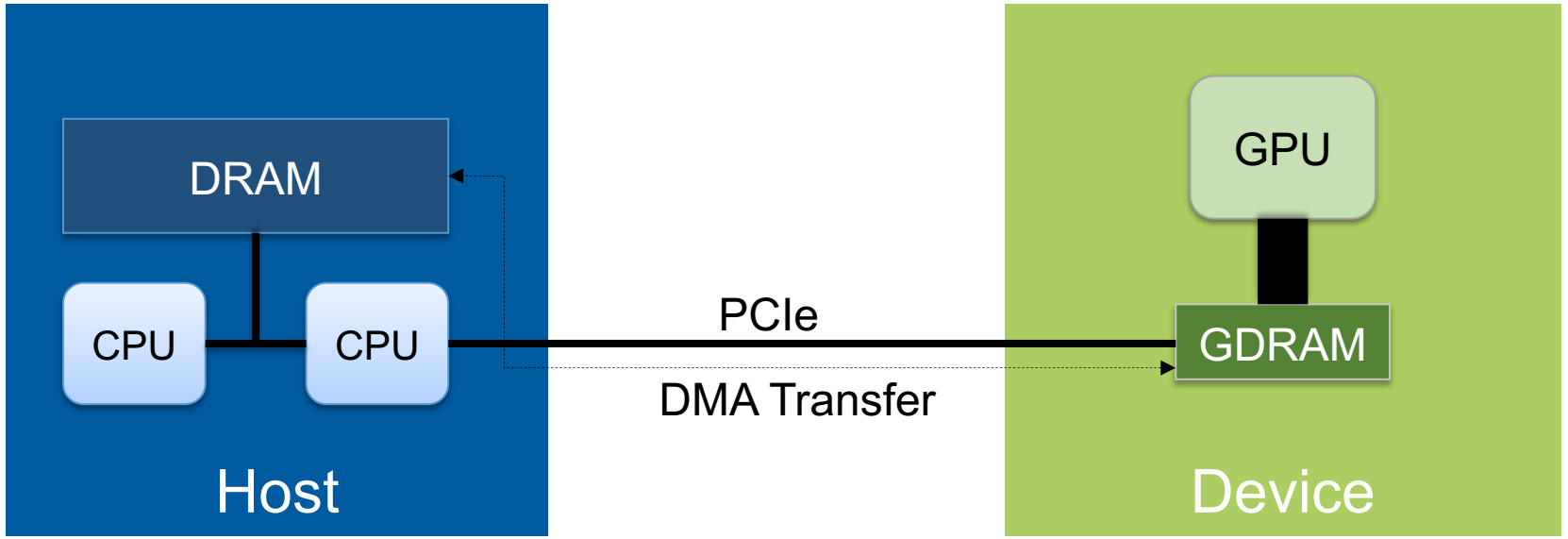
Google TPU (2017)



Intel Xeon Phi 7120P (2013)

GPU System Setup

- CUDA assumes a system with a host and a device with own memory each



Hardware

Software

Defining the Terms HOST and DEVICE

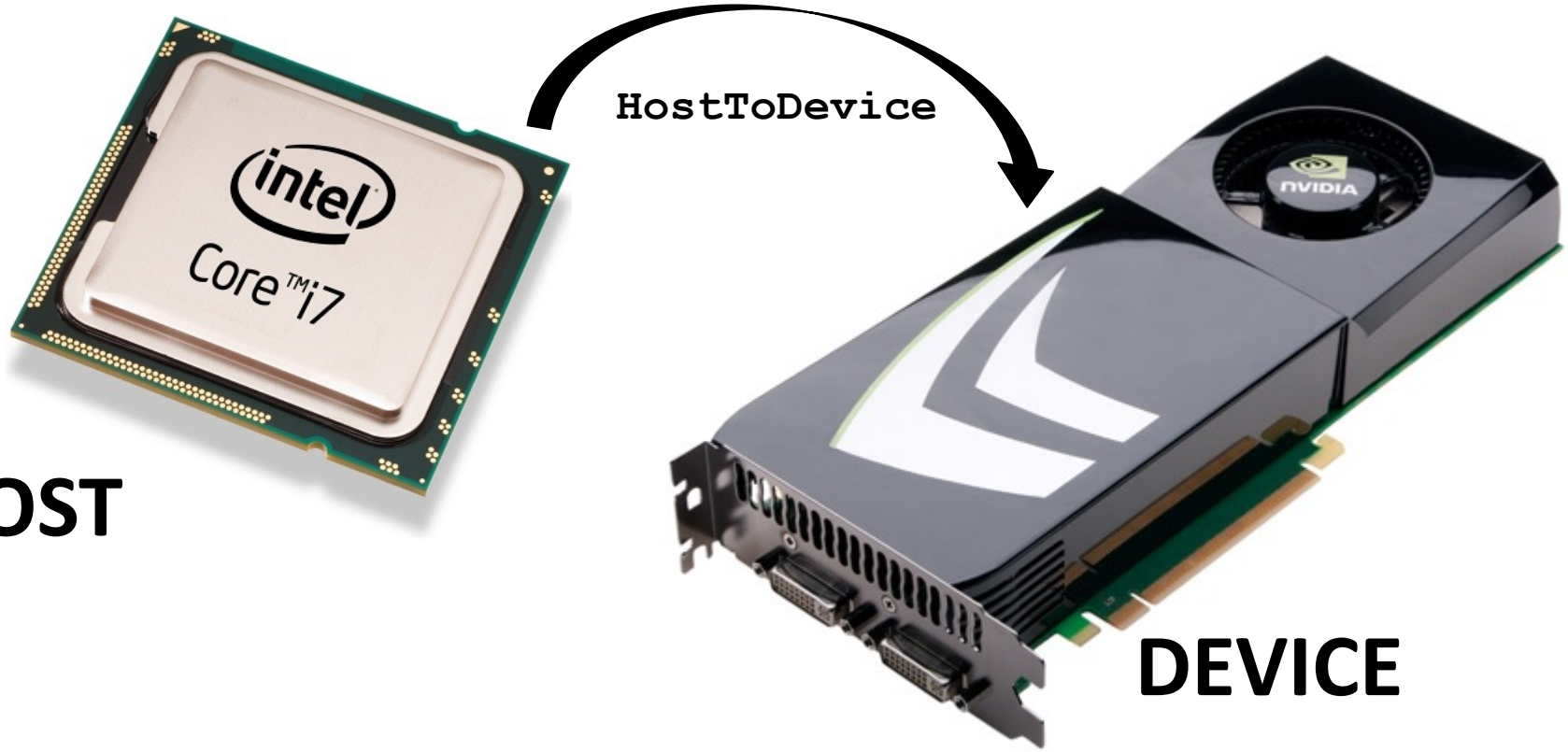
HOST

DEVICE

CUDA: Allocating and Freeing Memory on the Device

```
int main( void ) {

  float *device_pointer;        //Pointer to float element


  // The following line allocates memory for one float on the GPU and
  // sets device pointer to the beginning of that memory area
  cudaMalloc( &device_pointer, sizeof(float) );


  // The following line releases the allocated GPU memory for
  // device_pointer so that it may be used again by another allocation
  cudaFree( device_pointer );


  return 0;
}
```

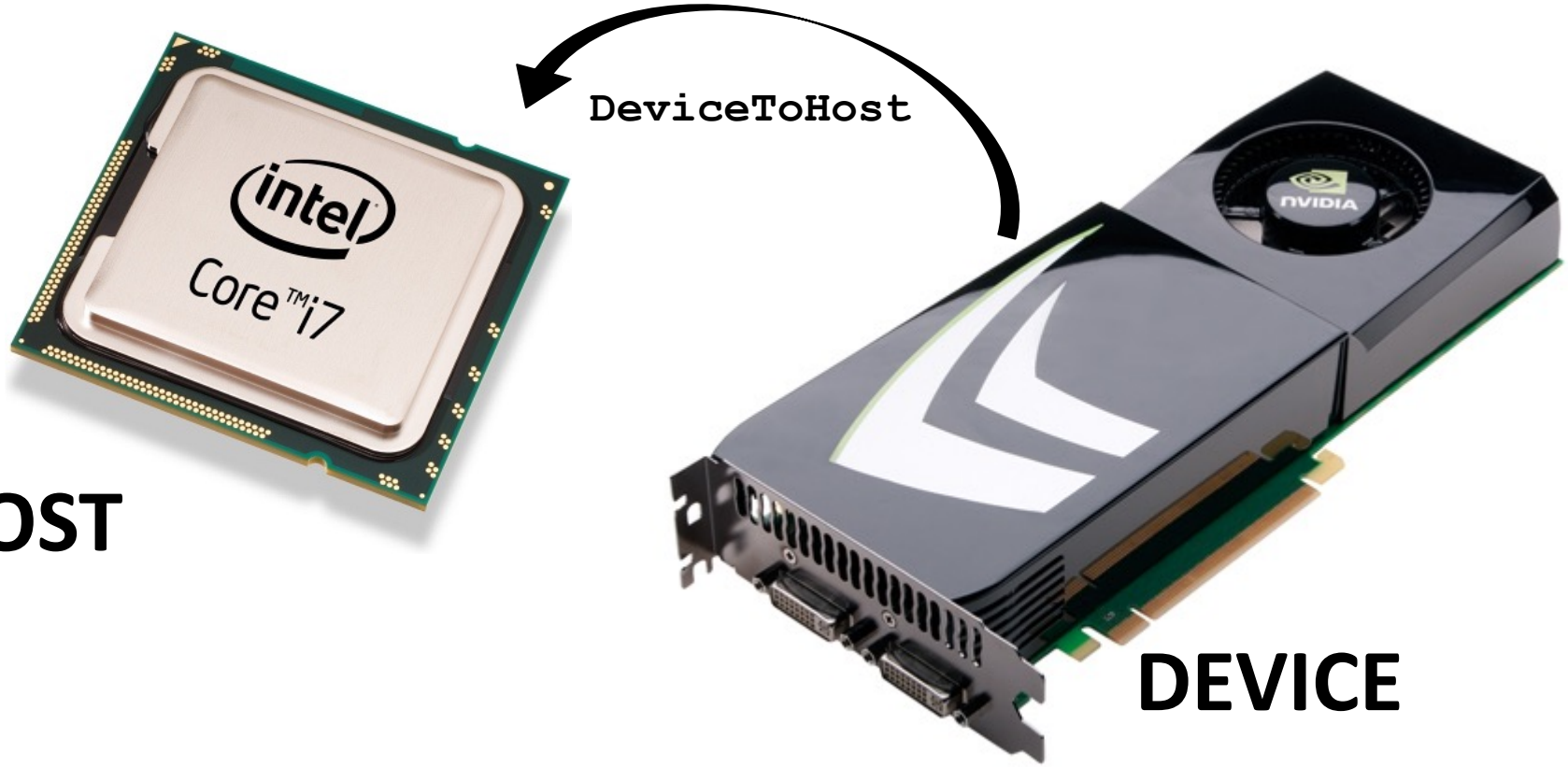
Data Transfers (1)



Data Transfer from HOST to DEVICE

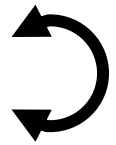
```
#define NELEMENTS 16


int main( void ) {
        float hostvariable[NELEMENTS];                  //float-array on the HOST
        float *device_pointer;                          //allocated pointer to the DEVICE
        cudaMalloc( &device_pointer, NELEMENTS*sizeof(float) );
        cudaMemcpy( device_pointer,                     //Pointer to DEVICE memory (dest.)
                hostvariable,                           //Pointer to host memory (source)
                NELEMENTS*sizeof(float),                //number of bytes to transfer
                cudaMemcpyHostToDevice );               //direction of transfer
        cudaFree( device_pointer );
        return 0;
}
```

Data Transfers (2)



Data Transfers from DEVICE to HOST

```
#define NELEMENTS 16

int main( void ) {
        float hostvariable[NELEMENTS];                          //float-array on the HOST
        float *device_pointer;                                  //allocated pointer to the DEVICE
        cudaMalloc( &device_pointer, NELEMENTS*sizeof(float) );
        cudaMemcpy( hostvariable,                               //Pointer to HOST memory (dest.)
                    device_pointer,                             //Pointer to DEVICE memory (source)
                    NELEMENTS*sizeof(float),                    //number of bytes to transfer
                    cudaMemcpyDeviceToHost );                   //direction of transfer
        cudaFree( device_pointer );

        return 0;

}
```



There is more than one copy operation available

cudaMemcpy
cudaMemcpy2D
cudaMemcpy2DArrayToArray
cudaMemcpy2DAsync
cudaMemcpy2DFromArray
cudaMemcpy2DFromArrayAsync
cudaMemcpy2DToArray
cudaMemcpy2DToArrayAsync
cudaMemcpy3D
cudaMemcpy3DAsync
cudaMemcpy3DPeer
cudaMemcpy3DPeerAsync
cudaMemcpyAsync

cudaMemcpyPeer
cudaMemcpyPeerAsync
cudaMemcpyArrayToArray
cudaMemcpyFromArray
cudaMemcpyFromArrayAsync
cudaMemcpyToArray
cudaMemcpyToArrayAsync
cudaMemcpyFromSymbol
cudaMemcpyFromSymbolAsync
cudaMemcpyToSymbol
cudaMemcpyToSymbolAsync

Possible transfer directions

`cudaMemcpyHostToDevice` → Transfers data from host to device

`cudaMemcpyDeviceToDevice` → Transfers data on the device

`cudaMemcpyDeviceToHost` → Transfers data from device to host

`cudaMemcpyHostToHost` → Transfers data on the host

`cudaMemcpyDefault` → Detects by memory address

requires Unified Virtual Addressing (UVA)

CUDA Hello World on GPU

```
#include <stdio.h>
// … CUDA headers are added by nvcc

__global__ void mykernel(void) {
        printf("Hello World!\n");
}
int main(void) {
        mykernel<<<1,1>>>();
        return 0;
}
```

… more on nvcc compiler coming soon

Instructions:

```
$ nvcc main.cu
$ ./a.out
Hello World!
```

Catching errors with `cudaError_t`

CUDA functions return a value of type cudaError_t:

```
typedef enum cudaError cudaError_t
```

Values can be for instance:

cudaSuccess	The API call returned with no errors. In the case of query calls, this can also mean that the operation being queried is complete (see `cudaEventQuery()` and `cudaStreamQuery()`).
cudaErrorMemoryAllocation	The API call failed because it was unable to allocate enough memory to perform the requested operation.
cudaErrorInitializationError	The API call failed because the CUDA driver and runtime could not be initialized.
cudaErrorLaunchFailure	An exception occurred on the device while executing a kernel. Common causes include dereferencing an invalid device pointer and accessing out of bounds shared memory.
…	…

Automatic Error Interpretation

In order to interpret **`cudaError_t`** one can use the following function

const char* **cudaGetErrorString**(cudaError_t *error*)

Example:

```
cudaError_t error = cudaGetLastError();
printf( "CUDA error: %s\n", cudaGetErrorString( error ) );
```

Catch and Handle Errors by Default

```
#define HANDLE_ERROR( err ) \
                              (handleCudaError( err, __FILE__, __LINE__ ) )

static void handleCudaError( cudaError_t err, const char *file, int line ) {

  if (err != cudaSuccess) {
    printf( "%s in %s at line %d\n", cudaGetErrorString( err ),  file, line );
    exit( EXIT_FAILURE );
  }
}
```

- Prepend all CUDA calls with HANDLE_ERROR, e.g.:

  HANDLE_ERROR(cudaMalloc(&devicepointer, sizeof(float)));

- Always check kernel launch:

  mykernel<<<1,1>>>();

  HANDLE_ERROR(cudaGetLastError());

HELMHOLTZ ZENTRUM
DRESDEN ROSSENDORF

Developer Tools

DRESDEN concept

CUDA Developer Tools

- IDE
  - **Nsight Eclipse / Visual Studio (Linux/Mac / Windows)**
    - also for Android: Nsight Tegra (Eclipse / Visual Studio)
  - **comes with integrated debugging and analysis tools**

- Standalone Performance Tools (CUDA >=10)
  - **Nsight Systems - System-wide application algorithm tuning**
  - **Nsight Compute - Debug/optimize specific CUDA kernel**
  - **Nsight Graphics - Debug/optimize specific graphics shader**

- Classic Tools Set
  - **Memory & Race Checker** `cuda-memcheck`
  - **Built-in profiler** `nvprof`
  - **Visual Profiler** `nvvp`
  - **Debugger** `cuda-gdb`

https://developer.nvidia.com/debugging-solutions
https://developer.nvidia.com/gameworks-tools-overview
https://developer.nvidia.com/develop4tegra

Tools Comparison

	NVIDIA © Nsight ™ Systems	NVIDIA© Nsight™ Compute	NVIDIA© Visual Profiler	Intel © VTune ™ Amplifier	Linux perf OProfile
Target OS	**Linux**	Linux, Windows	Linux, Mac, Windows	Linux, Windows	Linux
GPUs	**Pascal, Volta, Future**	Pascal, Volta, Future	Kepler, Maxwell, Pascal, Volta, Future	None	None
CPUs	**x86_64**	x86_64	x86, x86_64, Power	x86, x86_64	x86, x86_64, Power
Trace	**NVTX, CUDA, OpenGL, CuDNN, CuBLAS, System**	NVTX, CUDA	MPI, CUDA, OpenACC	MPI, ITT	Kernel
PC Sampling	**High Speed**	No	Yes	High Speed	High Speed
UVM, NVLINK, Power,Thermal	**Future**		Yes	No	No
Src Code View	**No**	Yes	Yes	Yes	No
Compare Sessions	**No**	Yes	Yes	Yes	No

S8718-Robert-Knight-John-Stone.pdf

General Tools Overview

Score-P

- collect profiles & traces (suited for large-scale parallel applications)

CUBE

- Profile browser

Vampir

- Trace visualizer (relies on Score-P)

AMD

- µProf, ROCm profiler, GPUPerfAPI

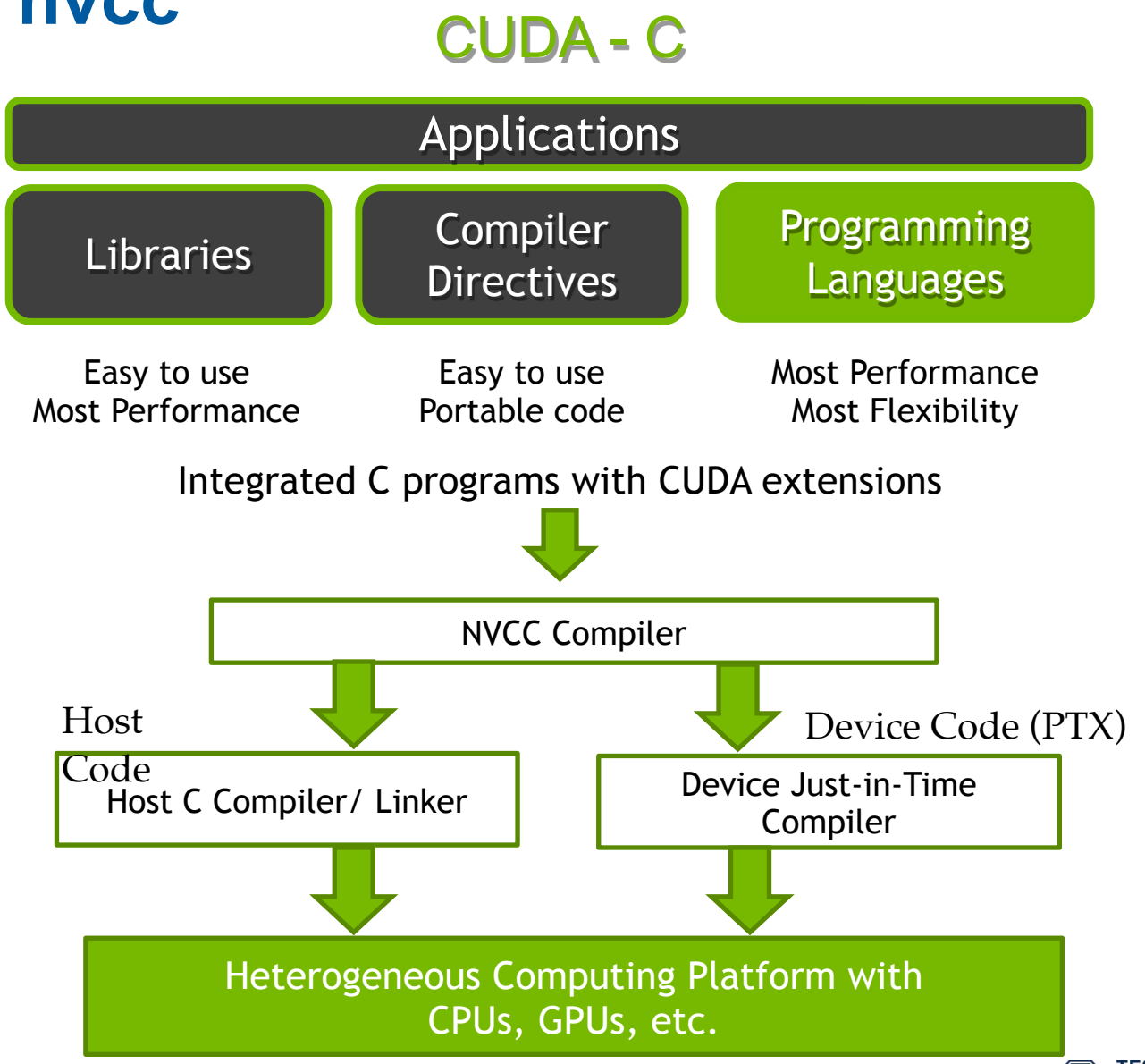
IntelVtune

- Performance Analyzer for Intel devices

Arm DDT

- find bugs on both small & large clusters

CUDA C and nvcc



Nvidia's Compute Capability (CC)

- Defines set of GPU hardware and programming features
- CC is a version tag = `<major>.<minor>`, e.g. 3.7 for K80, where "3" marks Kepler and "7" the minor version within Kepler architecture
- Kernel codes designed for older CC still work on latest GPUs
  (probably runs inefficiently)
- So Kepler code runs on Volta, but Volta code may not work on Kepler (if you use Volta features like thread-independent scheduling)
- GPU generation and corresponding CUDA features must be known to write good code
  - How much shared memory per block can be used?
  - How many blocks can be resident on an SM?
  - Is half-precision hardware available? ...

The CUDA wiki has a nice lookup matrix for the CC features.

Nvidia's CUDA Compiler

NVCC compiler (is based on LLVM)

- Compiles C/C++ CUDA code parts and delegates host code to host compiler (gcc, msvc, …)
- Allows to compile to a intermediate virtual language PTX (must be compiled at runtime by the driver) or directly to machine code (GPU must match CC)
- Different generations can be compiled into the binary

<=SM2x – Older cards such as GeForce GTX590 (removed as of CUDA9)

SM3x – **Kepler**

SM5x – **Maxwell**

SM6x – **Pascal** (requires CUDA8 to compile)

SM7.0-7.2 – **Volta** (requires CUDA9 to compile)

SM7.5 – **Turing** (requires CUDA10 to compile)

Compile and Run a CUDA Program

- On remote systems make sure that the CUDA environment is available (usually requires a **`module load cuda`** or similar)
- Name your CUDA files with the suffix **`.cu`**
- Compile your program using **`nvcc`** (e.g. `nvcc myprogram.cu`)
- Execute your program by running **`./a.out`**
- On remote systems the node you compile on might not feature a GPU, you will have to use a batch system to access node with GPUs
- nvcc supports some host compiler flags, otherwise use **`-Xcompiler`** to forward to host compiler, e.g. **`-Xcompiler -fopenmp`**
- Debugging flags
  - -g: include host debugging symbols
  - -G: include device debugging symbols (turns off optimizations!)
  - -lineinfo: include line information with symbols (also for profiling)

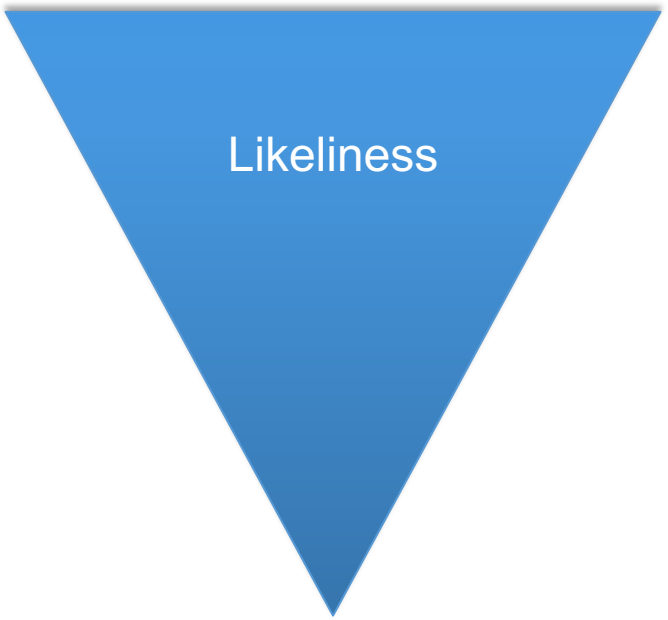
nvcc doc

The Five Basic CUDA Functions

- **`cudaMalloc`** to allocate memory on the device
- **`cudaMemcpy`** to transfer data to and from the device
- Kernel invocations
- Handling errors
- **`cudaFree`** to release allocated memory on the device

The Five Results of CUDA Programming

- Compiler error
- Program crashes
- Program produces wrong results
- Program runs very slow
- Program runs fast and correct



(Five) Recommendations

- Use comments
- Implement error handling
- Test for correctness
- Use expressive names for functions and variables, e.g. put a “_d” and a “_h” as a suffix to now data locality
- Use building blocks where possible (libraries, function reuse, etc.)