

# Introduction To GPUs

"The Angry Penguin", used under creative commons licence from Swantje Hess and Jannis Pohlmann.



# Why GPU?

- GPUs potentially have substantially higher computational capacity than CPUs
- Less true than it used to be for scientific calculations at high precision but still worthwhile
- NVIDIA H200 - 34TFlops
- 2x AMD 9755s - 9TFlops
- ~ 4 x difference

# Why Not GPU?

- GPUs are harder to program for than CPUs
- GPUs are very well suited for AI/Machine learning so they are often very heavily oversubscribed because that is a popular area at the moment
  - Because there is so much money in AI/ML newer GPUs optimise for the lower precision calculations that are needed there - performance is not really increasing for high precision numerical calculations
- GPU systems often have less memory than CPU systems
- We'll try to go in increasing complexity

# Massive Parallelism

- Earlier compared 256 CPU cores to 17,000 GPU cores and got ~4x performance difference
  - Each GPU core is much weaker than each CPU core
- **Have** to have work that can be split up thousands of ways or you can't get the performance
- Performance per core is going up a bit with GPU generation, but not much recently

# Separate Memory

- With a handful of recent, expensive exceptions GPUs have separate memory to the CPU of the computer they sit in
- The user directly interacts with the computer in CPU code
  - So do the hard disks etc.
- You have to have enough work to make it worth moving data from the CPU to the GPU and getting the result back again

# Specific Programming Model

- GPUs aren't programmed like CPUs
- CUDA - most popular but specific to NVIDIA GPUs
- HIP - AMD GPU library very like CUDA, specific to AMD GPUs
- Performance portability libraries - aim to allow you to write your code once will run on CPU/GPU automatically
- OpenMP - You can program GPUs with OpenMP using similar but different syntax to the CPU parallel code

# Kernels

- Except for OpenMP, the other GPU programming models are **kernel** models
- You write code for what happens to a given numbered data element and the code is automatically called on all of the data elements
- Important to note that you don't just write code for a single element, you are given/work out an index for which element you should be working on

# CUDA Kernel

```
__global__ void smoothKernel(float* input, float* output, int width, int height) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    //4 point average
    if (x >= 1 && x < width - 1 && y >= 1 && y < height - 1) {
        float sum = 0.0f;
        sum += input[(y - 1) * width + x]; // -1,0
        sum += input[y * width + (x - 1)]; // 0,-1
        sum += input[y * width + (x + 1)]; // 0,+1
        sum += input[(y + 1) * width + x]; // +1,0
        output[y * width + x] = sum / 4.0f;
    }
}
```

- This is a simple CUDA kernel to smooth an image
- It shows an important element of CUDA
  - Processing is split up into blocks and threads

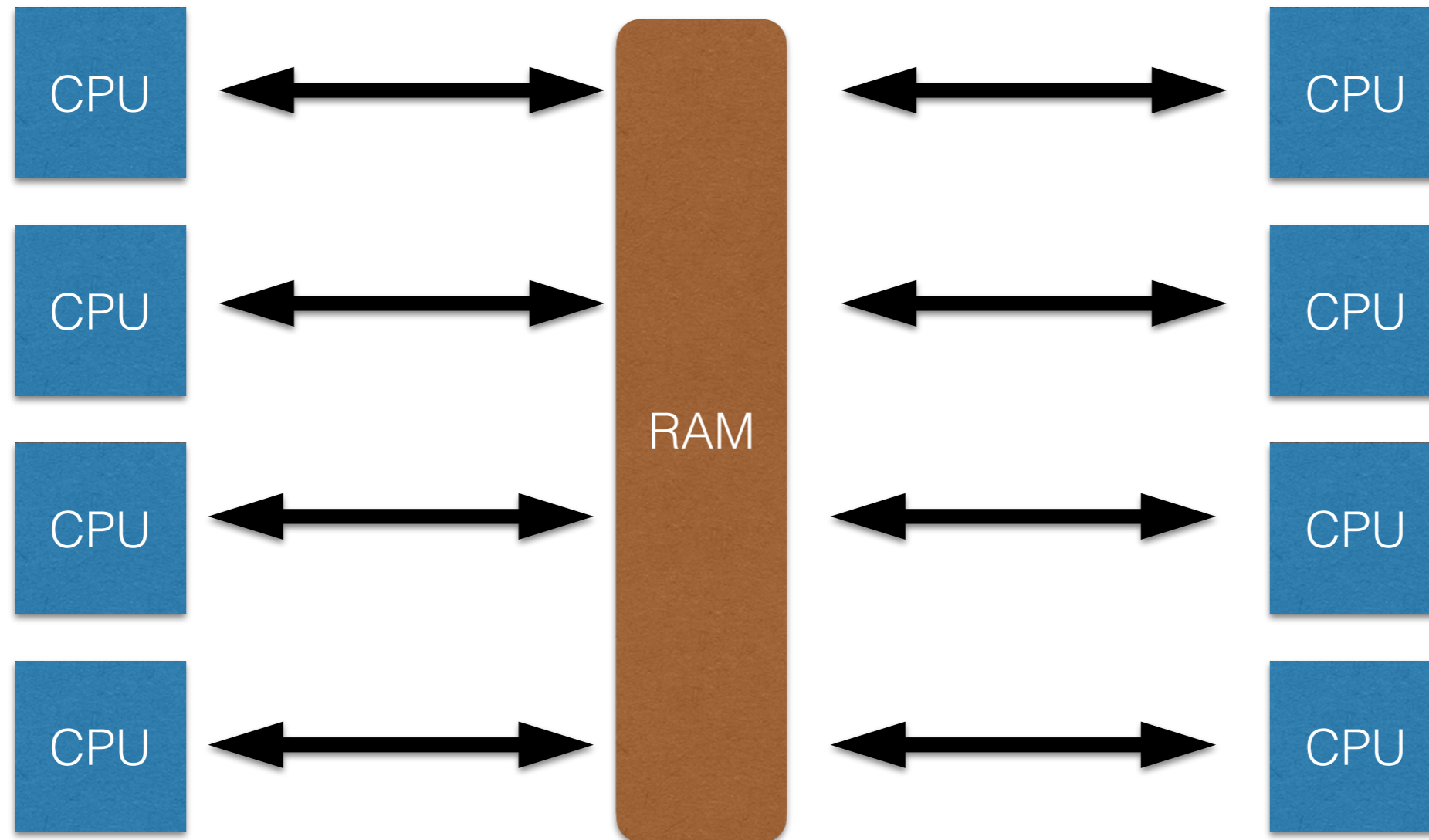
# Blocks and Threads

- This terminology is NVIDIA/CUDA specific, but the idea is common to all native GPU programming models
- Threads are like threads in a CPU - they execute instructions "independently"
- Blocks are groups of threads that share resources
  - You can chose how many threads you want in a block and how many blocks you want to cover your data
    - Important to get it right for performance

# Shared Resources

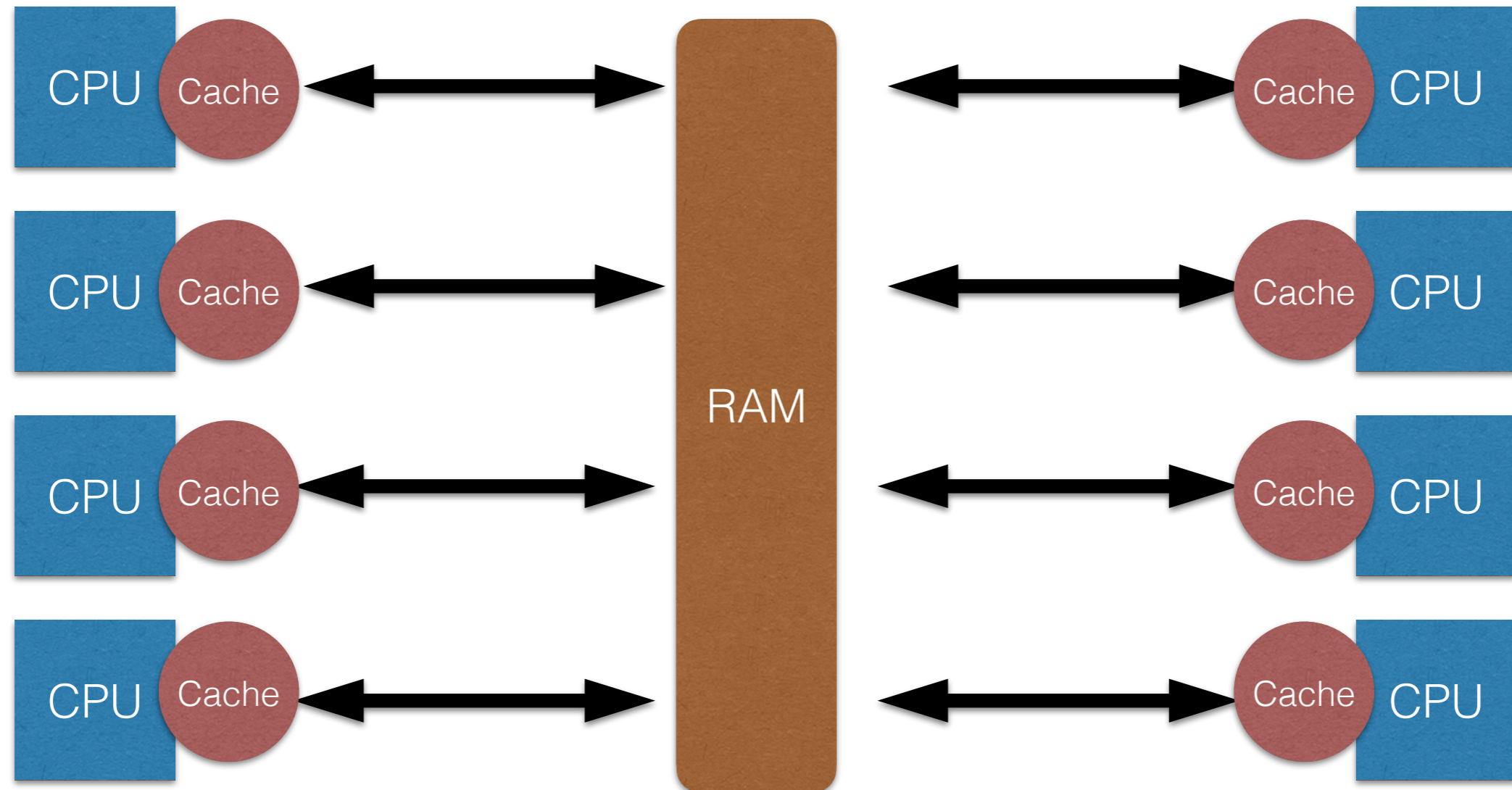
- Blocks are “threads that share resources”
  - What does that mean, and why is it important?
- Have to delve a bit into how computer memory works
- In particular, no modern computer can keep it's CPU (or GPU) fed with data for full speed computation
- If you do one operation on a piece of data then move onto the next piece your CPU will have to wait for that data to come from memory
- If you do enough computation per piece then this doesn't matter, but to improve performance in other cases, there are hardware tricks to try and mask the problem

# CPU Cache Memory



- Saw this model for "shared memory" earlier
- It is missing an important step

# CPU Cache Memory

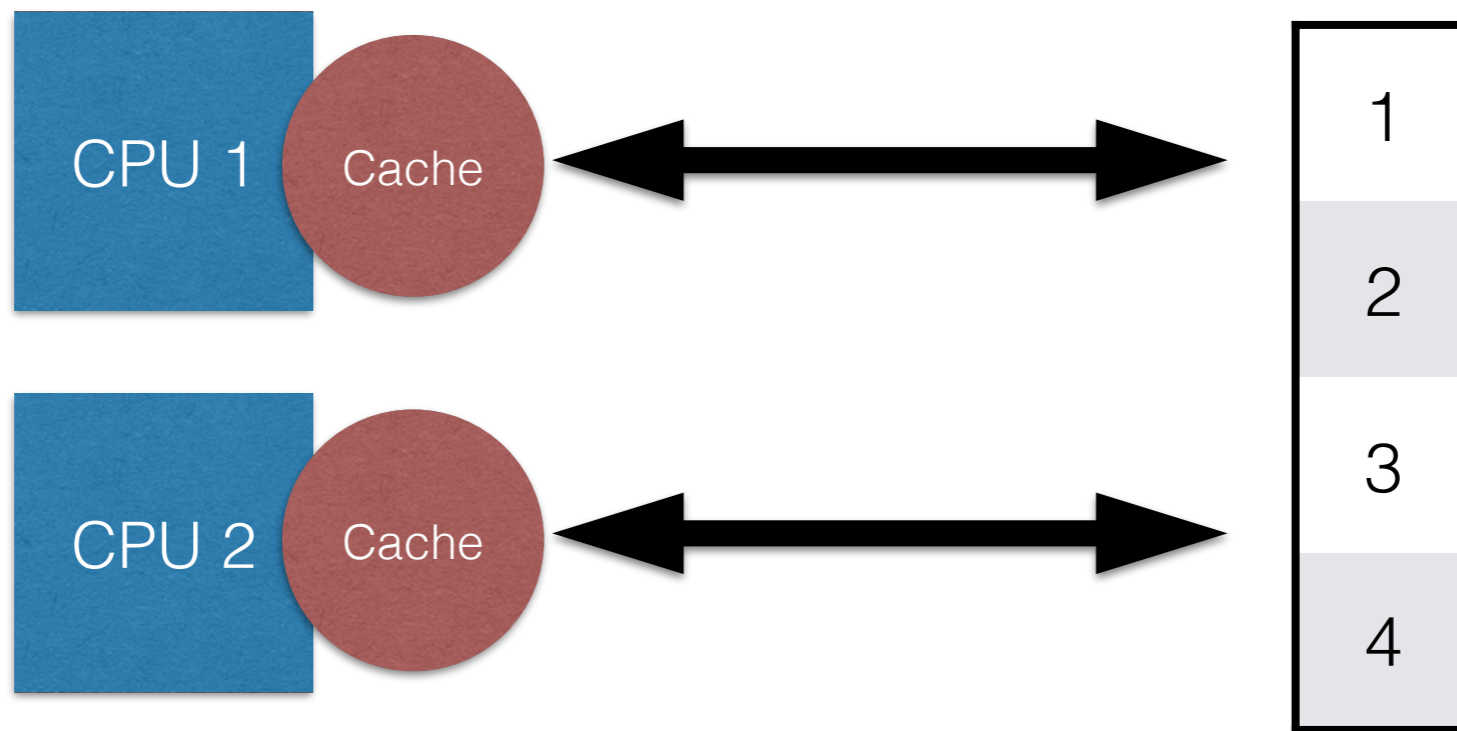


- Each CPU has a small "cache" of very fast memory that contains data that has either been used recently or was "near" memory that was used recently

# CPU Cache Memory

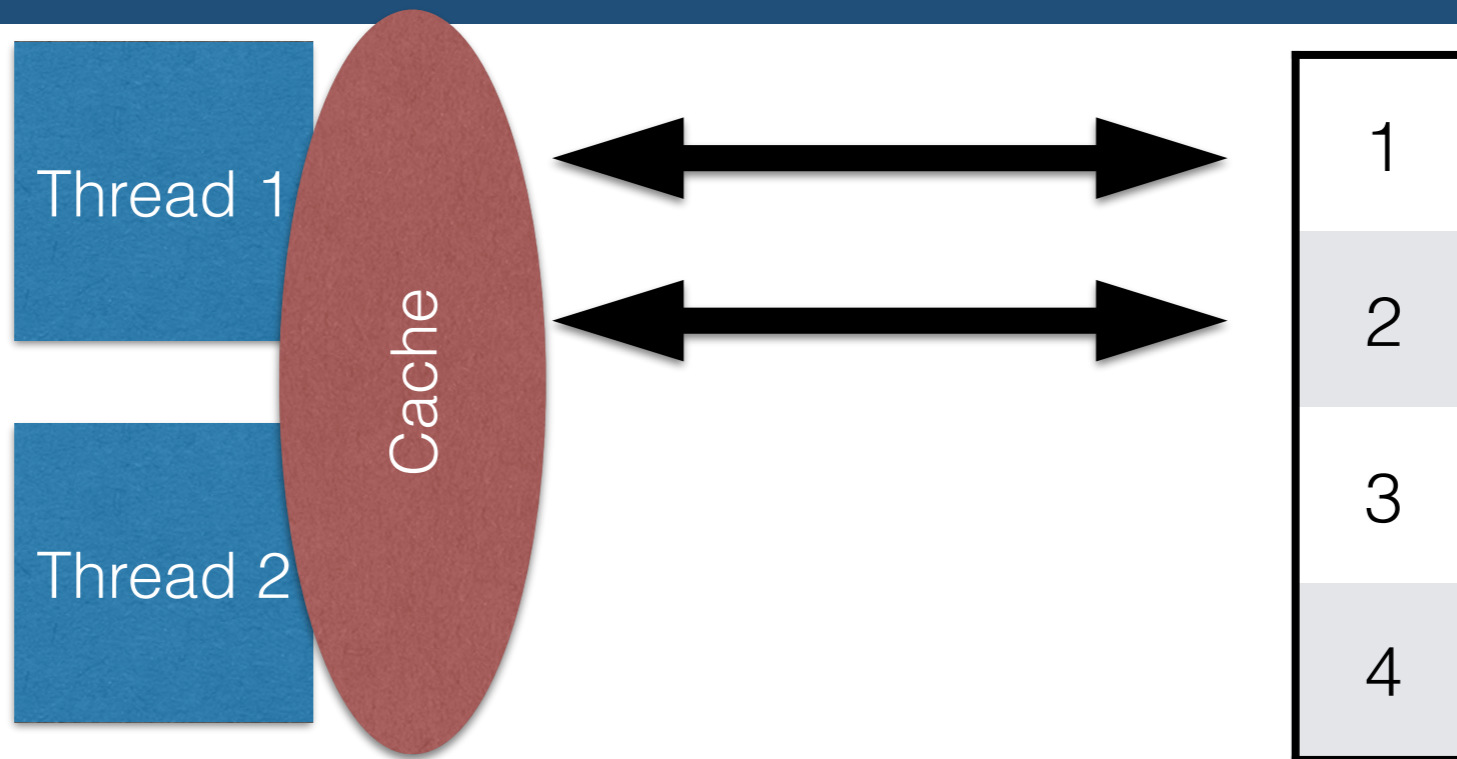
- The idea of cache memory is to try and “guess” what data you will want in advance and already have it stored in the cache memory
- The name comes from the fact that it keeps recently used data
- Also, it tries to **prefetch**
  - When accessing an array element then further elements of the array are pulled into memory as well on the basis that you might need them
- If you use memory predictably enough then this can hide some of the problems with transferring the data from memory

# CPU Cache Memory



- Consider the above simplified example of two processors
- When CPU 1 loads data item 1, it will also pull in data item 2
- If CPU 2 works on data item 2 it will have to be loaded again to go into CPU 2's cache
- Better performance if CPU 1 works on data items 1&2, and CPU 2 works on data items 3&4
- This is **massively** simplified compared to reality because there are also shared caches between CPUs as well as per CPU caches, but it shows the general approach

# GPU Cache Memory



- On a GPU the resources shared between threads in a block include the cache
- This means that you can have adjacent threads working on adjacent data items
- Combine this with the fact that some collections of threads have to do the same thing at the same time and you generally **want** to have this type of operation which is called **coalesced** operation in GPU terminology

# GPU Cache Memory

- In fact, GPUs are even harder to keep provided with data than CPUs are
- Most of GPU programming is making sure that the data that you want is available when you want it
- There are quite complex ideas about memory at advanced levels of which the most commonly encountered is
- Global memory - shared memory between threads in a block. Like cache, but you decide what goes into it rather than the hardware guessing. Very small, few 10s of **kB** in size
  - Don't confuse it with **device memory** which is shared between **all** blocks and is the headline memory on your GPU, so GB in size
- You will also generally have to write code that performs calculations on some data while other data is being moved around in memory

# GPU-Like things

- A few years ago, there was a belief that there were going to be dozens of type of **accelerator**, of which GPUs were only one type
- Almost all of them except GPUs have vanished
- The remaining one is the **Field Programmable Gate Array (FPGA)**
  - FPGAs are arrays of logic gates that can be electronically programmed
- They are rather like creating a custom processing chip, but can be reprogrammed again and again

# GPU-Like things

- “Proper” FPGA programming uses special languages like Verilog and HDL
- You can use some parallel programming tools, like OpenMP and the performance portability libraries to develop for FPGAs
- For the right problems, they are the fastest you can get and need much less power than either GPUs or CPUs
- Mostly those problems are things like signal processing
- For general computing tasks, the performance of FPGAs is often underwhelming
  - Probably not worth the effort to learn to program for them properly

# Multi GPU

- You can write code that spans multiple GPUs, but it isn't especially easy
- Simplest level - multiple problems, each running on separate GPUs in a single machine
- Intermediate level - Using libraries like NVIDIA's P2P (peer to peer) to share data between GPUs
- Hardest level - Using tools like MPI (again!) or NVSHMEM to work between GPUs in multiple computers

# Massive Hybrid

- If you want to go truly mad, you can combine **everything** that we have talked about
- You can write codes that use **OpenMP** for parallelism between CPUs on a single computer, **CUDA** or another library for GPU acceleration of parts of the computation and **MPI** for communication between computers of both CPU and GPU data
  - There are some clever hardware features that mean that data can be copied by MPI between GPUs without involving the CPU
- Very, very few codes like this exist and they are mostly intended for exascale computation