GPU COMPUTING

An Introduction

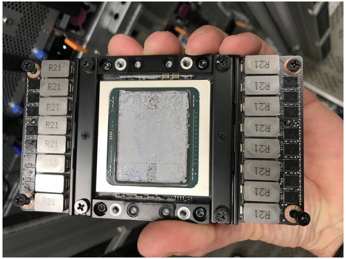
August 10, 2021 | Luis Altenkort, Marius Neumann, Marcel Rodekamp, Christian Schmidt, Xin Wu

THE COMPETENCE NETWORK FOR HIGH-PERFORMANCE COMPUTING IN NRW.

WHAT ARE GPUS AND WHY DO WE NEED THEM?

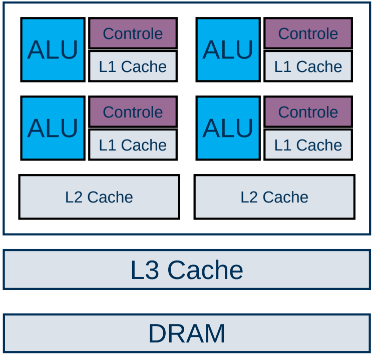
HPC.NRW

- A graphics processing unit (GPU) is a processor featuring a highly parallel structure, making it efficient at processing large blocks of data.
- Large data sets can be worked on by multiple cores. Two methods are commonly used:
  - multi CPU ($\sim 10 - 100$ cores)
  - GPUs (~ 5000 cores)
- Where are GPUs used?
  - video/graphics applications
  - linear algebra
  - machine learning
  - general purpose GPU (GPGPU)



CPU SCHEMATICS

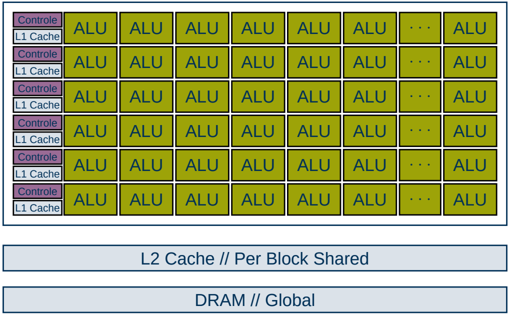
HPC.NRW



- $\mathcal{O}(10)$ Algorithmic-Logic-Units (ALUs) with broad instruction set
- On Chip:
  - Sophisticated Controle Units
  - Local Per Core L1 Cache
  - Per Core L2 Cache
  - L3 Cache
- Off Chip DRAM
- low latency optimized
- versatile usage

GPU SCHEMATICS

HPC.NRW



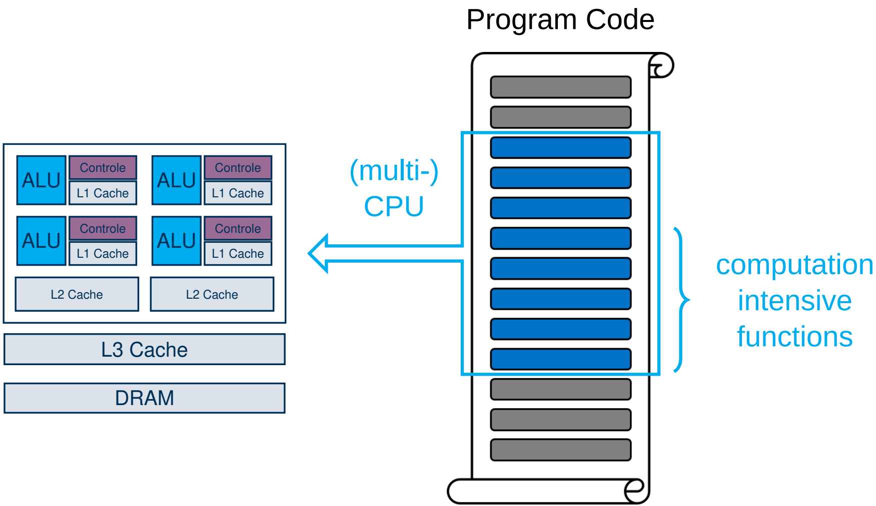
- $\mathcal{O}(10^9)$ ALUs with limited instruction set
- On Chip:
  - Basic Controle Units
  - Local Per Thread L1 Cache
  - Per Block L2 Cache
- Off Chip DRAM
- high throughput optimized
- limited usage

HARDWARE SPECIFICATION

	CPU	GPU
company	Intel	Nvidia
processor	Xeon Silver 4114	Tesla V100
launch (year)	2017	2017
clock freq. (GHz)	2.20 - 3.00	1.4
no. of cores	10	5120
cache (MB)	13.75	6.0
memory size (GB)	404	32
TDP (W)	85	300
SP peak (TFLOP/s)	> 0.7	> 15.7
bandwidth (GB/s)	4	300

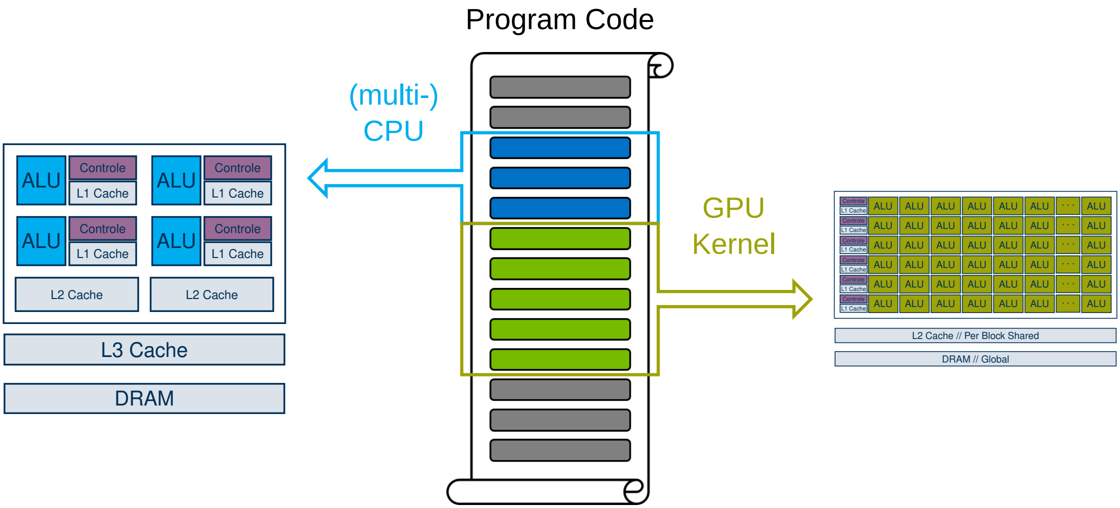
CODE STRUCTURE: CPU CODE

HPC.NRW



CODE STRUCTURE: GPU CODE

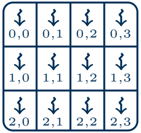
HPC.NRW



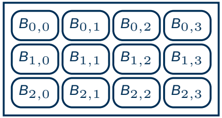
THE KERNEL

- The kernel spawns Threads, Blocks and Grids
  - Thread: Smallest executing unit
    - ALU with limited instruction set, but many availible
  - Block: collection of threads
    - share memory
  - Grid: collection of blocks
    - do not share memory

Threads



2D Block

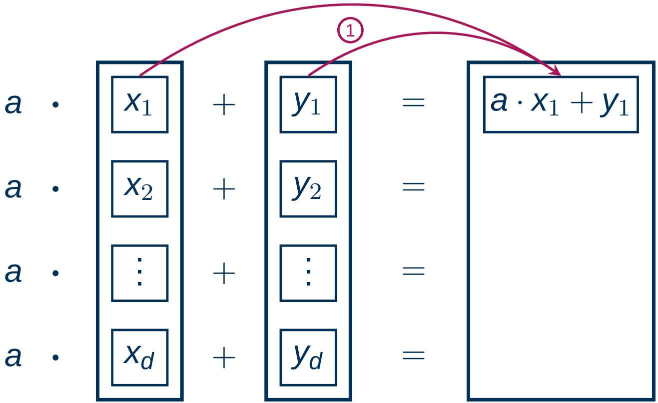


2D Grid

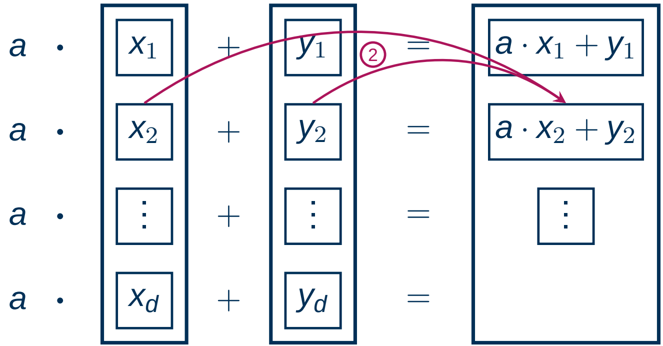
AN EXAMPLE: SAXPY

$$a \cdot \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{pmatrix} + \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_d \end{pmatrix} = \begin{pmatrix} a \cdot x_1 + y_1 \\ a \cdot x_2 + y_2 \\ \vdots \\ a \cdot x_d + y_d \end{pmatrix}$$

- SAXPY: **S**ingle Precision **A** times **X** **P**lus **Y**
  - Compute $a \cdot \vec{x} + \vec{y}$

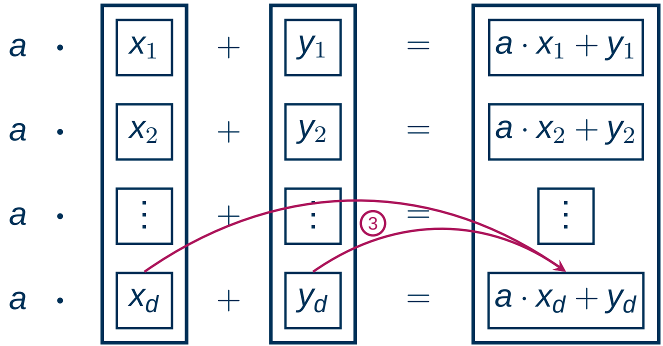


- SAXPY: **S**ingle Precision **A** times **X** **P**lus **Y**
  - Compute $a \cdot \vec{x} + \vec{y}$
- Sequential
  1. Compute $a \cdot x_1 + y_1$



- SAXPY: **S**ingle Precision **A** times **X** **P**lus **Y**
  - Compute $a \cdot \vec{x} + \vec{y}$
- Sequential
  1. Compute $a \cdot x_1 + y_1$
  2. Compute $a \cdot x_2 + y_2$

  $\dots$

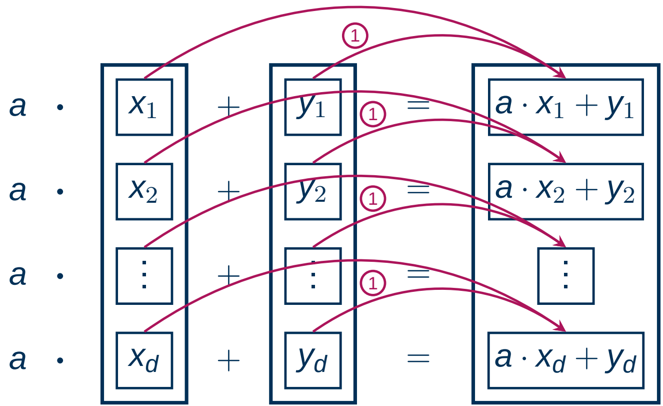


- SAXPY: **S**ingle Precision **A** times **X** **P**lus **Y**
  - Compute $a \cdot \vec{x} + \vec{y}$
- Sequential
  1. Compute $a \cdot x_1 + y_1$
  2. Compute $a \cdot x_2 + y_2$

     ...

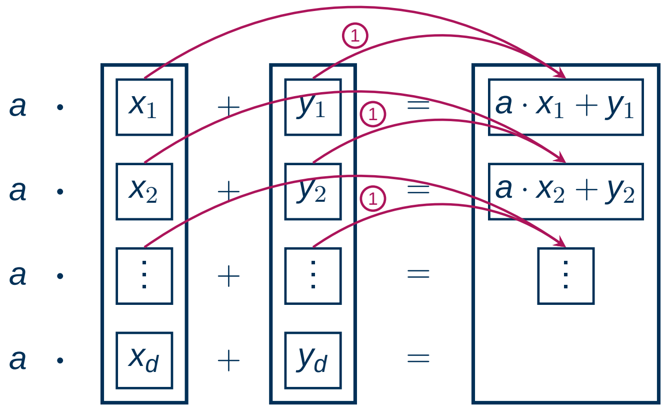
  3. Compute $a \cdot x_d + y_d$

AN EXAMPLE: SAXPY



- SAXPY: **S**ingle Precision **A** times **X** **P**lus **Y**
  - Compute $a \cdot \vec{x} + \vec{y}$
- Sequential
- Parallel
  1. Compute $a \cdot x_i + y_i$ for all $i \in [1, d]$ at once

AN EXAMPLE: SAXPY

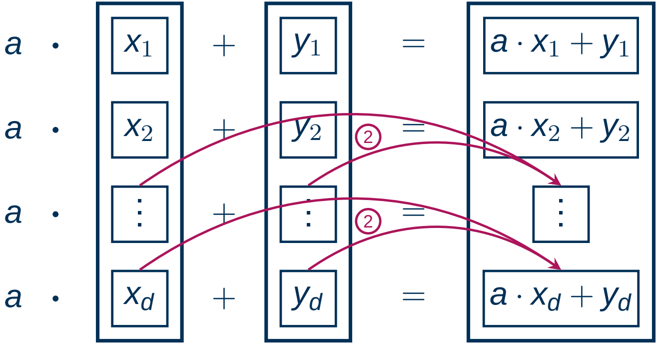


- SAXPY: **S**ingle Precision **A** times **X** **P**lus **Y**
- Sequential
- Parallel
  1. Compute $a \cdot x_1 + y_1$

     . . .

     Compute $a \cdot x_4 + y_4$

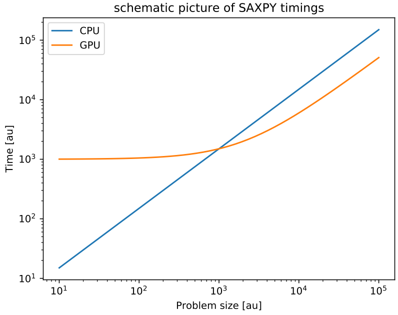
AN EXAMPLE: SAXPY



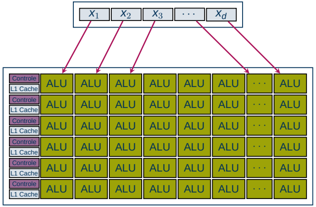
- SAXPY: **S**ingle Precision **A** times **X** **P**lus **Y**
- Sequential
- Parallel
  2. Compute $a \cdot x_5 + y_5$
     $\cdots$
     Compute $a \cdot x_8 + y_8$
  3. $\cdots$

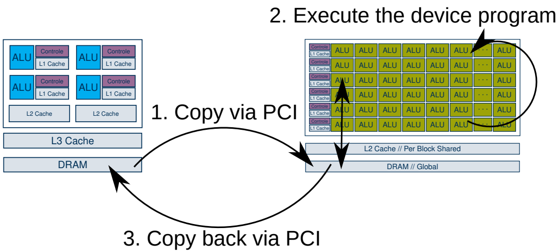
TIMING

- What speedups do we expect for CPU and GPU?
- CPU
  - neglectible overhead
  - linear
- GPU
  - high overhead
  - faster than CPU once overhead is small compared to problem size



- Why is the GPU sometimes slower?
  - A GPU core is less performant than a CPU core
  - Data copy required
  - Arithmetic intensity
  - SAXPY requires too less flops
  - ⇒ Memory bound





SEVERAL WAYS TO SAXPY ON GPUS

– Several languages we are going to look at:

  – cuBLAS
  – openACC
  – OpenMP
  – Thrust
  – CUDA C++
  – CUDA Fortran
  – Julia + CUDA.jl
  – CUDA Python
  – OpenCL
  – Sycl
  – OneAPI
  – Magma
  – Kokkos