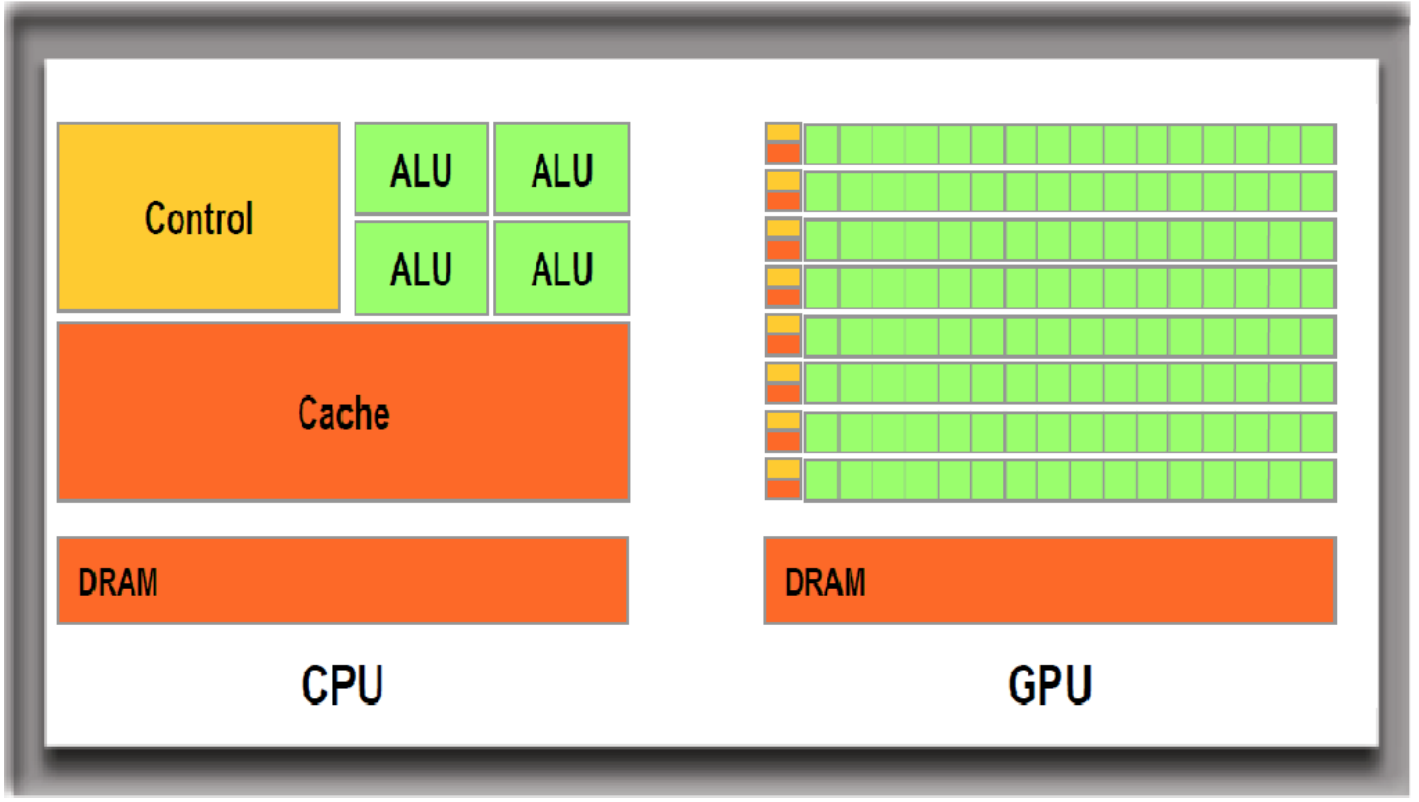
# GPUs vs CPUs, OpenACC vs OpenMP

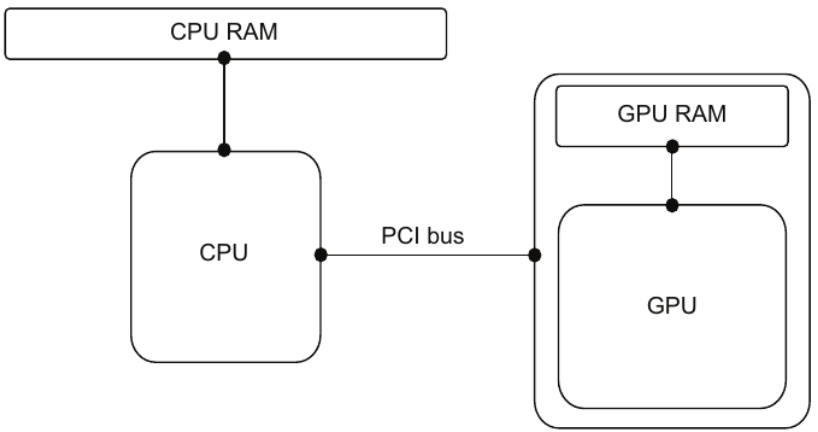
- What is a GPU?
- Its programming model
- OpenACC
- the shallow water model



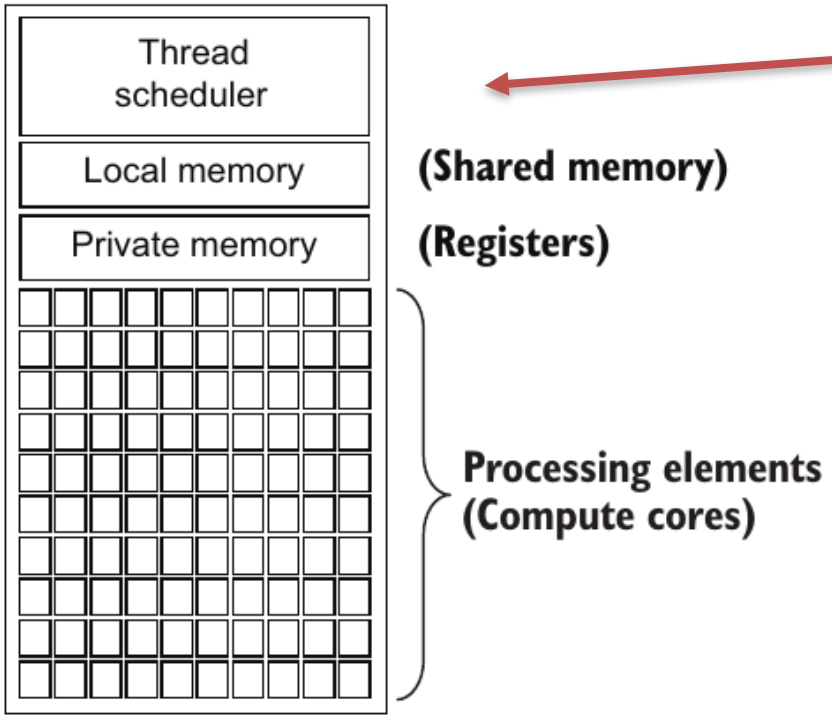

CPUs are designed to handle a wide-range of tasks, but are limited in the concurrency of tasks that can be running.

GPUs can be an order of magnitude faster due to parallelism, but they are not as versatile as CPUs.

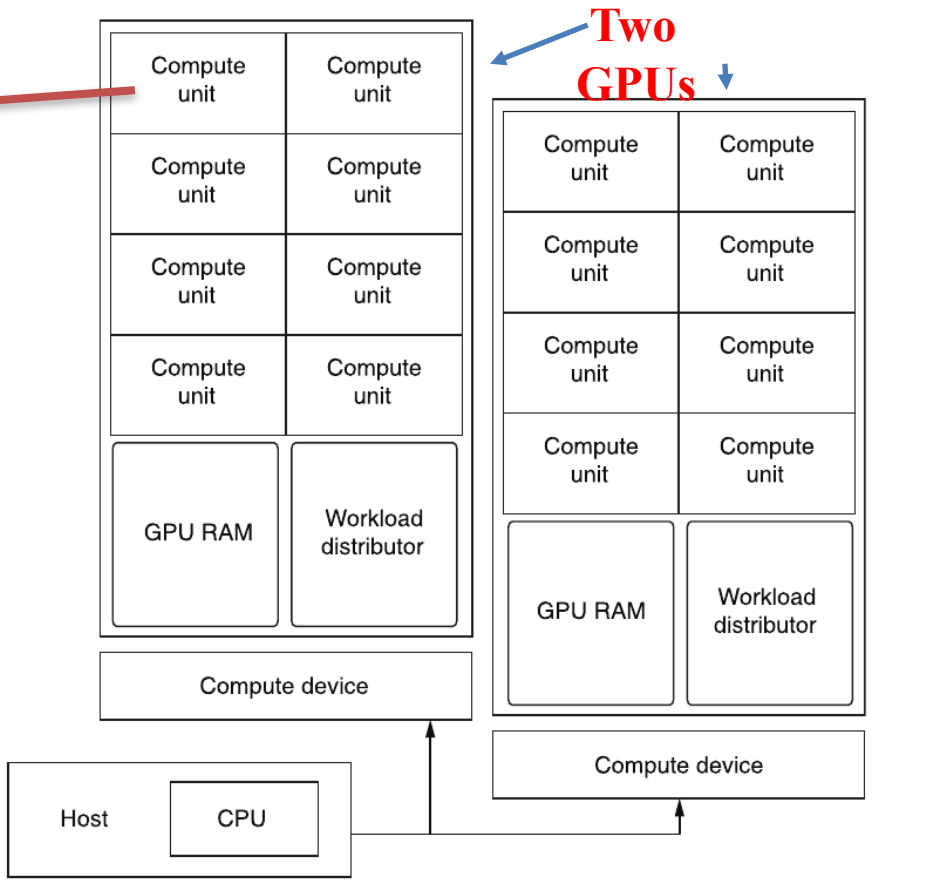
# A closer look, and some labels




- CPU memory levels
  - Registers
  - Cache memory
  - Main memory
  - Network memory
- GPU memory levels
  - Registers (GPU)
  - Local memory (GPU)
  - Global memory (GPU)
  - Main memory (HOST)
  - Network memory (REMOTE)
- The key to performance: Data handling and latency hiding



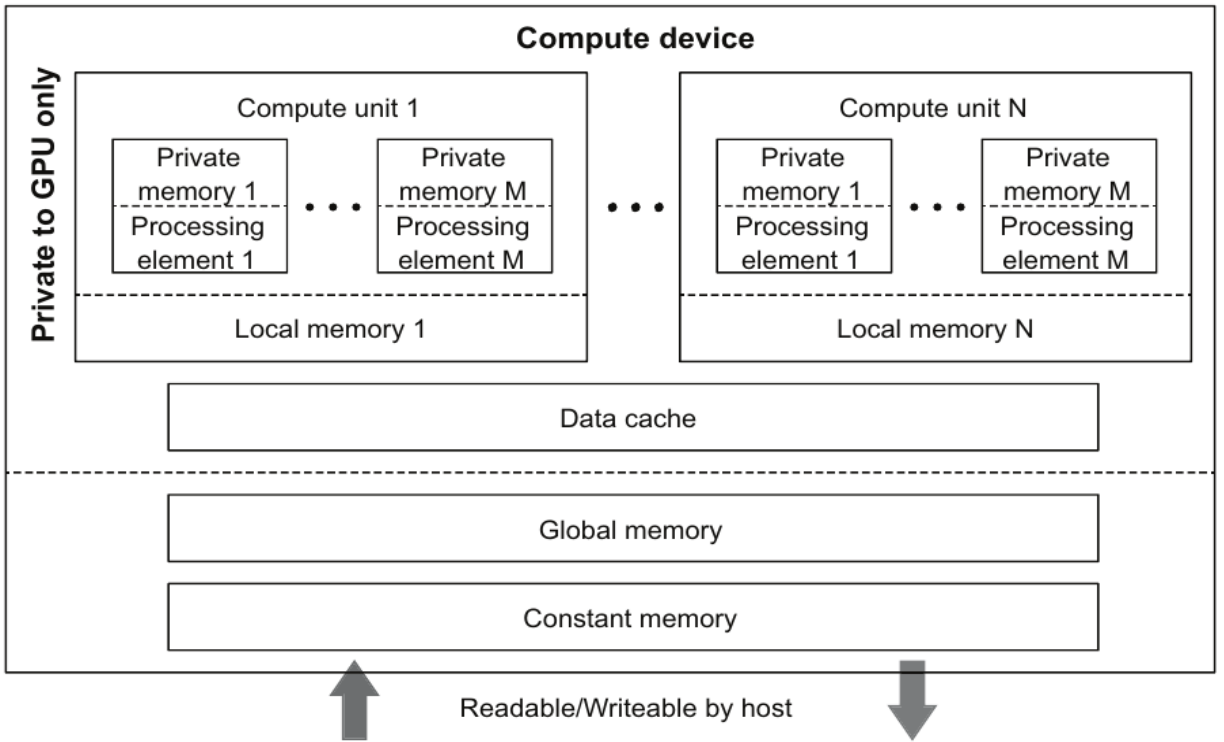

Simplified block diagram of
a compute unit (CU) with a large number
of processing elements (PEs).




A GPU compute device has multiple CUs. (CU, *compute unit*, is the term agreed to by the community for the OpenCL standard.) NVIDIA calls them *streaming multiprocessors* (SMs), and Intel refers to them as *subslices*.

Peak Theoretical Flops (GFlops/s) = Clock rate MHZ × Compute Units × Processing units × Flops/cycle

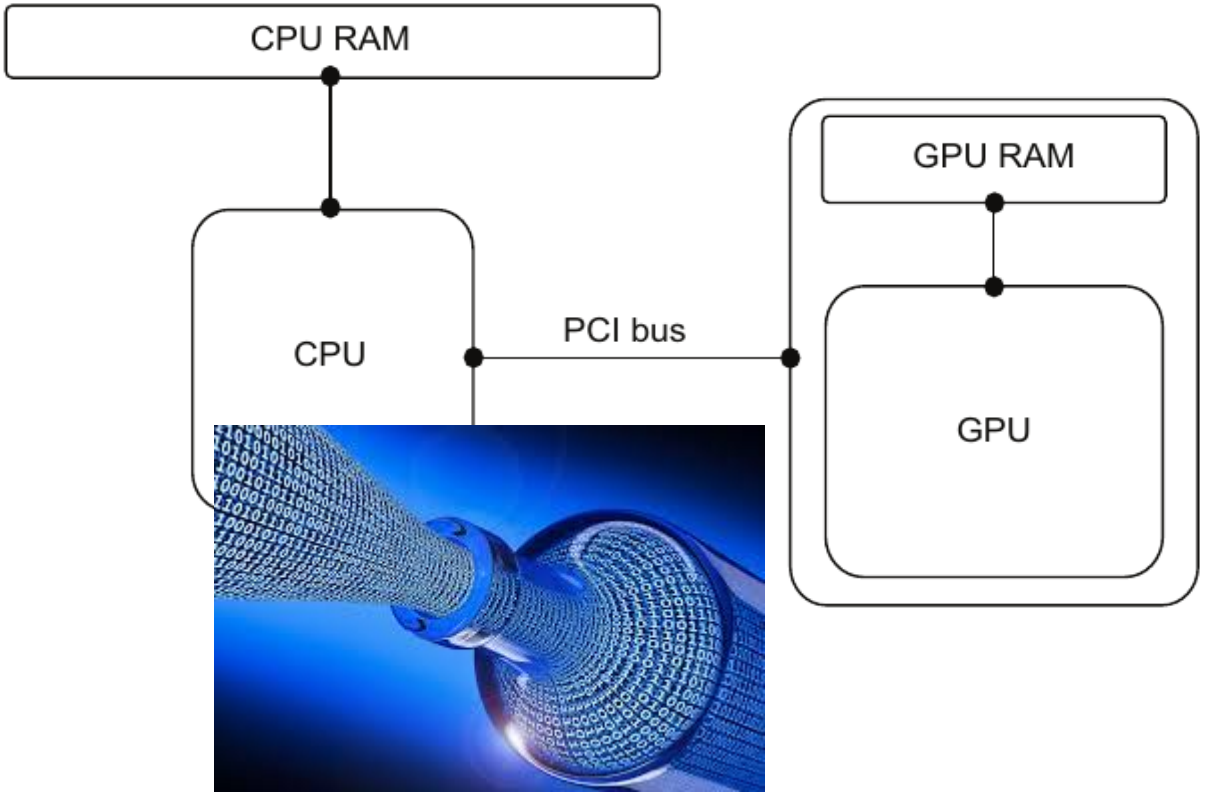
# The Memory




Specialized global memory with higher bandwidth than CPUs

Theoretical Bandwidth = Memory Clock Rate (GHz) × Memory bus (bits) × (1 byte/8 bits)

# Lastly: the Peripheral Component Interconnect (PCI Bus)

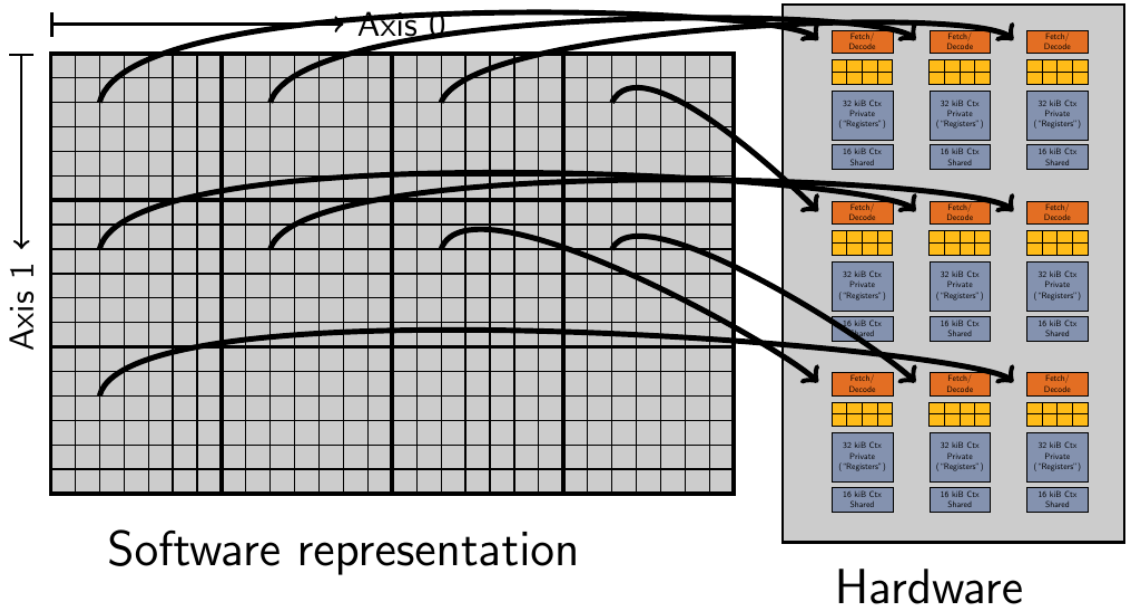



Theoretical Bandwidth (GB/s) = Lanes × TransferRate (GT/s) × OverheadFactor(Gb/GT) × byte/8 bits

# What we'll use today:


2 Nvidia A30, sliced into 12 vGPUs each, so there are
24 vGPUs with 6GB and 14 CU each. 80k dkk in total.

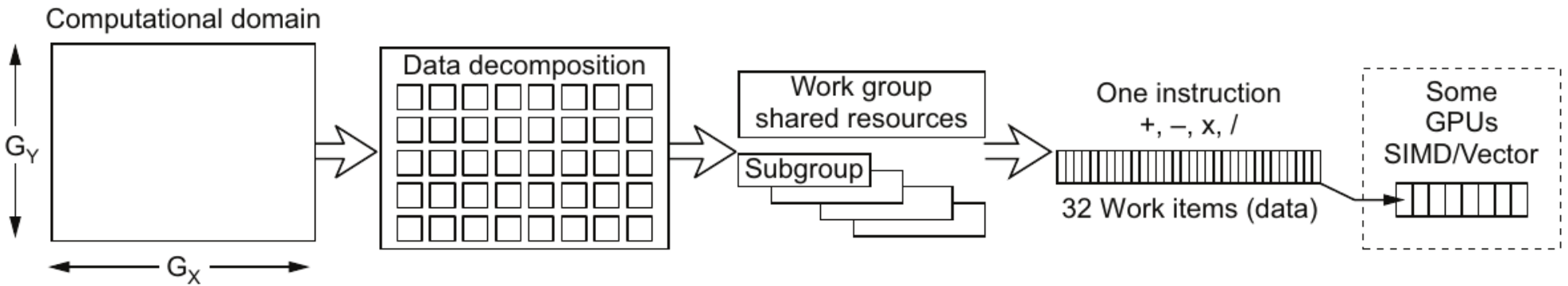
**… just for you, but please log-off if you don't use them ;-)**




- Program as if an infinite amount of cores
- Data centric programming model
- Extremely parallel programming model
  - One thread per data element

# GPU programming model

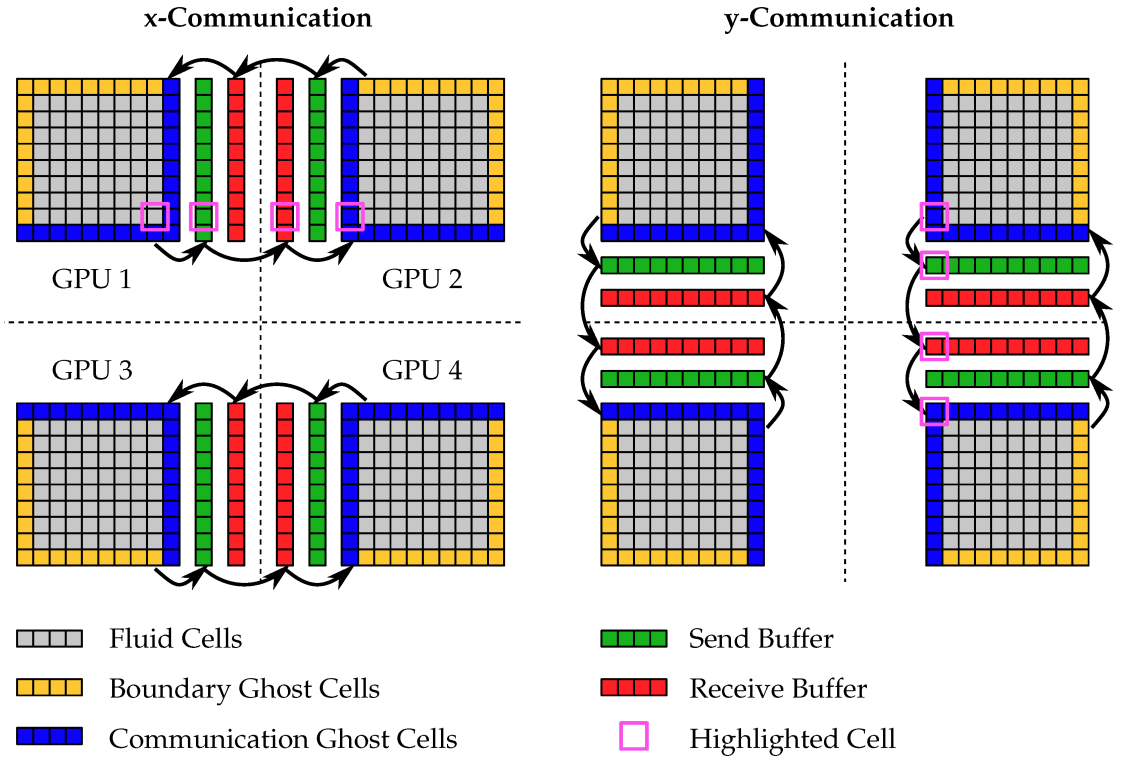
## Common programming abstractions across various GPU models.



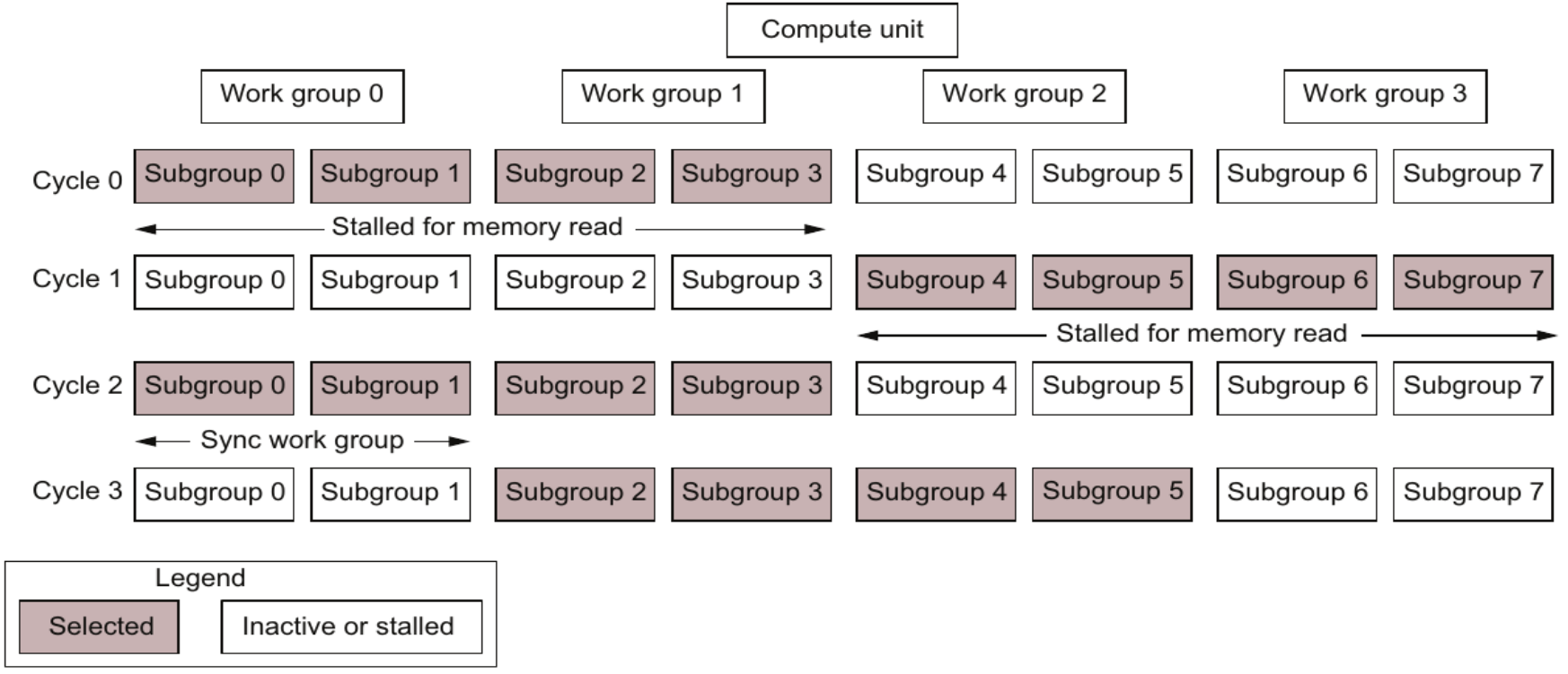

**Terminology can be confusing:**

| | | |
|---|---|---|
| **Ndrange** | **= grid** | **= loop bounds** |
| **Work group** | **= thread block** | **= loop block** |
| **Subgroup** | **= wavefront** | **= SIMD length** |
| **Work item** | **= thread** | |

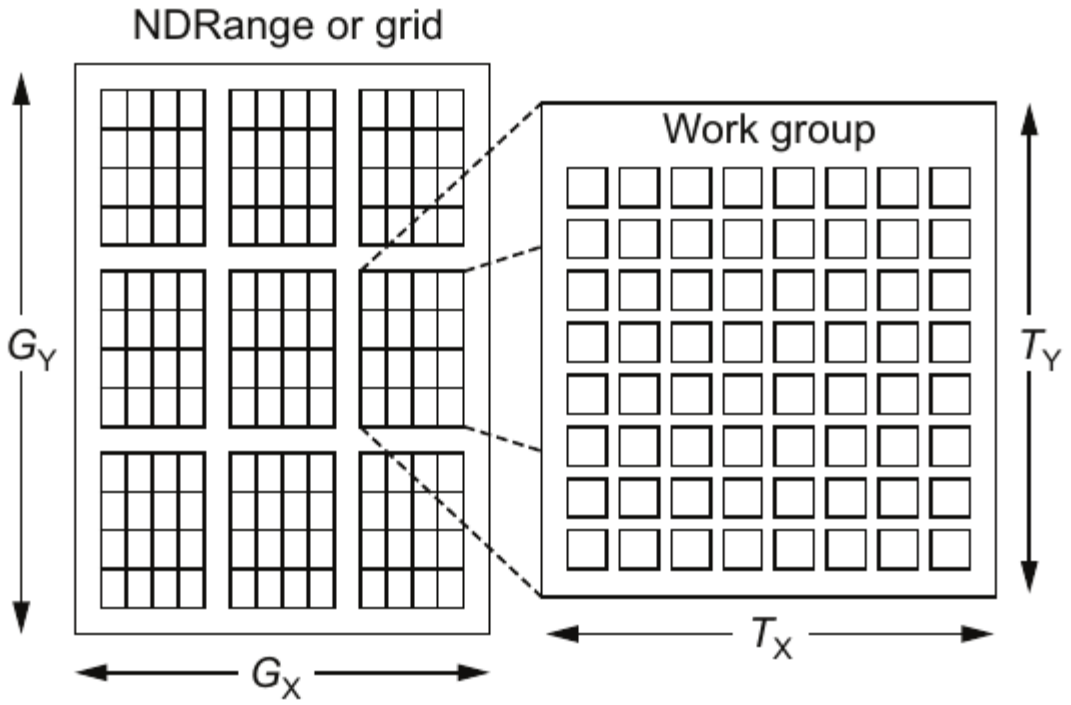
**Key: little coordination needed within operations**

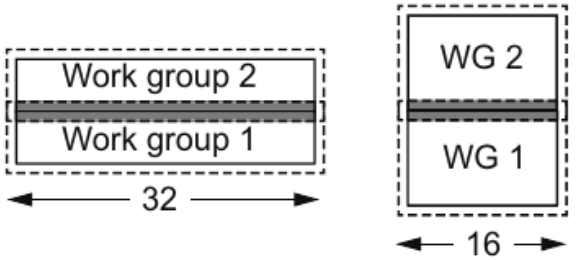



Communication for stencil operations through ghost cells




**Latency hiding/ reduces the need for deep memory hierarchy**




Shape is relevant for groups that need
Data from neighbour (minimize surface)

Each work group has access to
Global and constant memory and
Is independent and asynchronous.

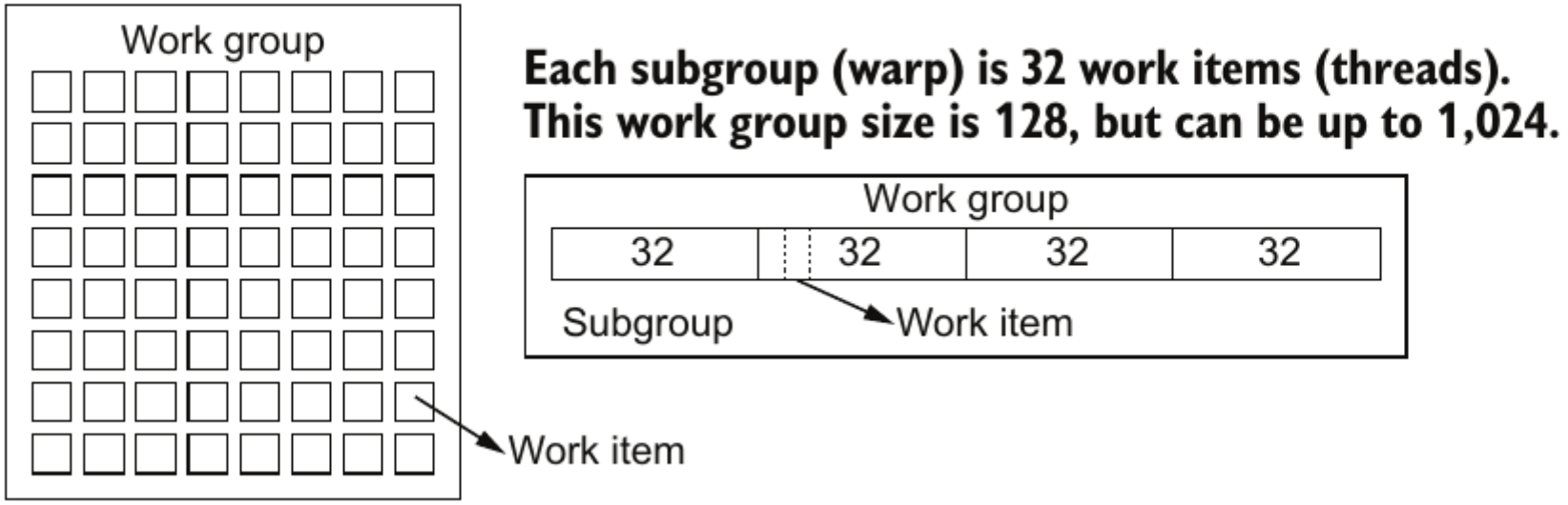
Data decomposition of a 1024×1024 2D computational domain

If we set a tile size of 16x8, the data decomposition will be as follows:

NTx = Gx/Tx = 1024/16 = 64

NTy = Gy/Ty = 1024/8 = 128

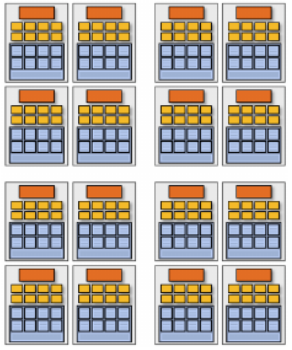
# Workgroups




The work groups spreads out the work across the threads on a compute unit and have access to local, shared memory.

… and finally, each work item may be able to do a vector or SIMD

- Do we care about the number of cores?
- Do we care about the number ALU's per core?



- NO !!!

  Program as if an infinitely amount of cores
  Program as if an infinitely amount of ALU's per core

# Practicalities

#pragma acc <directive> [clause]
#pragma omp <directive> [clause]

Many similarities between OpenACC and OpenMP, there attempts to merge both.

OpenMP (1997) long focused on CPUs.
OpenACC (2011) specifically designed to Handle GPUs.


It appears to me that in OpenACC begin
have less opportunities to make mistake
OpenMP experts have more control.

| Steps | CPU | GPU |
|---|---|---|
| Original | a[1000]<br>`for` or `do` loop<br>`for` or `do` loop<br>free a | |
| 1. Offload work to GPU | a[1000]<br>**#work pragma** →<br>← `for` or `do` loop (128 threads)<br>**#work pragma** →<br>← `for` or `do` loop (128 threads)<br>free a | |
| 2. Manage data movement to GPU | **#data pragma{**<br>#work pragma<br>#work pragma<br>**}** | a[1000]<br>`for` or `do` loop (128 threads)<br>`for` or `do` loop (128 threads)<br>free a |
| 3. Optimize GPU kernels | #data pragma{<br>#work pragma<br>#work pragma<br>} | a[1000]<br>`for` or `do` loop (**256 threads**)<br>`for` or `do` loop (**64 threads**)<br>free a |

# Benefits of using OpenACC

•achieve parallelization on GPUs without having to learn an accelerator language such as CUDA
•similar philosophy to OpenMP and very easy to learn

```
#pragma acc kernels
{
    for (int i = 0; i < N; i++) {
        y[i] = a * x[i] + y[i];
    }
}
```

```
#pragma acc parallel loop
{
    for (int i = 0; i < N; i++) {
        y[i] = a * x[i] + y[i];
    }
}
```

# Examples of some useful directives:

#pragma acc parallel [clause [[,] clause]…] new-line

#pragma acc parallel loop present(data)

#pragma acc parallel loop collapse(2) present(a, b)

**To move data around (sparingly ;-)**

#pragma acc data copy(myworld)

#pragma acc update self(myworld)

A full list of directives can be had from https://www.openacc.org/resources,
The reference card under Guides.


"That's all Folks!"