High Performance Computing

Term 4 2018/2019

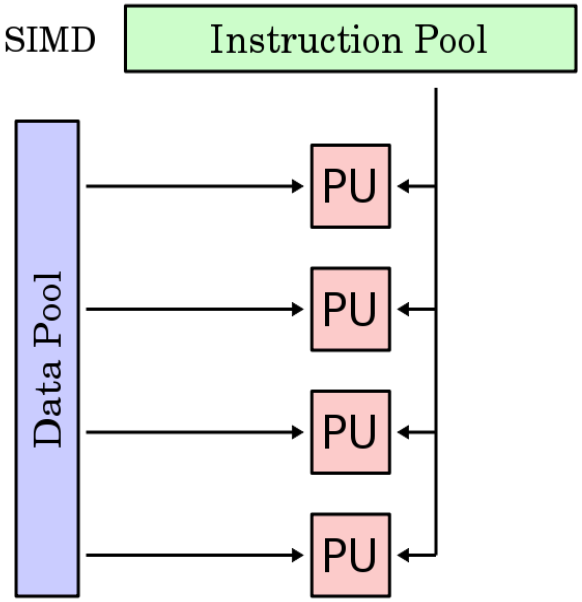
Lecture 7

What we have learned so far...

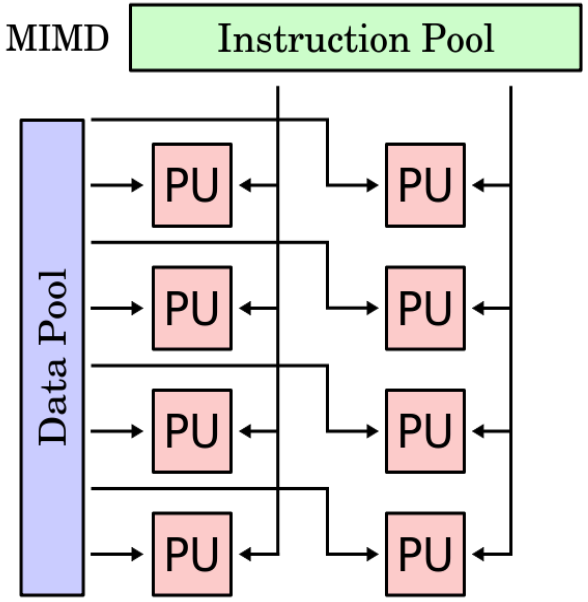
- Amdahl‘s law
- Shared memory vs distributed memory systems
- Flynn‘s taxonomy
- OpenMP compiler directives for shared systems
- Message Passing Interface for distributed systems
- MPI for Python (mpi4py)
- MPI + OpenMP hybrid coding
- You have accomplished a pretty complex task of Schelling model (just imagine people are publishing similar ideas in journals)!

Flynn's taxonomy

GPU



CPU



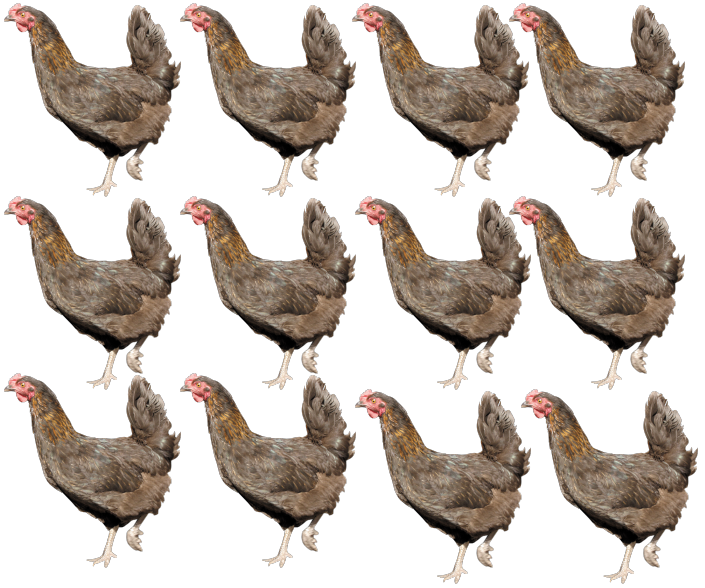
Why GPUs?

Chickens vs oxen

Seymour Roger Cray

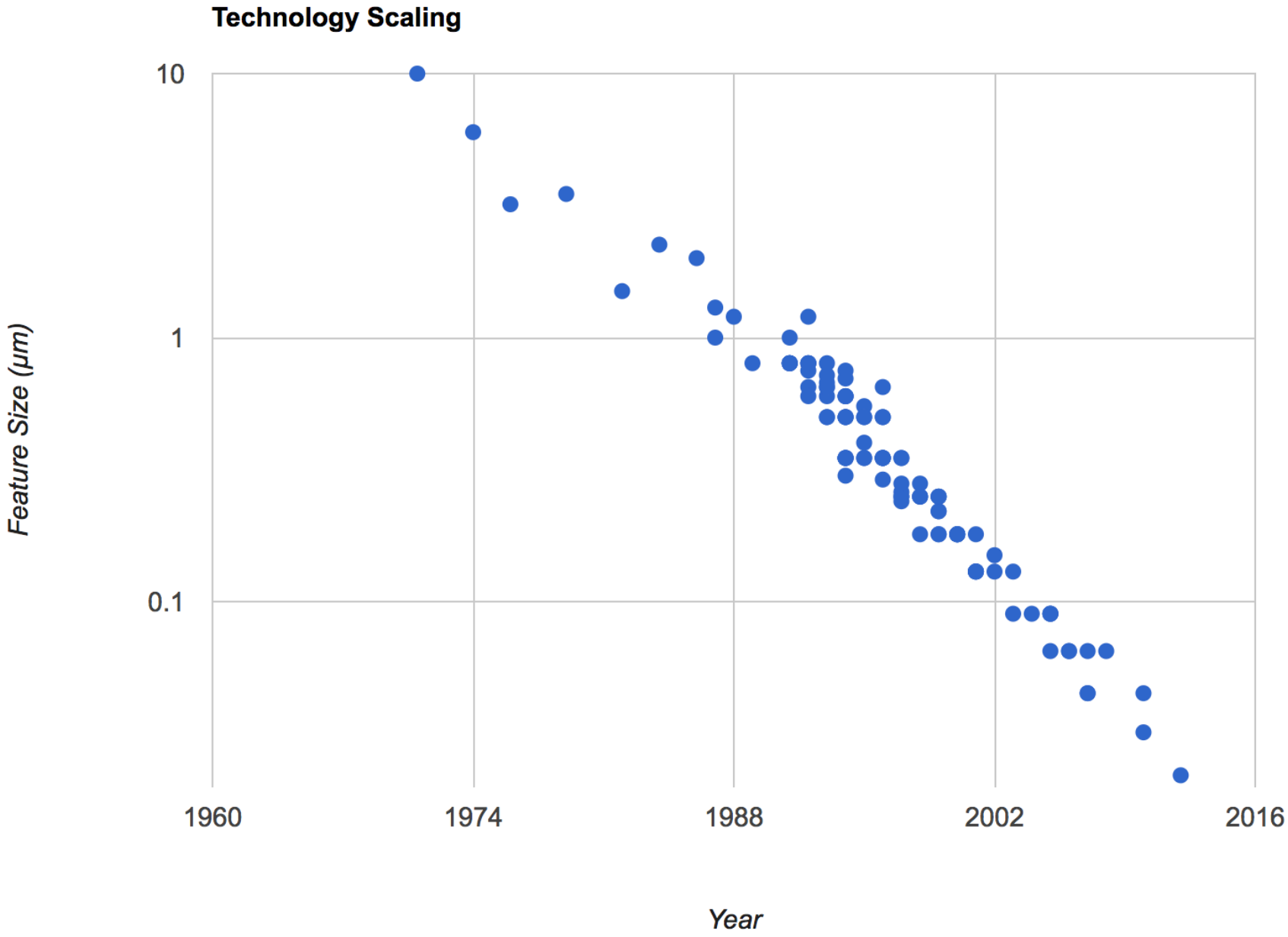
"If you were plowing a field, which would you rather use: two strong oxen or 1024 chickens?"

Modern CPU
~10 cores

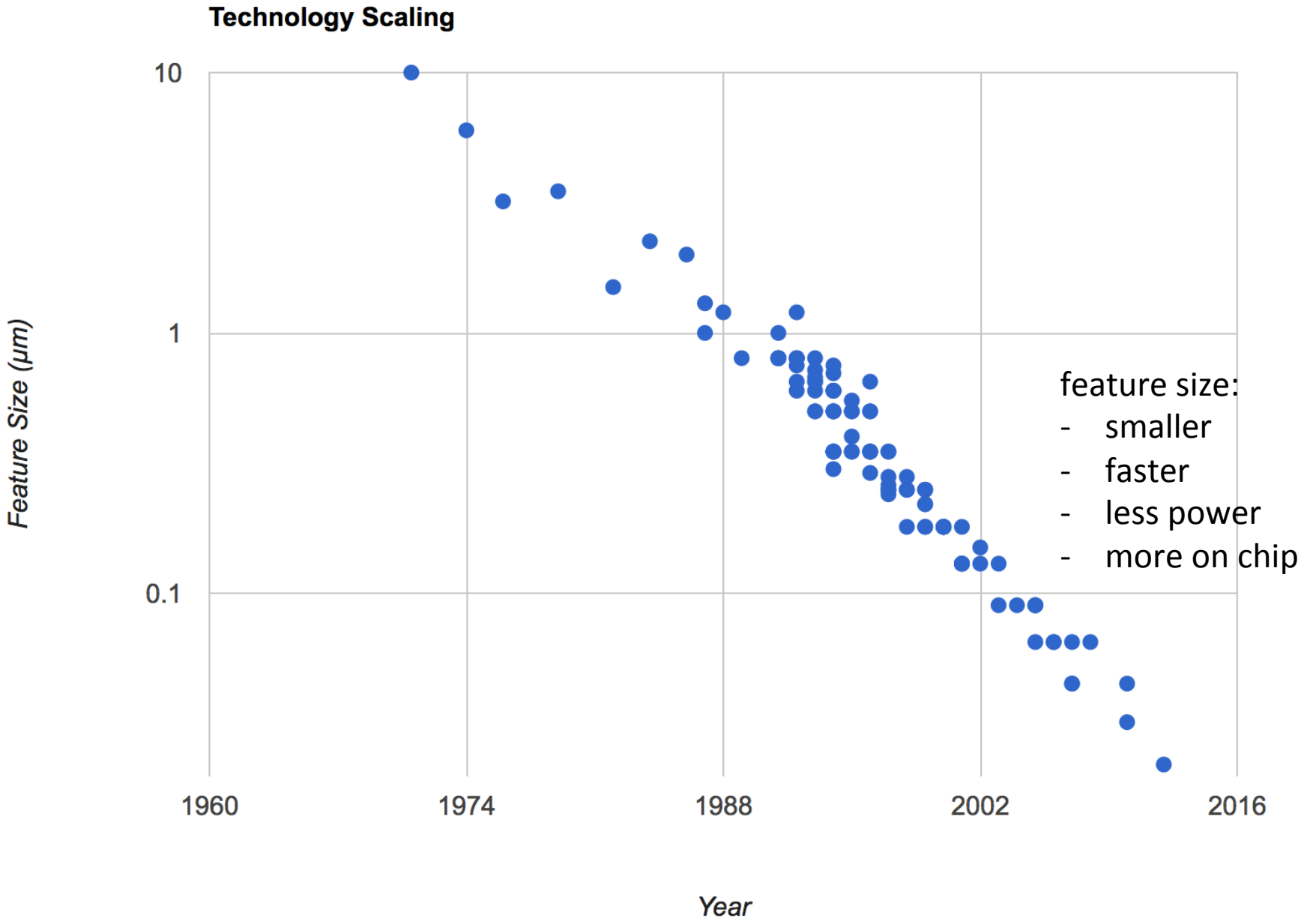


Modern GPU
~5000 cores

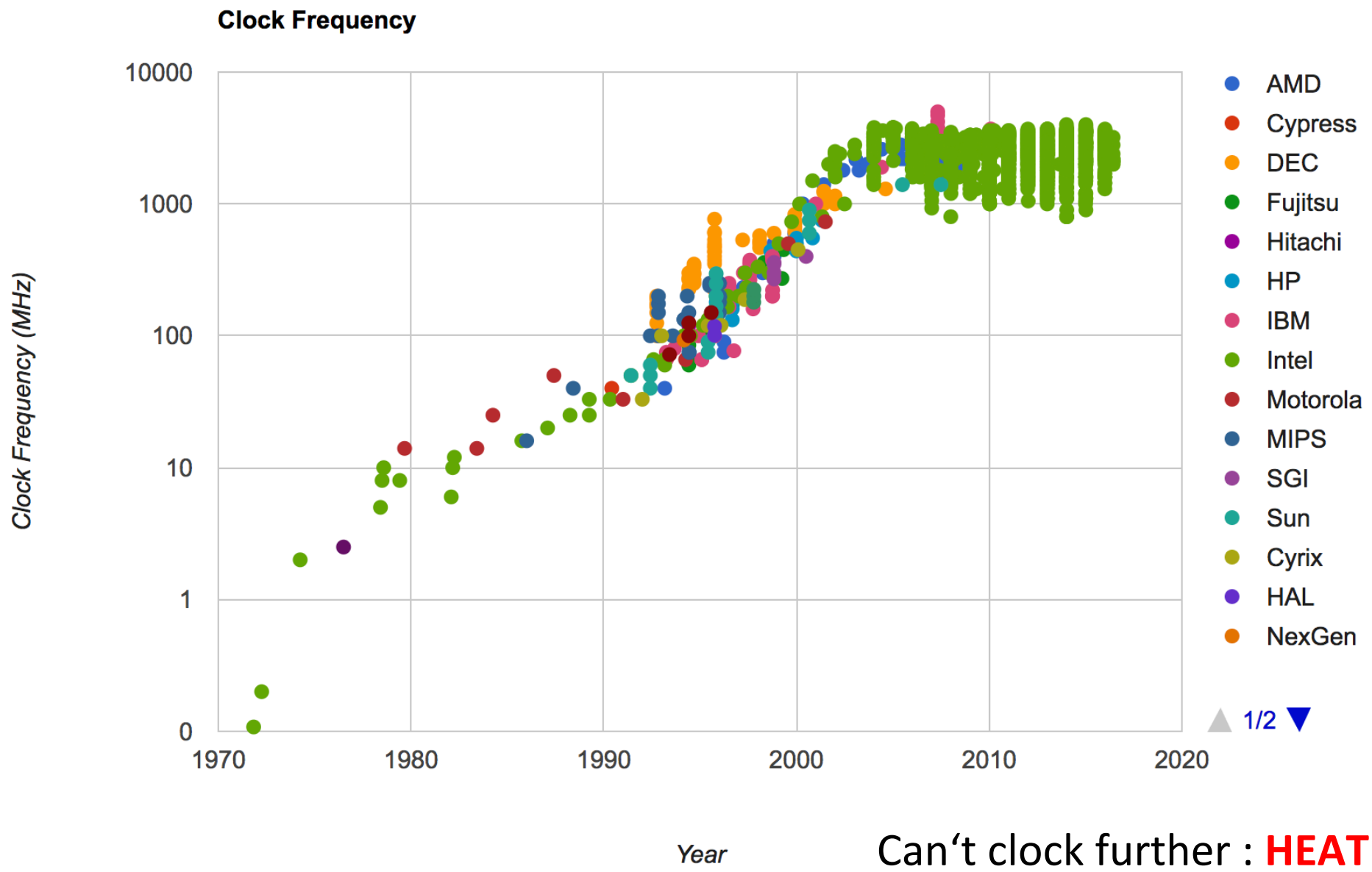
CPU Database (cpudb.stanford.edu)



CPU Database (cpudb.stanford.edu)



CPU Database (cpudb.stanford.edu)



Can‘t clock further : **HEAT**

Power is the new constraint

Consequence:
- Better to build smaller more efficient processors (in terms of power)
- More of them

What processors can we build if power is the main constraint?

Modern **CPUs** are great all-around processors capable of doing all sorts of things:
+ flexibility + performance
- power consumption (expensive flops/watt)

Modern **GPUs** have simpler control structure (simpler processing units)
+ simpler processing units (many of those)
+ more power efficient (more flops/watt)
- you have to think how to program those

Latency vs Throughput

Latency vs Throughput

Latency	vs	Throughput
(time)		(tasks/time)
(seconds)		(flops)

Supermarket

Latency vs Throughput

CPU design -- design from the point of view of customers (want to get out of supermarket as fast as possible, even though the cashier will not have any tasks to do afterwards, optimize for latency)

GPU design − design from the point of view of the supermarket owner (cashiers are getting paid for the whole day − they better have constant work, optimize for throughput)

Latency vs Throughput

200 km/hour
2 people

Latency =

Throughput =

50 km/hour
40 people

Latency =

Throughput =

Latency vs Throughput

200 km/hour
2 people

Latency = 4 hours

Throughput = 0.5 people/hour

50 km/hour
40 people

Latency = 16 hours

Throughput = 2.5 people / hour

Core GPU design tenets

1. Lots of simple compute units that trade simple control for more compute;
2. Explicitly parallel programming model;
3. Optimized for throughput not latency

NVIDIA CUDA

1. CUDA is a parallel computing platform and application programming interface (API) that can be used on NVIDIA GPU cards.
2. Proprietary with closed source compilers (unlike OpenCL).

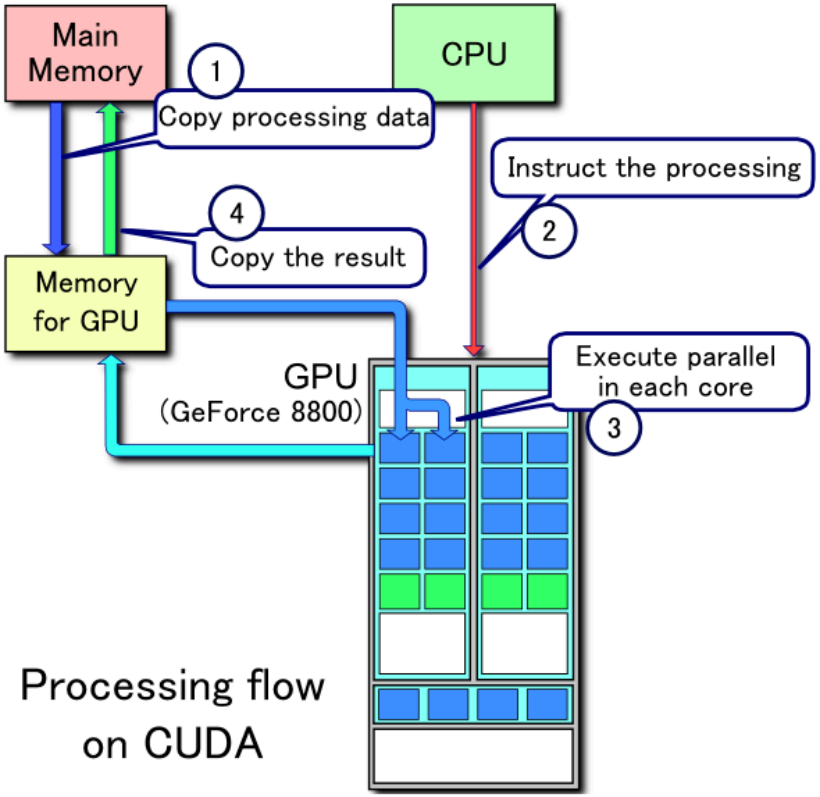
CUDA 8.0:

- CUBLAS - CUDA Basic Linear Algebra Subroutines library, see main and docs
- CUDART - CUDA RunTime library, see docs
- CUFFT - CUDA Fast Fourier Transform library, see main and docs
- CURAND - CUDA Random Number Generation library, see main and docs
- CUSOLVER - CUDA based collection of dense and sparse direct solvers, see main and docs
- CUSPARSE - CUDA Sparse Matrix library, see main and docs
- NPP - NVIDIA Performance Primitives library, see main and docs
- NVGRAPH - NVIDIA Graph Analytics library, see main and docs
- NVML - NVIDIA Management Library, see main and docs
- NVRTC - NVIDIA RunTime Compilation library for CUDA C++, see docs

CUDA 9.0:

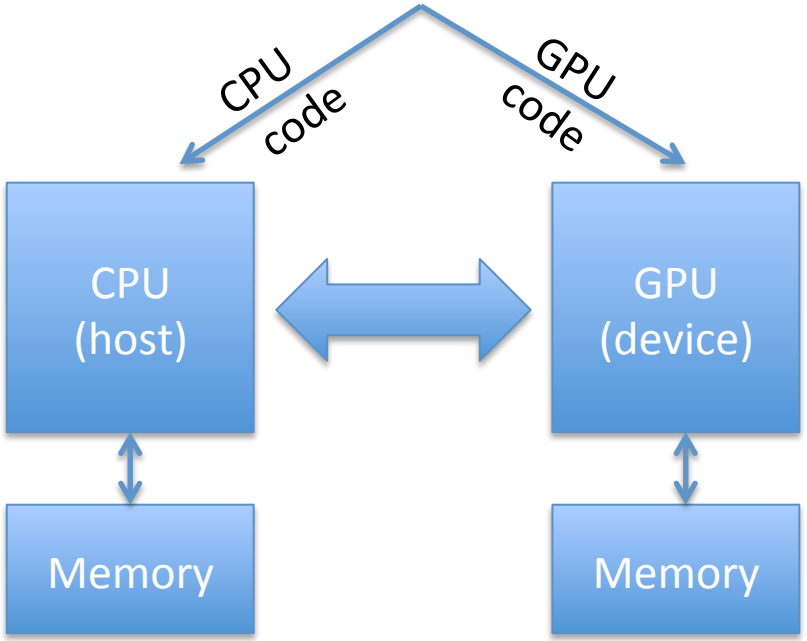
- CUTLASS 1.0 - custom linear algebra algorithms, see CUDA 9.2 News, Developer News and dev blog

CUDA workflow and conventions



Processing flow on CUDA

CUDA program
written in C/C++ with extension



CPU is the master and is responsible for:

1) moving data from CPU mem to GPU mem
2) moving data from GPU mem to CPU mem
3) allocating memory on the GPU
4) launching kernels on GPU („Host launches kernels on the device“)

(1,2 – cudaMemcpy; 3 – cudaMalloc)

Typical CUDA program

1. CPU allocates and initializes storage on CPU (malloc)
2. CPU allocates storage on GPU (cudaMalloc)
3. CPU copies data to GPU (cudaMemcpy)
4. CPU launches kernels on GPU (we'll learn how)
5. CPU copies data back from GPU (cudaMemcpy)

Remember: Try to minimize copying between the CPU and GPU!!!

Core concept of CUDA

Come back to this slide after doing examples: make sure you understand this

1. Kernels look like serial programs
2. Write your program as if it will run on one thread
3. The GPU will run that program on many threads (you tell the GPU how many threads to launch)

GPUs are good at:

1. Efficiently launching lots of threads (you can tell the GPU to run 10000000 of threads)
2. Running lots of threads in parallel (GPU has a maximum of threads it can run at once)

Simple example

```
In:   float array [0 1 2 3 ... 63]
Out:  float array [0 1 4 9 ... 63*63]

Kernel: square
```

CPU code (boring):

```
for(int i=0; i<64; ++i)
{
   out[i]=in[i]*in[i];
}
```

Simple example

```
In:   float array [0 1 2 3 ... 63]
Out:  float array [0 1 4 9 ... 63*63]

Kernel: square
```

```
CPU code (boring):
for(int i=0; i<64; ++i)
{
    out[i]=in[i]*in[i];
}
```

CPU code:
allocate memory
copy data to GPU
launch kernel

here we explicitly express parallelism (tell how many threads)

GPU code:
express out=in*in

here we say nothing about parallelism

Simple example

```
In:   float array [0 1 2 3 ... 63]
Out:  float array [0 1 4 9 ... 63*63]

Kernel: square
```

```
CPU code (boring):
for(int i=0; i<64; ++i)
{
    out[i]=in[i]*in[i];
}
```

CPU code:
allocate memory
copy data to GPU
launch kernel

here we explicitly express parallelism (tell how many threads)

GPU code:
express out=in*in

here we say nothing about parallelism

```
CPU code will look like:
squarekernel<<<64>>>(outArray, inArray)
```

Simple example

```
In:   float array [0 1 2 3 ... 63]
Out:  float array [0 1 4 9 ... 63*63]

Kernel: square
```

```
CPU code (boring):
for(int i=0; i<64; ++i)
{
    out[i]=in[i]*in[i];
}
```

CPU code:
allocate memory
copy data to GPU
launch kernel

here we explicitly express parallelism (tell how many threads)

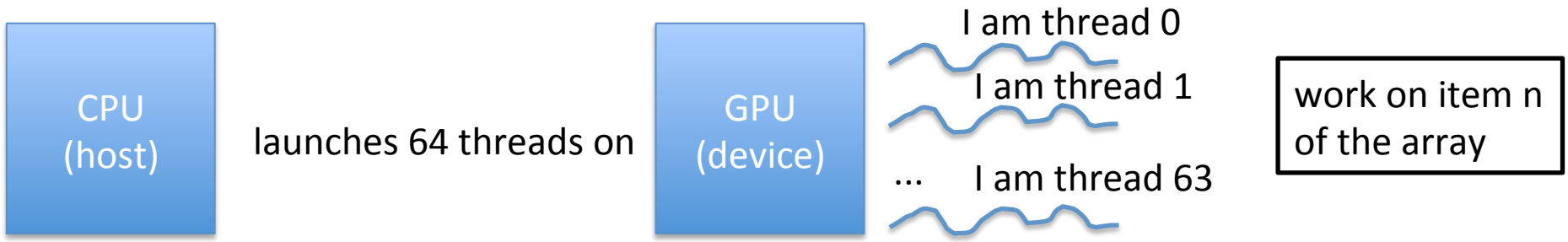
GPU code:
express out=in*in

here we say nothing about parallelism

```
CPU code will look like:
squarekernel<<<64>>>(outArray, inArray)
```

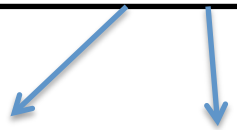
How does it work?

```
CPU code will look like:
squarekernel<<<64>>>(outArray, inArray)
```



Configuring the kernel call

```
square<<<1,64>>>(d_out, d_in)
```



1 block of 64 threads

GPU:

1. Can run many blocks at once;
2. Maximum number of threads/block - 1024 (on most GPUs, 512 on older ones)

If we want to launch 128 threads:

```
square<<<1,128>>>(d_out, d_in)
```

If we want to launch 1280 threads:

```
square<<<10,128>>>(d_out, d_in)
```

```
square<<<5,256>>>(d_out, d_in)
```

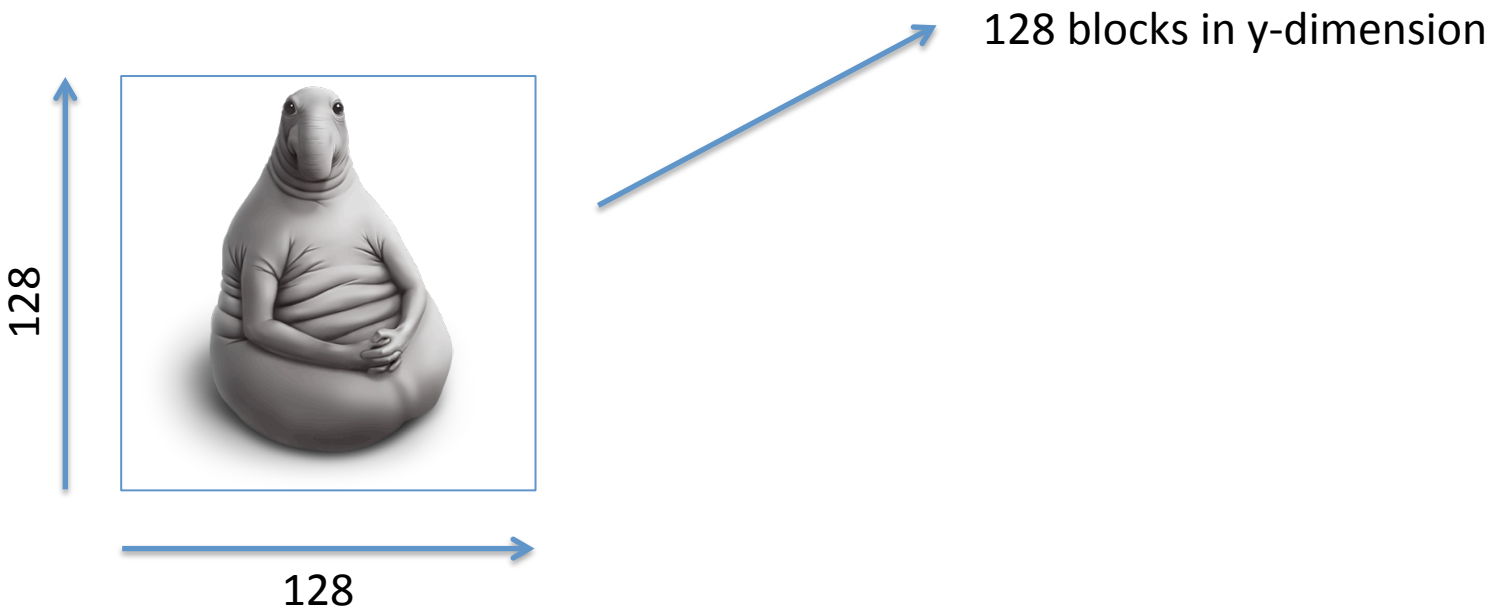
```
square<<<1,1280>>>(d_out, d_in)
```

???

Each thread knows its index within the block + index of the block. We can also do 2D and 3D mapping of blocks and threads.

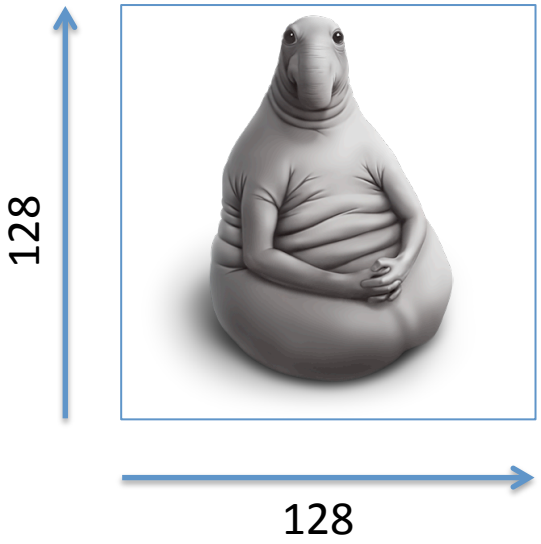
Configuring the kernel call

Image processing: one thread per pixel



Configuring the kernel call

Image processing: one thread per pixel



128 blocks in y-dimension
128x1 block of threads in x dimension

8x8 grid of blocks
each block is 16x16 threads

kernel<<<GRID OF BLOCKS, BLOCK OF THREADS>>>(...)

1d,2d or 3d

1d,2d or 3d

dim3(x,y,z)

dim3(w,1,1)=dim3(w)=w

Configuring the kernel call

General call:

kernel<<<dim3(bx,by,bz),dim3(tx,ty,tz)>>>(...)

Useful:

threadIdx.x - thread index within block
threadIdx.y
threadIdx.z

blockDim - size of a block (blockDim.x, blockDim.y, blockDim.z)
blockIdx − block within grid (blockIdx.x, blockIdx.y, blockIdx.z)
gridDim − size of a grid (gridDim.x, gridDim.y, gridDim.z)

kernel<<<dim3(8,4,2), dim3(16,16)>>>(...)

How many blocks?
How many threads per block?
Total threads?

Map parallel abstraction

MAP:

- set of elements to process [64 floats]
- function to run on each element [“square”]

MAP(elements, function)

GPUs are good at MAP:

- GPUs have many processors
- GPUs optimize for throughput

MAP communication pattern: