GPU Architectures

Francesco Andreucci, SISSA

ICTP CIFRA Magurele School, 3/07/2025

MAX

LIGHTHOUSE CODES

- QUANTUMESPRESSO
- Yambo
- siesta™
- www.flapw.de Fleur
- BigDFT

DOMAIN EXPERTS & CODE DEVELOPERS

- Consiglio Nazionale delle Ricerche
- SISSA
- ICN2 Institut Català de Nanociència i Nanotecnologia
- JÜLICH Forschungszentrum
- cea
- Universität Bremen
- CSIC Consejo Superior de Investigaciones Científicas
- UNIMORE Università degli Studi di Modena e Reggio Emilia

HPC EXPERTS & DATA CENTRES

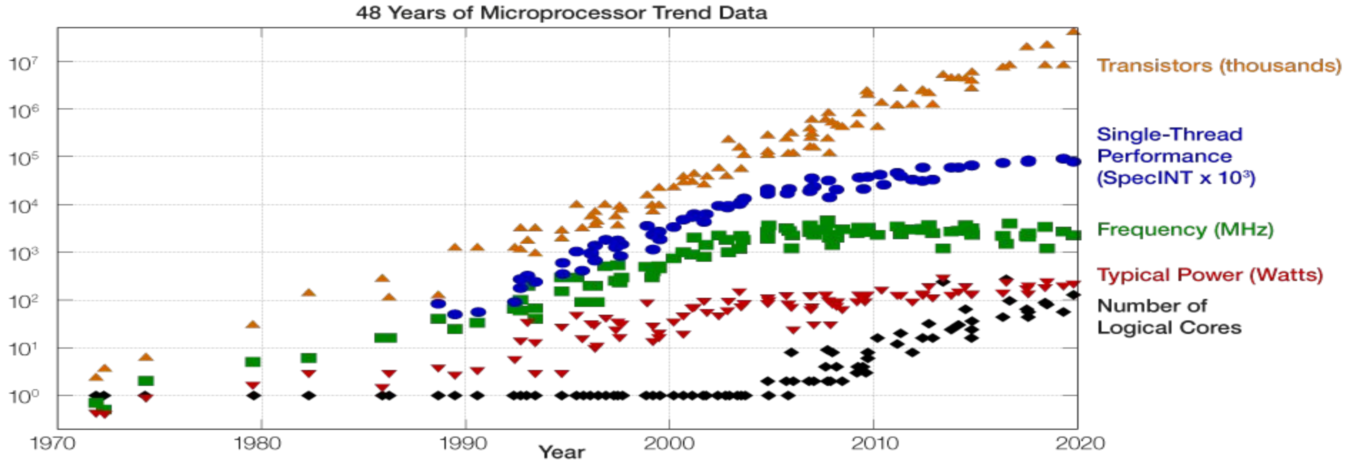
- CINECA
- JÜLICH Forschungszentrum
- cea
- BSC
- IT4I
- Jožef Stefan Institute

TECHNOLOGY & CO-DESIGN PARTNERS

- Atos
- SIPEARL The Silicon Pearl
- E4 Computer Engineering
- LEONARDO

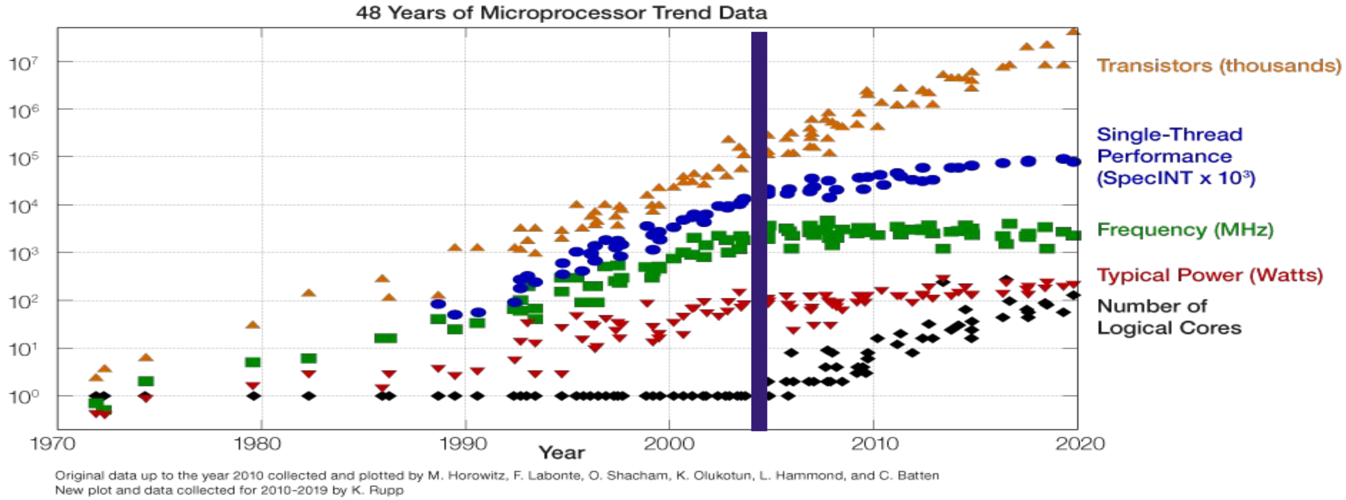
MAX coordination and management: Cnr – Modena, Italy

Why GPUs?

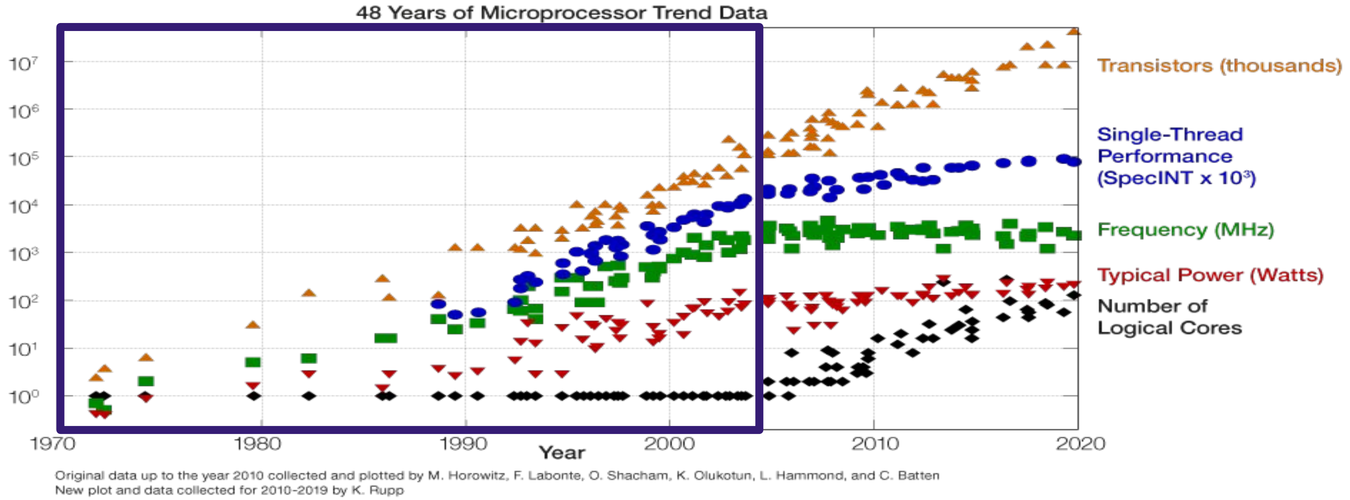


Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2019 by K. Rupp

Why GPUs?


48 Years of Microprocessor Trend Data
10^7
10^6
10^5
10^4
10^3
10^2
10^1
10^0
1970
1980
1990
2000
2010
2020
Year
Transistors (thousands)
Single-Thread Performance (SpecINT x 10^3)
Frequency (MHz)
Typical Power (Watts)
Number of Logical Cores
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2019 by K. Rupp

Dennard scaling ('70/~05)



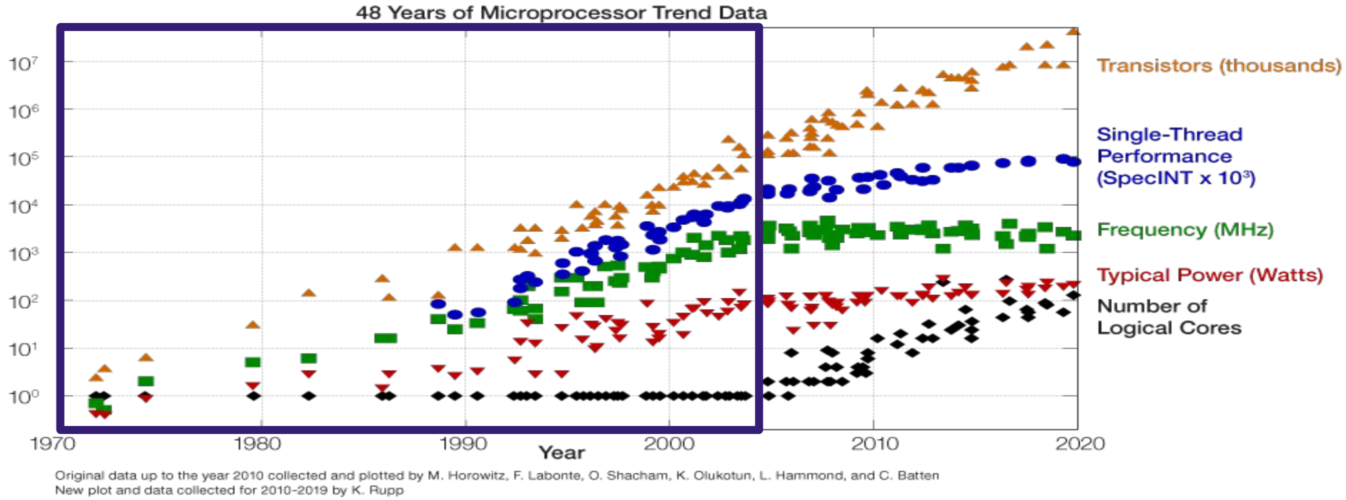
Dennard Scaling:
-) Shrink transistor and decrease voltage

-) Increase frequency

-) Power density stays constant!

Moore law: The number of transistor per chip doubles every 2 yrs

Dennard scaling (‘70/~05)



Dennard Scaling:
-) Shrink transistor and decrease voltage

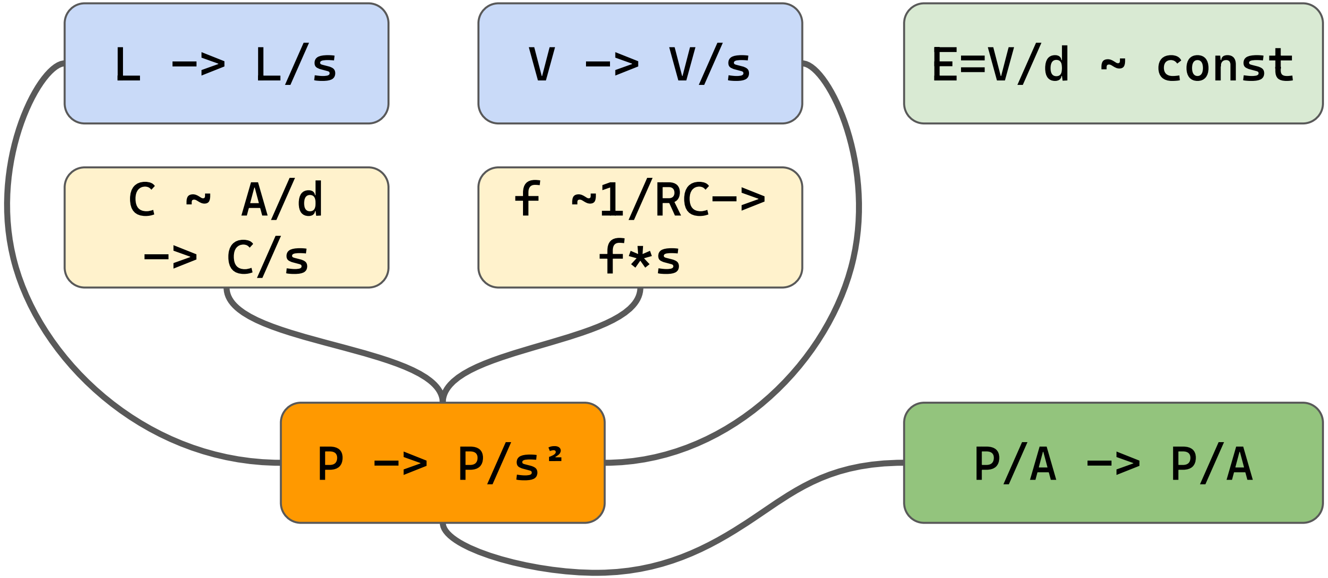
-) Increase frequency

-) Power density stays constant!

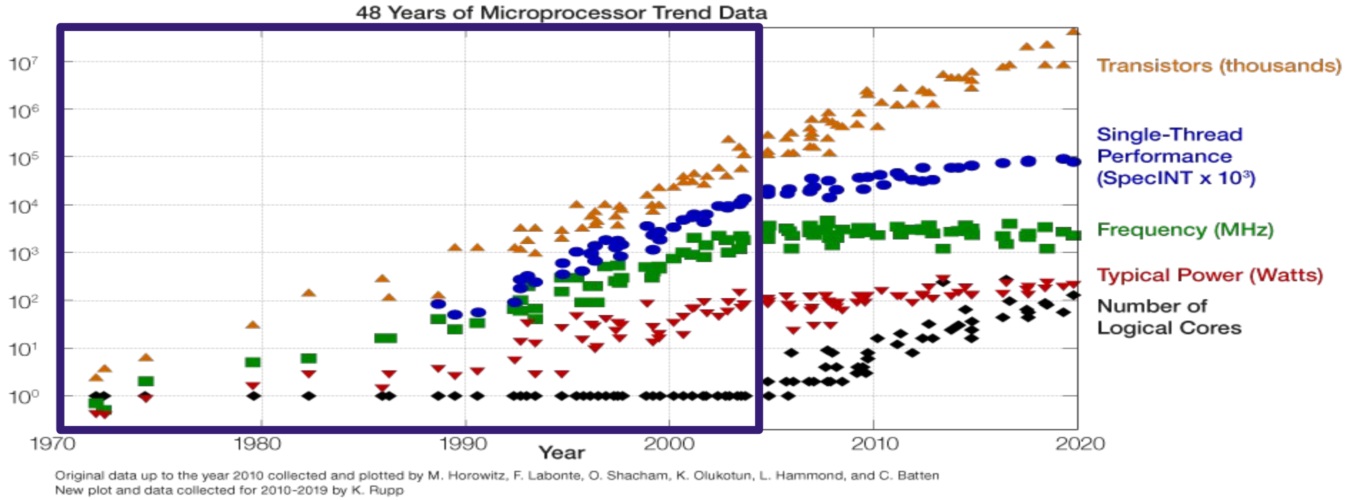
Moore law: The number of transistor per chip doubles every 2 yrs

Dennard scaling (‘70/~05): constant E

- Model the transistor as an RC circuit kept at voltage V:
- $P = a * (CV^2) * f$



Dennard scaling (‘70/~05)



Dennard Scaling:
-) Shrink transistor and decrease voltage

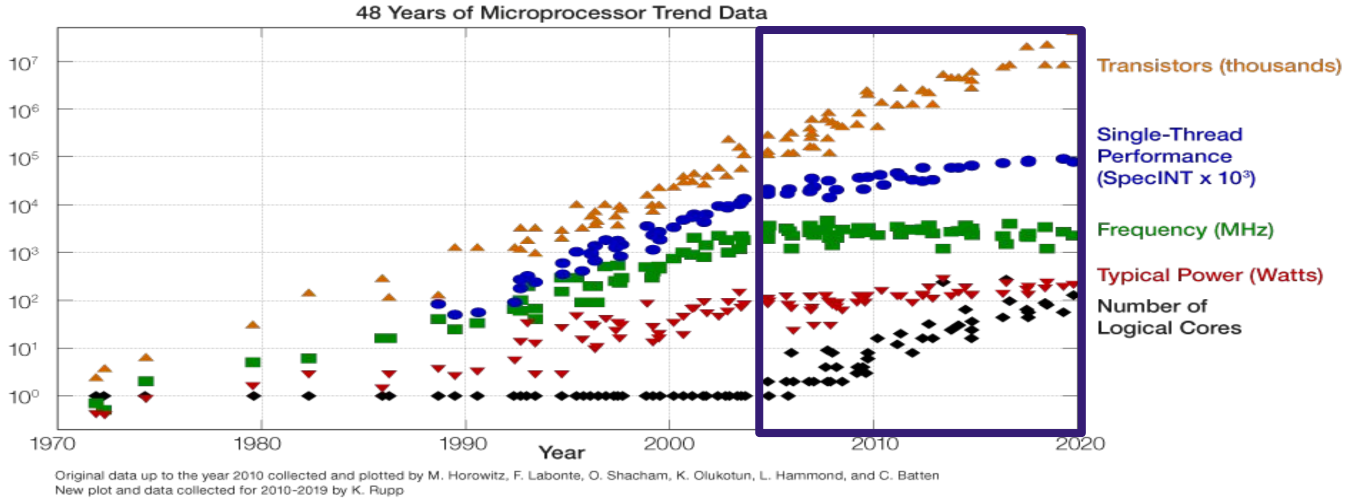
-) Increase frequency

-) Power density stays constant!

Moore law: The number of transistor per chip doubles every 2 yrs

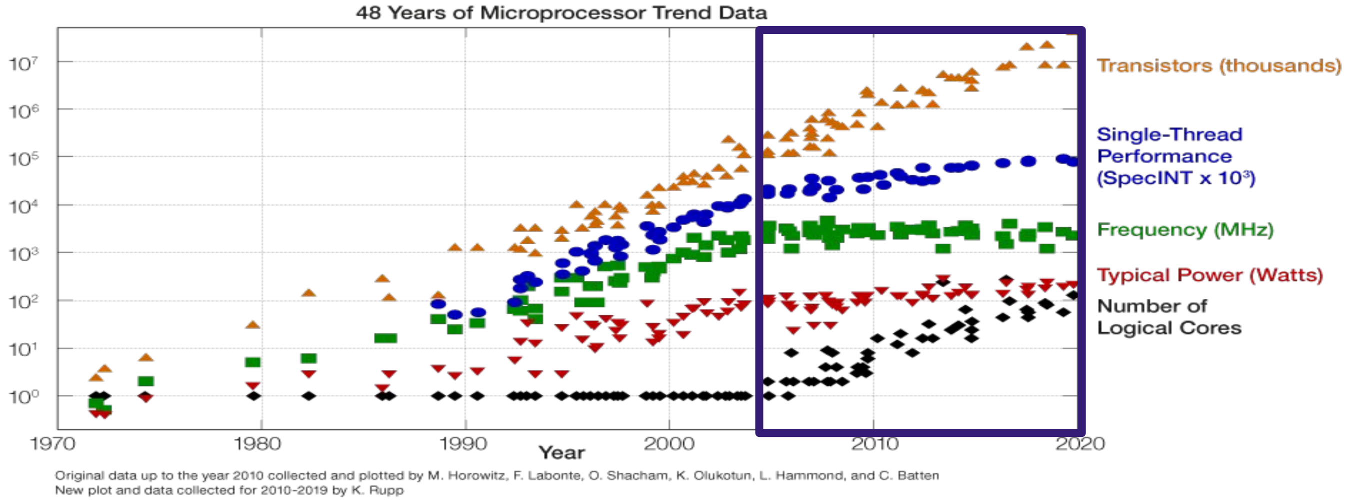
Free lunch!

The power wall



1. When the voltage becomes too small the transistor is not reliable anymore
2. Scaling at fixed voltage yields an increase in power density
3. We cannot increase frequency as before!

The power wall

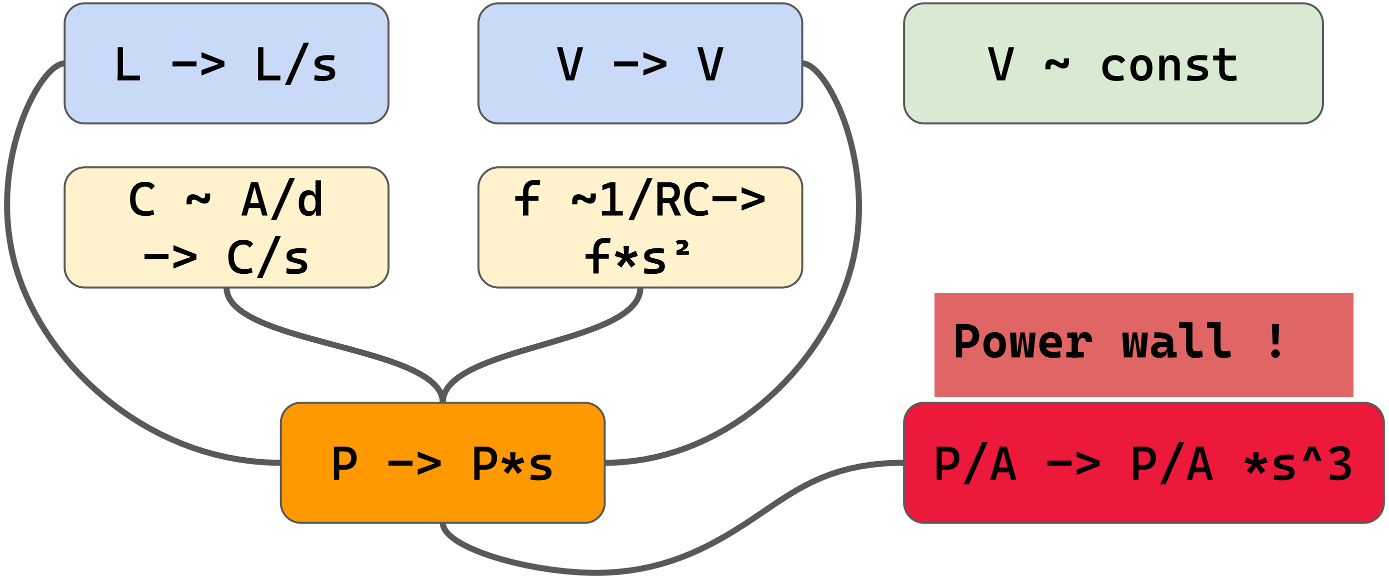


1. When the voltage becomes too small the transistor is not reliable anymore
2. Scaling at power density
3. We cannot i

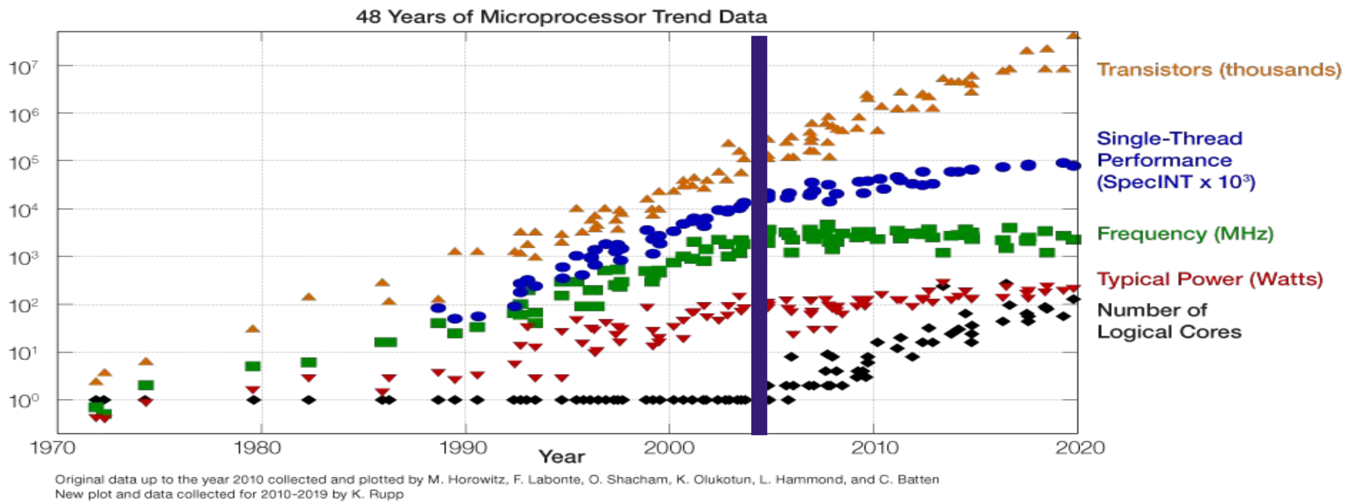
No more free lunch!

Dennard scaling (‘70/~05): constant V

- The transistor has capacitance C, resistance R, and is kept at voltage V :
- $P = a * (CV^2) * f$



So what?



More parallel:

Multicore CPUs

(but still at most 10^2)

More smart:

1. ILP, vectorization, Branch predictor
2. Specialized units: (AES-NI)

More throttled:

Not all transistor are active at the same time (Dark silicon)

Origins of GPU massive parallelism

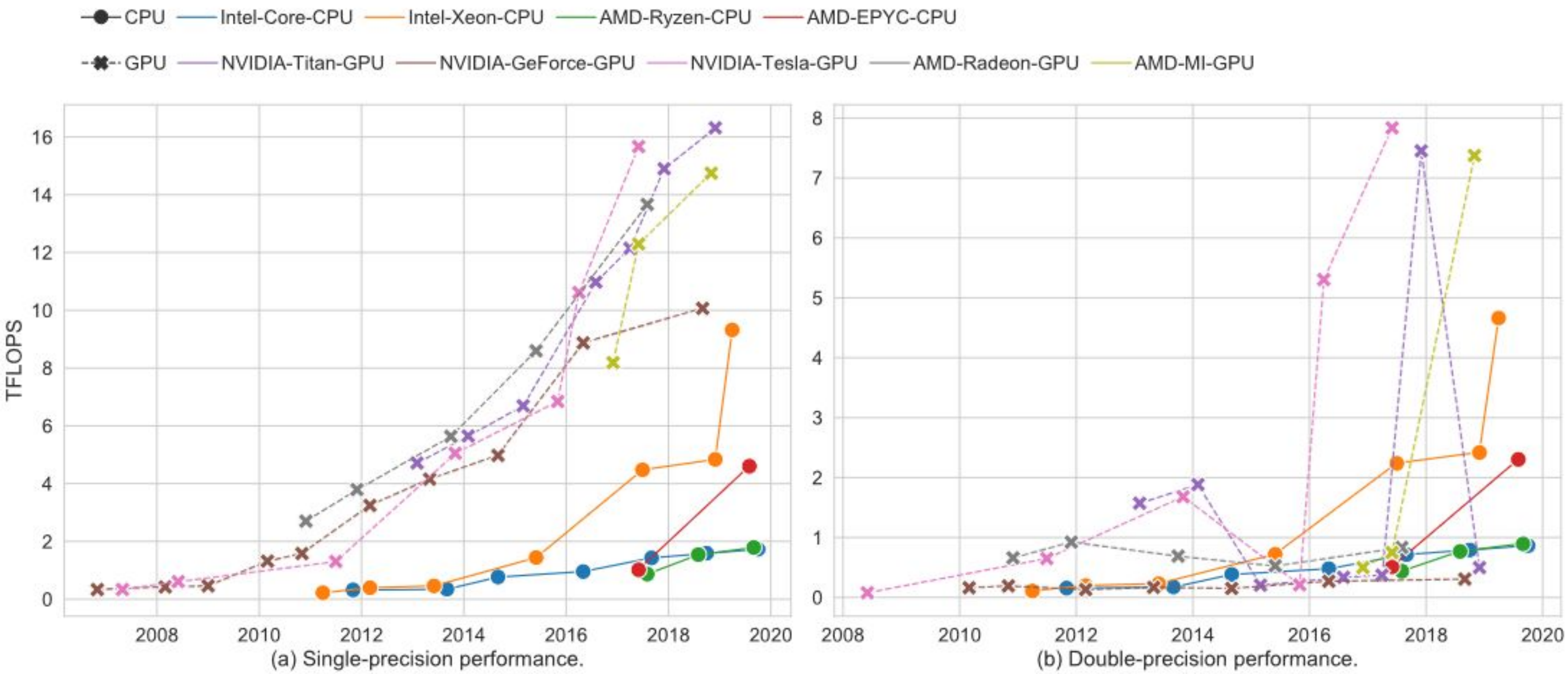
GPUs were introduced in the 90s for 3D rendering

1. On each point (~10^6) you apply ops (linear algebra) independently from other points

2. Each op is independent on the others and they are performed in parallel at the same time

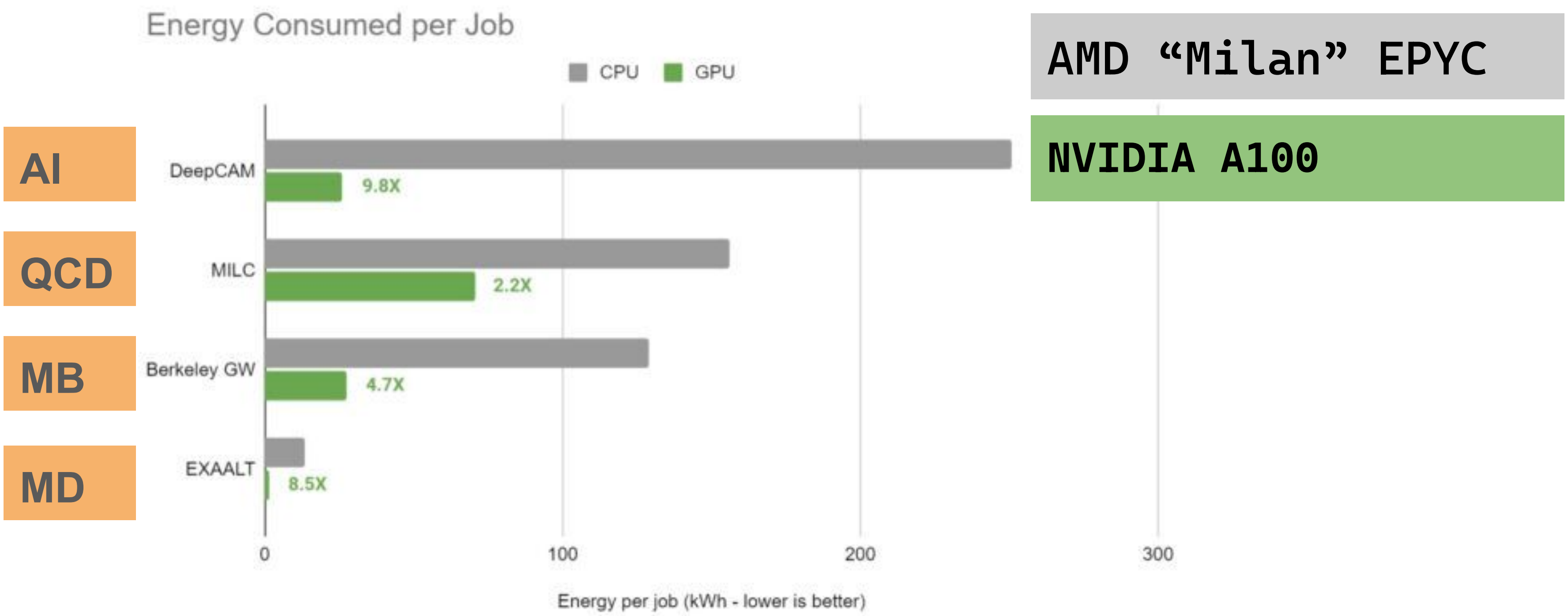
Large number of threads to process the data concurrently: order of processing is not important!

GPUs vs CPUs over the years



http://arxiv.org/pdf/1911.11313

Energy consumption (Perlmutter, 2023)



https://blogs.nvidia.com/blog/gpu-energy-efficiency-nersc/

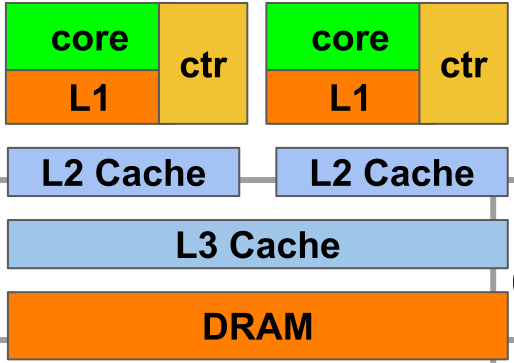
Top500: then (2009)....no GPUs

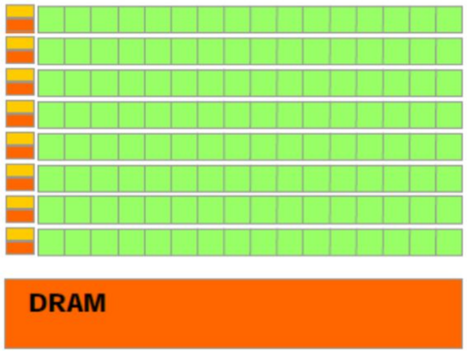
Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	**Jaguar** - Cray XT5-HE Opteron 6-core 2.6 GHz, Cray/HPE DOE/SC/Oak Ridge National Laboratory United States	224,162	1,759.00	2,331.00	6,950
2	**Roadrunner** - BladeCenter QS22/LS21 Cluster, PowerXCell 8i 3.2 Ghz / Opteron DC 1.8 GHz, Voltaire Infiniband, IBM DOE/NNSA/LANL United States	122,400	1,042.00	1,375.78	2,345
3	**Kraken XT5** - Cray XT5-HE Opteron 6-core 2.6 GHz, Cray/HPE National Institute for Computational Sciences/University of Tennessee United States	98,928	831.70	1,028.85	3,090
4	**JUGENE** - Blue Gene/P Solution, IBM Forschungszentrum Juelich (FZJ) Germany	294,912	825.50	1,002.70	2,268
5	**Tianhe-1** - NUDT TH-1 Cluster, Xeon E5540/E5450, ATI Radeon HD 4870 2, Infiniband, NUDT National SuperComputer Center in Tianjin/NUDT China	71,680	563.10	1,206.19	
6	**Pleiades** - SGI Altix ICE 8200EX, Xeon QC 3.0 GHz/Nehalem EP 2.93 Ghz, HPE NASA/Ames Research Center/NAS United States	56,320	544.30	673.26	
7	**BlueGene/L** - eServer Blue Gene Solution, IBM DOE/NNSA/LLNL United States	212,992	478.20	596.38	
8	**Intrepid** - Blue Gene/P Solution, IBM DOE/SC/Argonne National Laboratory United States	163,840	458.61	557.06	
9	**Ranger** - SunBlade x6420, Opteron QC 2.3 Ghz, Infiniband, Oracle Texas Advanced Computing Center/Univ. of Texas United States	62,976	433.20	579.38	
10	**Red Sky** - Sun Blade x6275, Xeon X55xx 2.93 Ghz, Infiniband, Oracle Sandia National Laboratories / National Renewable Energy Laboratory United States	41,616	423.90	487.74	

Top500: …and now (2025)

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**El Capitan** - HPE Cray EX255a, AMD 4th Gen EPYC 24C 1.8GHz, AMD Instinct MI300A, Slingshot-11, TOSS, HPE DOE/NNSA/LLNL United States	11,039,616	1,742.00	2,746.38	29,581
2	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE Cray OS, HPE DOE/SC/Oak Ridge National Laboratory United States	9,066,176	1,353.00	2,055.72	24,607
3	**Aurora** - HPE Cray EX - Intel Exascale Compute Blade, Xeon CPU Max 9470 52C 2.4GHz, Intel Data Center GPU Max, Slingshot-11, Intel DOE/SC/Argonne National Laboratory United States	9,264,128	1,012.00	1,980.01	38,698
4	**JUPITER Booster** - BullSequana XH3000, GH Superchip 72C 3GHz, NVIDIA GH200 Superchip, Quad-Rail NVIDIA InfiniBand NDR200, RedHat Enterprise Linux, EVIDEN EuroHPC/FZJ Germany	4,801,344	793.40	930.00	13,088
6	**HPC6** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, RHEL 8.9, HPE Eni S.p.A. Italy	3,143,520	477.90	606.97	8,461
7	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	537.21	29,899
8	**Alps** - HPE Cray EX254n, NVIDIA Grace 72C 3.1GHz, NVIDIA GH200 Superchip, Slingshot-11, HPE Cray OS, HPE Swiss National Supercomputing Centre (CSCS) Switzerland	2,121,600	434.90	574.84	7,124
9	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	2,752,704	379.70	531.51	7,107
10	**Leonardo** - BullSequana XH2000, Xeon Platinum 8358 32C 2.6GHz, NVIDIA A100 SXM4 64 GB, Quad-rail NVIDIA HDR100 Infiniband, EVIDEN EuroHPC/CINECA Italy	1,824,768	241.20	306.31	7,494

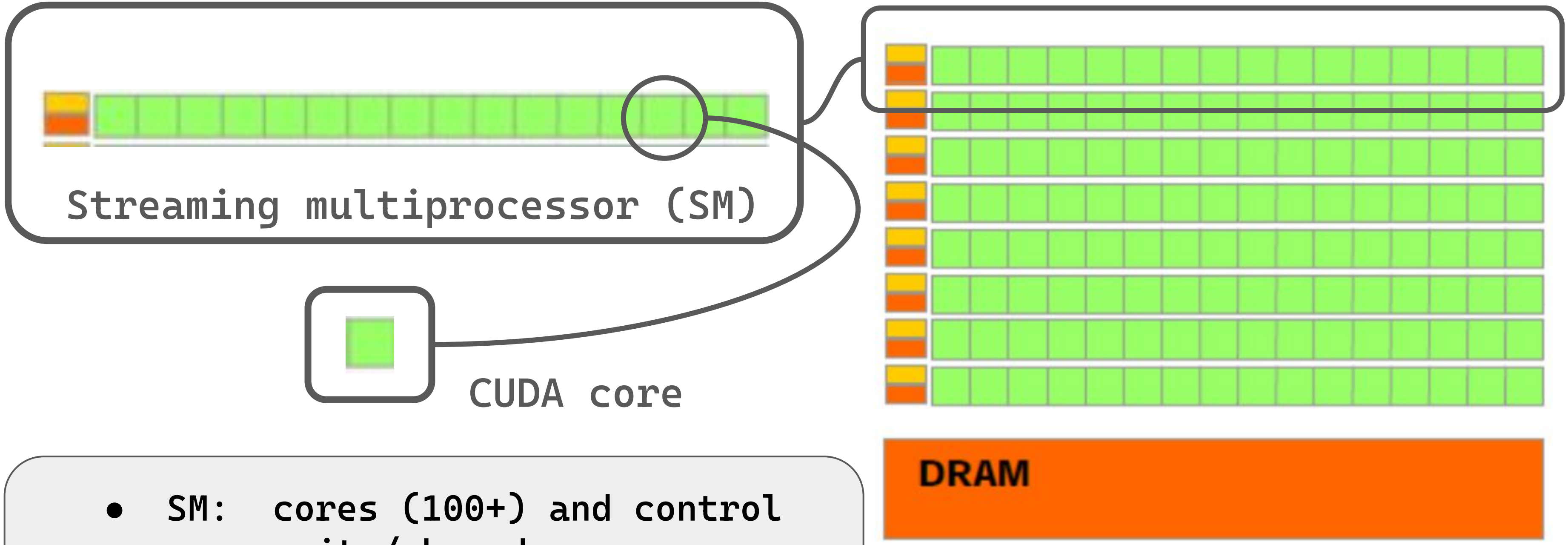
CPU vs GPU





CPU	GPU
Low latency	High throughput
Few complex cores (8-64)	Many “simpler” cores (5000+)
Task parallelism/irregular workflows	Data parallelism

GPU architecture



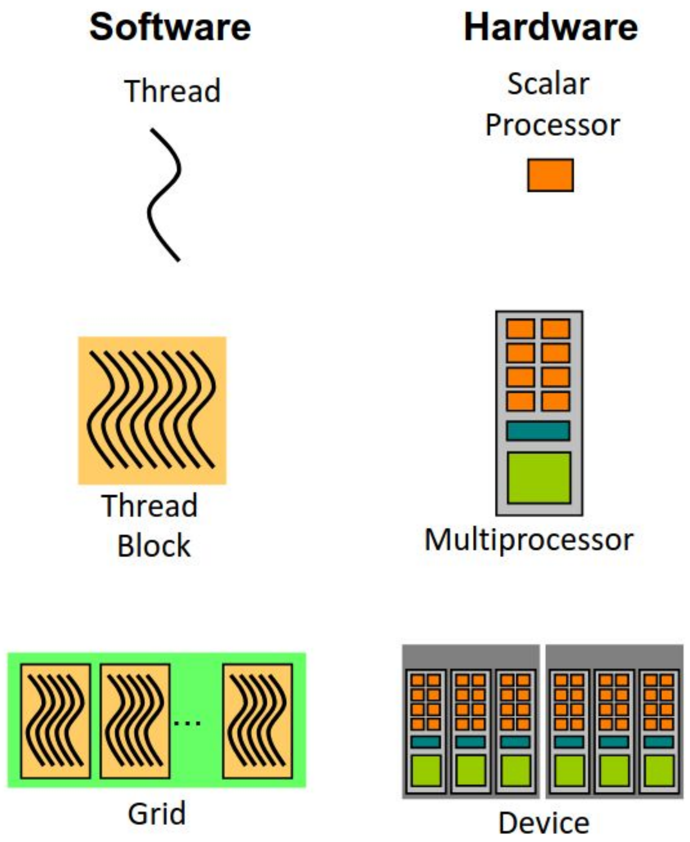
- SM: cores (100+) and control units/shared memory
  (Up to 100 SM per device)

SMs over the years

GP(ascal)100 (2016)

A(mpere)100 (2020)

Levels of parallelism on the GPU



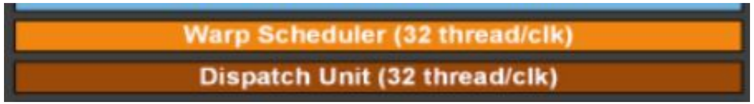
Threads are executed by scalar processor

Thread blocks are executed by SM

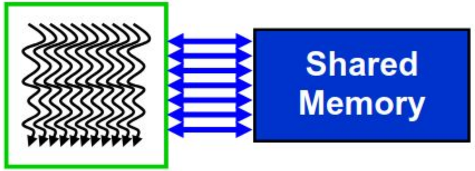
Thread block **do not migrate**

Several blocks share the resources of the SM

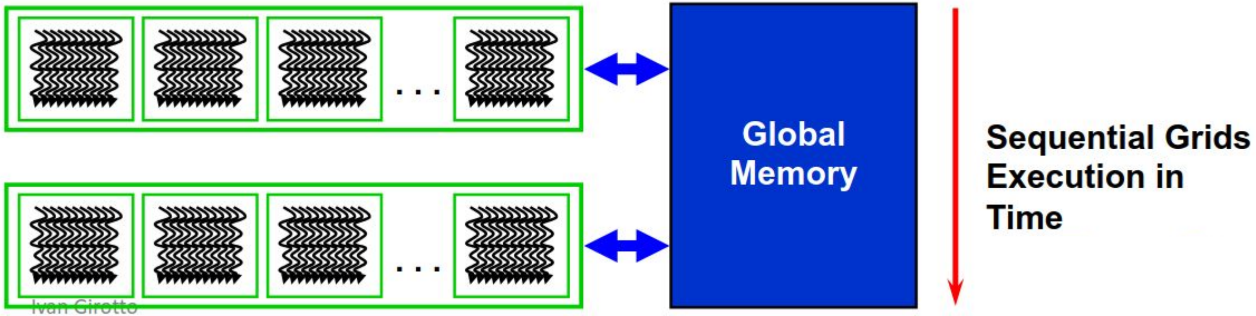
A Kernel is launched spawning a **grid** of thread blocks



Memory structure



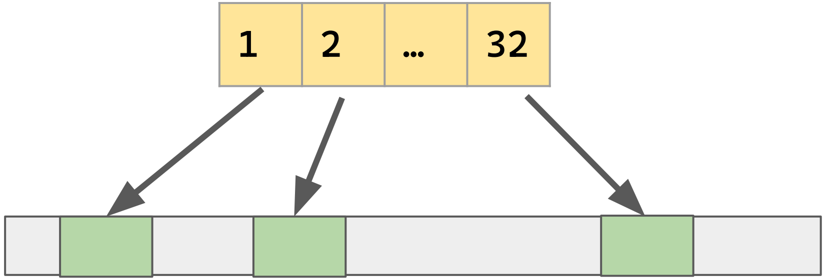
- Registers (fast up to availability)
  - Limited num. registers available per block
- Shared Memory: per-block
  - Shared by threads of the same block
  - *Fast* inter-thread communication
- Global Memory: per-application
  - Shared by all threads
  - Inter-Grid communication



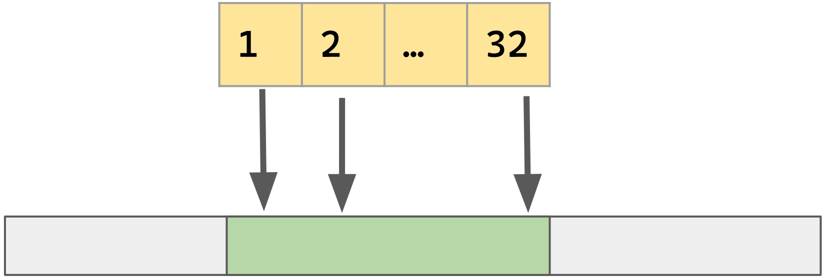
Coalesced access

Global memory access can only happen in transactions of 32/128B
(The hardware will try to request as few transactions as possible)

Example: assume the warp needs 32 integers (4 bytes each)



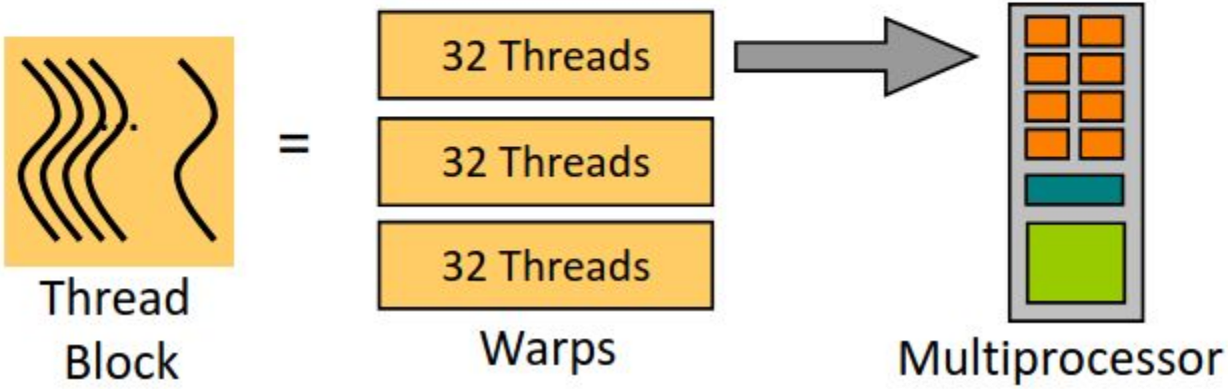
The data are scattered on the global memory:
32*128 bytes loaded but 128 needed (at worst)



The data are contiguous on the global memory:
128 bytes loaded and 128 needed

Another possibility is loading the data on the shared memory !

Warps



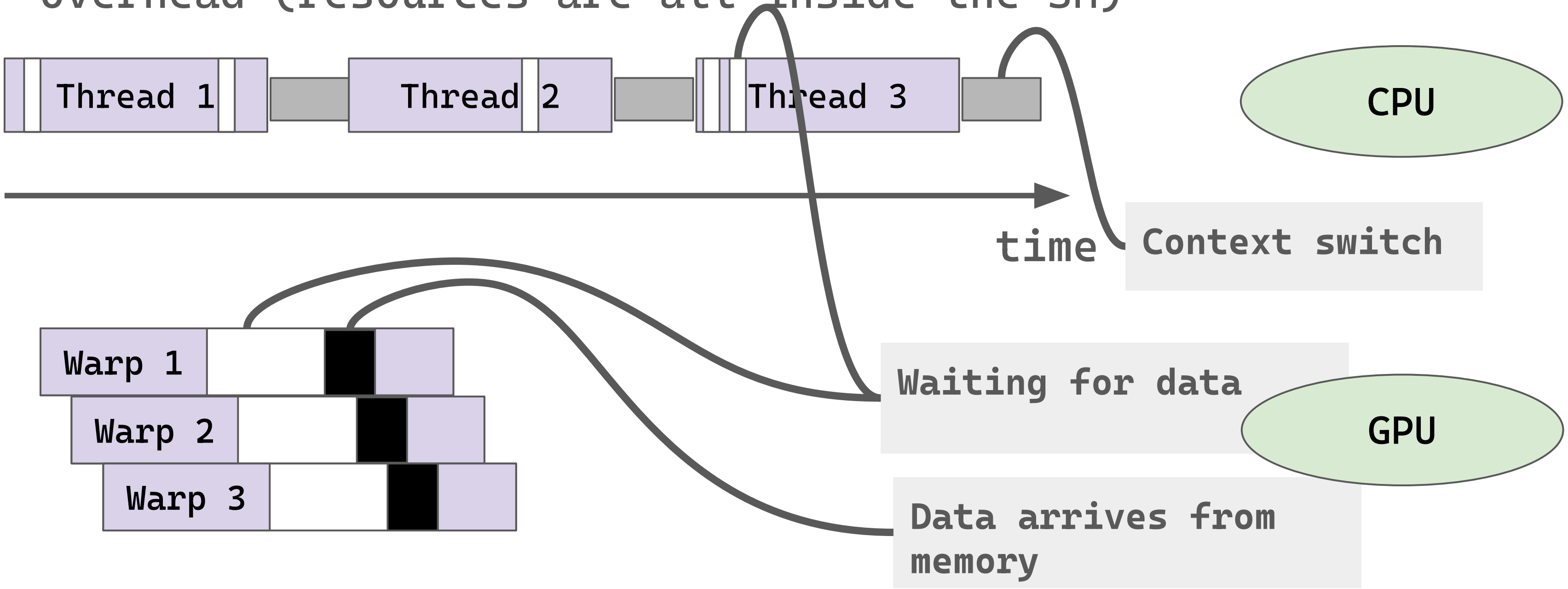
- Blocks are processed in units of 32 threads, called **Warps**
- SM executes warps coming from different blocks
- Warps are executed in a SIMD-like fashion:
  -) All threads execute the same instruction

  -) If one thread stalls, all 32 stall (and another warp is scheduled)

Warps: context switch

On the CPU for the OS is very expensive to swap threads (saving state of the current thread+restoring another one)

On the GPU the scheduler can switch warps with very little overhead (resources are all inside the SM)



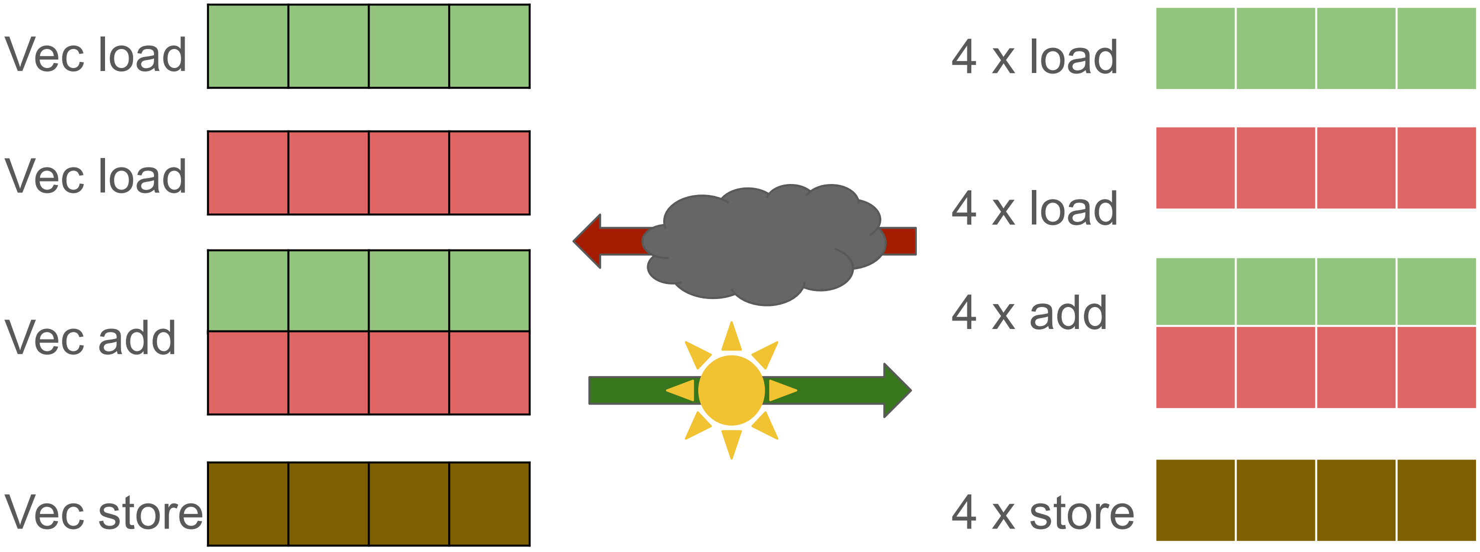
Warps: SIMD vs SIMT

Single Instruction Multiple Data:

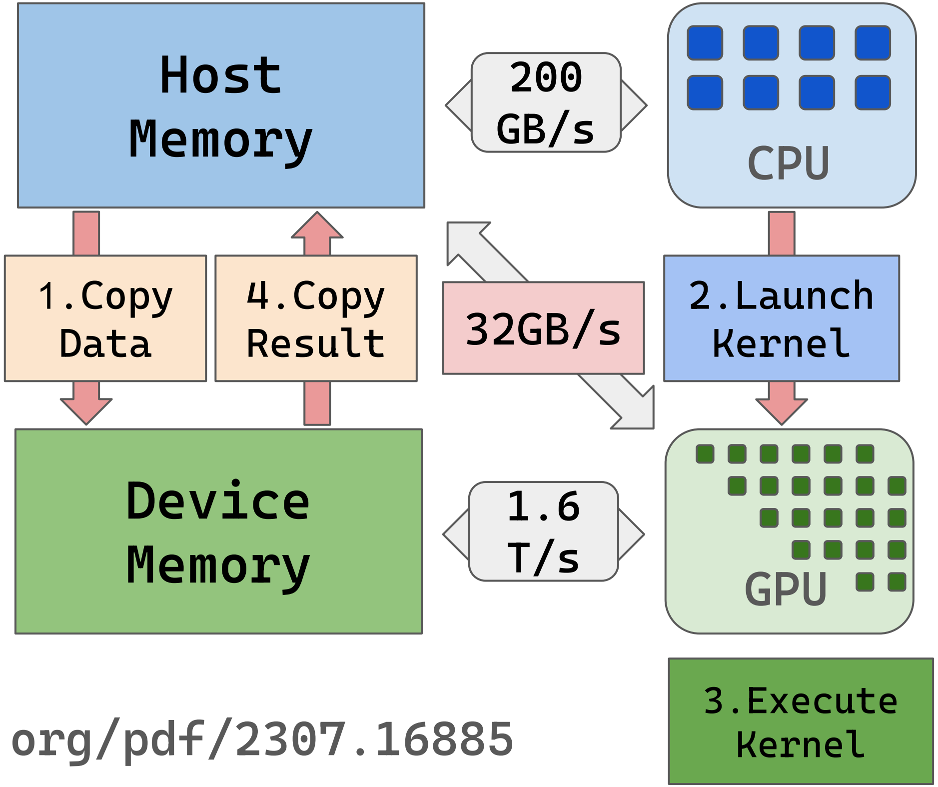
- Vector instruction:same instruction on contiguous data

Single Instruction Multiple Threads:

- Hardware enables parallel scalar instructions on not necessarily contiguous data

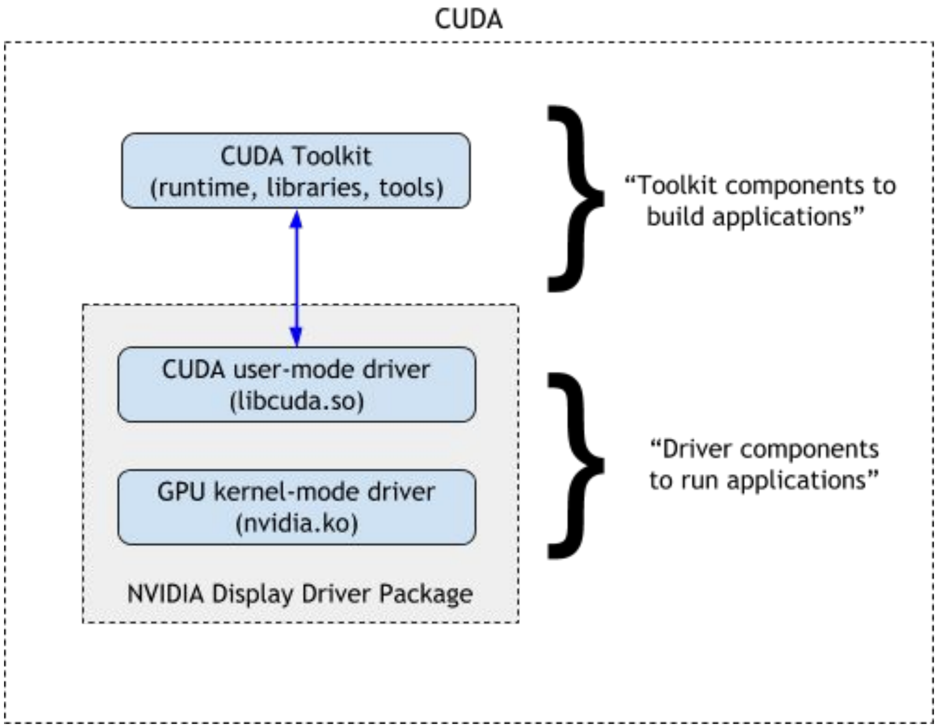


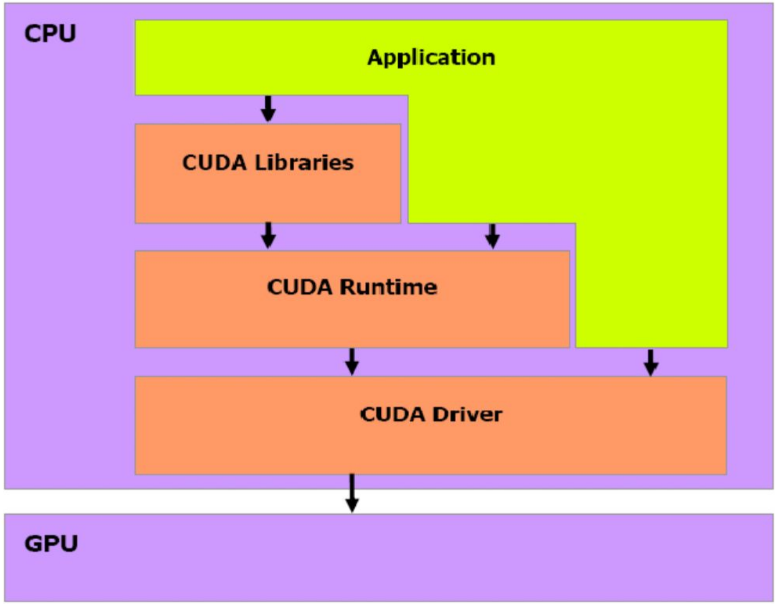
GPU-CPU interconnection (Leonardo numbers)



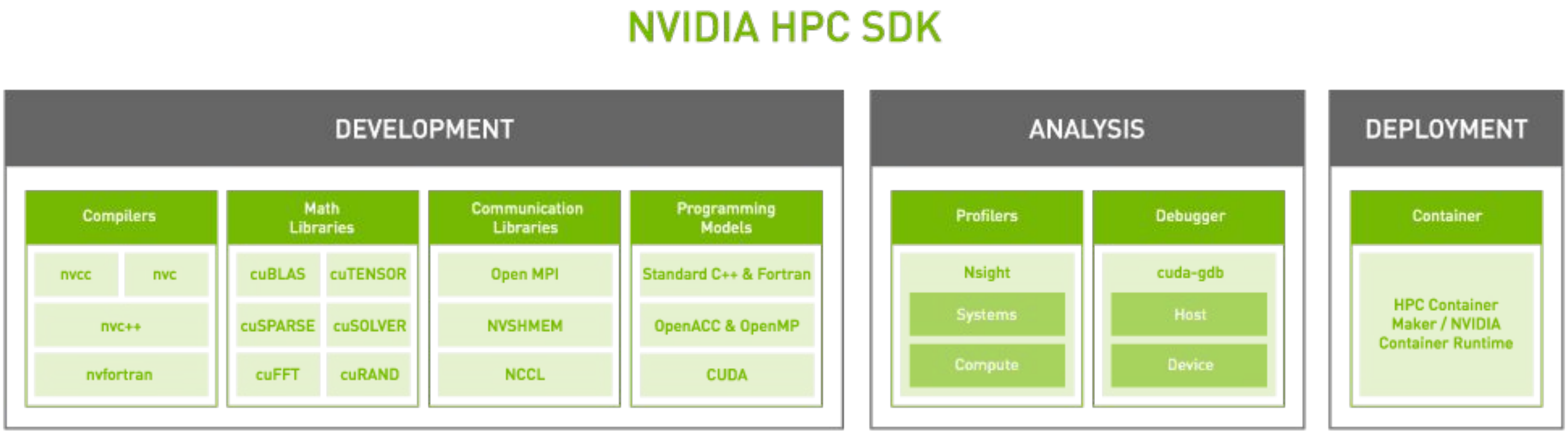
https://arxiv.org/pdf/2307.16885

What do we need to use a GPU?





What do we get?



https://developer.nvidia.com/hpc-sdk

There is also a python CUDA implementation: https://cupy.dev/

How write a code that exploits a GPU

Your application

Libraries (cuFFT…)	Directives (openACC/MP)	Programming languages (CUDA,HIP)
Replace functions Guaranteed perf Fixed interface	Custom code Portable Compiler helps	High flexibility Max perf “Low” level

Effort →

Basics of CUDA (Compute Unified Device Architecture)

- Computation partitioning:
  - On definition: __host__ __global__ __device__

	__host__	__device__	__global__
Called from	CPU	GPU	CPU
Executed on	CPU	GPU	GPU

- Data management, copy from/to device/host:
  - cudaMemcpy(h_data, d_data, size, CudaMemcpyDeviceToHost)

- …and much more (asynchronous programming……)

Minimal (trivial) CUDA code example

```
__global__ void sum (int a, int b, int *sum) { *sum=a+b}

int main(){
    int *dev_sum, h_sum;

    cudaMalloc(&dev_sum, sizeof(int));

    sum<<<1,1>>>(1,2,dev_sum);

    cudaMemcpy(&h_sum, dev_sum, cudaMemcpyDeviceToHost);

    printf("%d\n", h_sum);

    cudaFree(dev_sum);
```

- Save the file with **.cu** extension
- Compile: nvcc test.cu		Run: ./a.out

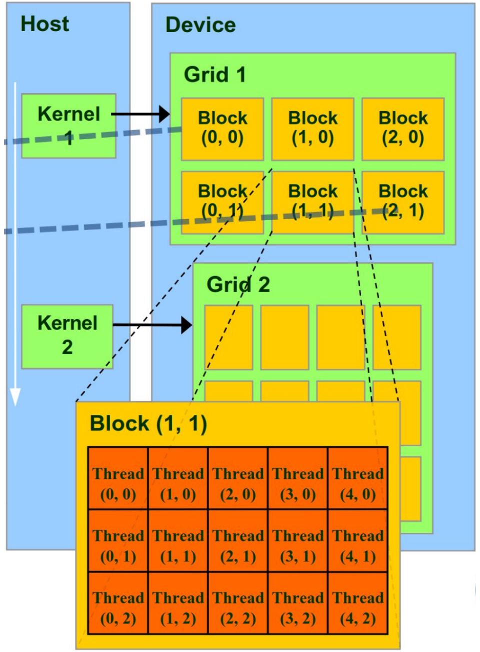
Example of Cuda C code

```
int main(){
  int dim; int s=sizeof(int)*dim
  int * h_a = (int *) malloc(h_a,s);
  int * h_b = (int *) malloc(h_b,s);
  int * h_c = (int *) malloc(h_c,s);
  int *d_a,*d_b,*d_c;
  for(int i=0; i<dim;i++){
      h_a[i]=1; h_b[i]=2;
  }

  cudaMalloc((void **)&d_a,s);
  cudaMalloc((void **)&d_b,s);
  cudaMalloc((void **)&d_c,s);

  cudaMemCpy(d_a,h_a,s,CudaMemcpyHostToDevice)
  cudaMemCpy(d_b,h_b,s,CudaMemcpyHostToDevice)

  add<<1,1>>(d_a,d_b,d_c,dim);
  cudaMemCpy(h_c,d_c,s,CudaMemcpyDeviceToHost)

  free(h_a); free(h_b); free(h_c);
  cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
```

```
int main(){
  int dim; int s=sizeof(int)*dim
  int * h_a = (int *) malloc(h_a,s);
  int * h_b = (int *) malloc(h_b,s);
  int * h_c = (int *) malloc(h_c,s);

  for(int i=0; i<dim;i++){
      h_a[i]=1; h_b[i]=2;
  }

  for(int i=0; i<dim;i++){
      h_c[i]=h_a[i]+h_b[i];
  }


  free(h_a); free(h_b); free(h_c);
```

Kernel calling syntax

```
myKernel <<<grid_size, block_size>>> (args)
```



grid_size: number of blocks along x,y,z

block_size:number of threads along x,y,z

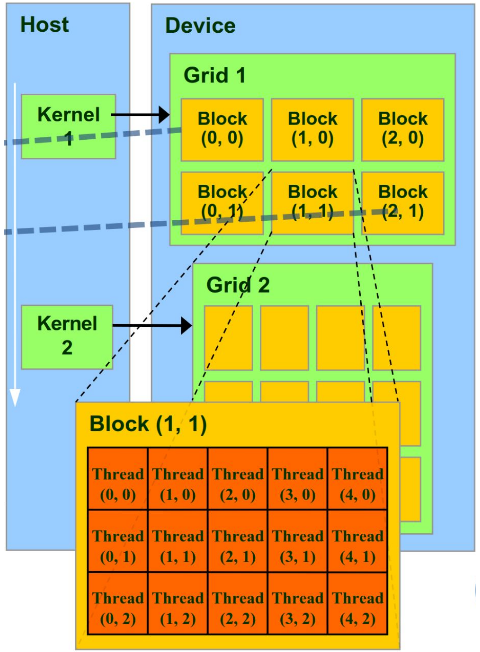
e.g: grid_size-->(3,2) block_size-->(5,3)

Typically one uses the CUDA structure Dim3 to set the grid and block size

If grid and block are integers, then the runtime generates a 1d grid composed of 1d blocks

Kernel calling syntax

```
myKernel <<<grid_size, block_size>>> (args)
```



grid_size: number of blocks along x,y,z

block_size:number of threads along x,y,z

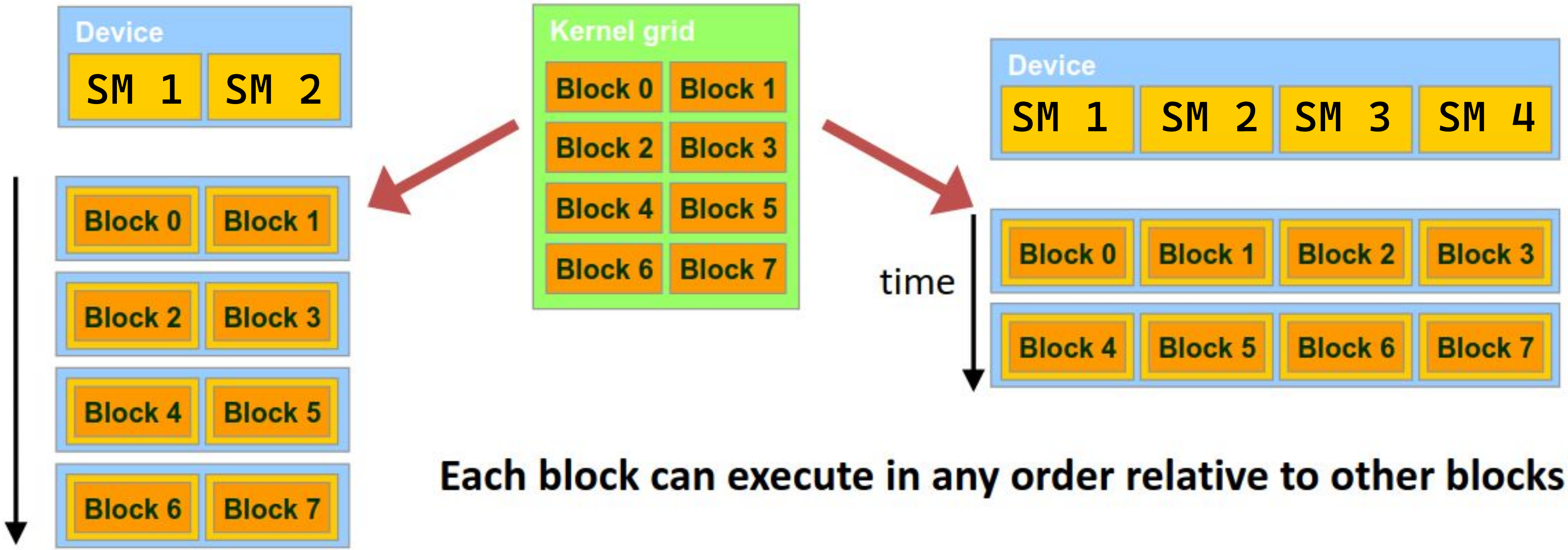
e.g: grid_size=(3,2) block_size=(5,3)

Each thread and block is identified by three indices:

(threadIdx.x, threadIdx.y, threadIdx.z)

(blockIdx.x, blockIdx.y, blockIdx.z)

Block execution



Each block can execute in any order relative to other blocks!

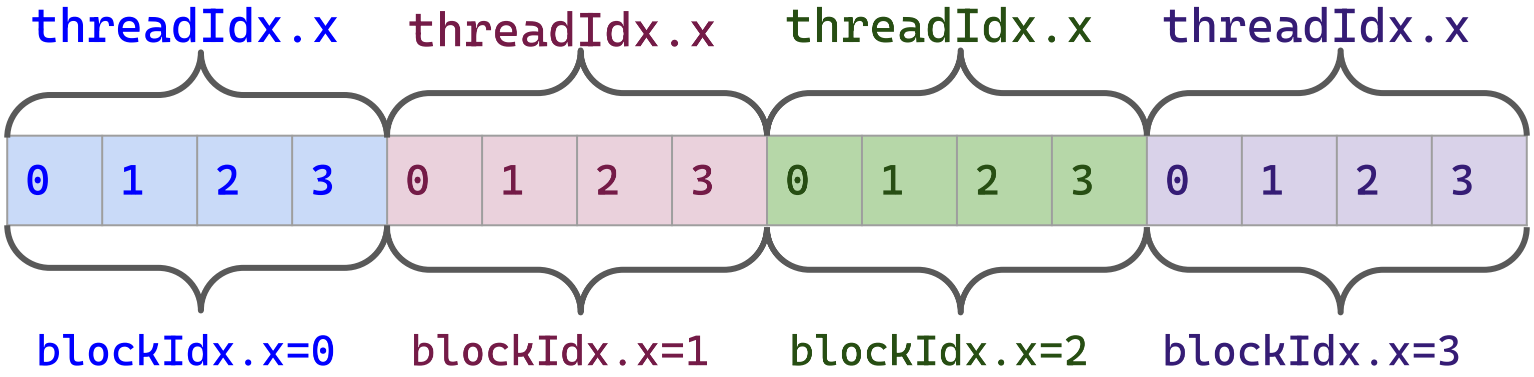
Example of Cuda C code

```
int main(){
  int dim; int s=sizeof(int)*dim
  int * h_a = (int *) malloc(h_a,s);
  int * h_b = (int *) malloc(h_b,s);
  int * h_c = (int *) malloc(h_c,s);
  int *d_a,*d_b,*d_c;
  for(int i=0; i<dim;i++){
      h_a[i]=1; h_b[i]=2;
  }

  cudaMalloc((void **)&d_a,s);
  cudaMalloc((void **)&d_b,s);
  cudaMalloc((void **)&d_c,s);

  cudaMemCpy(d_a,h_a,s,CudaMemcpyHostToDevice)
  cudaMemCpy(d_b,h_b,s,CudaMemcpyHostToDevice)

  add<<1,1>>(d_a,d_b,d_c,dim);
  cudaMemCpy(h_c,d_c,s,CudaMemcpyDeviceToHost)

  free(h_a); free(h_b); free(h_c);
  cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
```

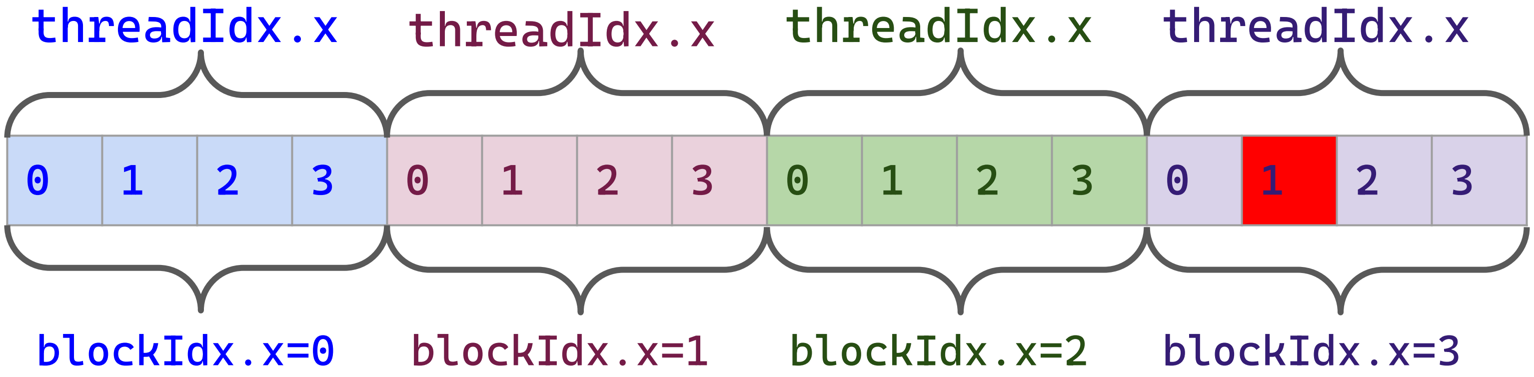
```
int main(){
  int dim; int s=sizeof(int)*dim
  int * h_a = (int *) malloc(h_a,s);
  int * h_b = (int *) malloc(h_b,s);
  int * h_c = (int *) malloc(h_c,s);

  for(int i=0; i<dim;i++){
      h_a[i]=1; h_b[i]=2;
  }

  for(int i=0; i<dim;i++){
      h_c[i]=h_a[i]+h_b[i];
  }


  free(h_a); free(h_b); free(h_c);
```

1 block, 1 thd per block

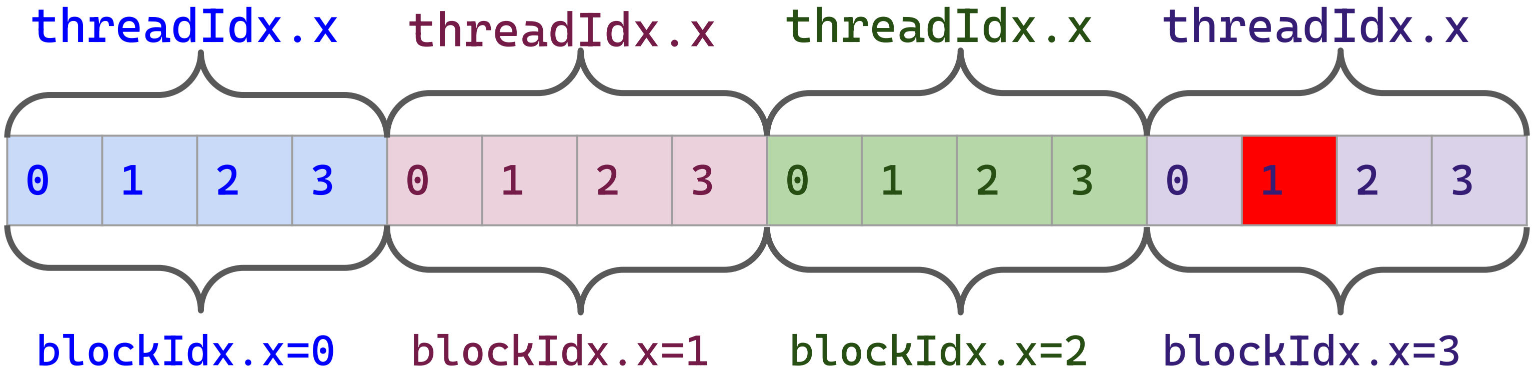
Thread Indexing (global vs local, again)



Thread Indexing (global vs local, again)



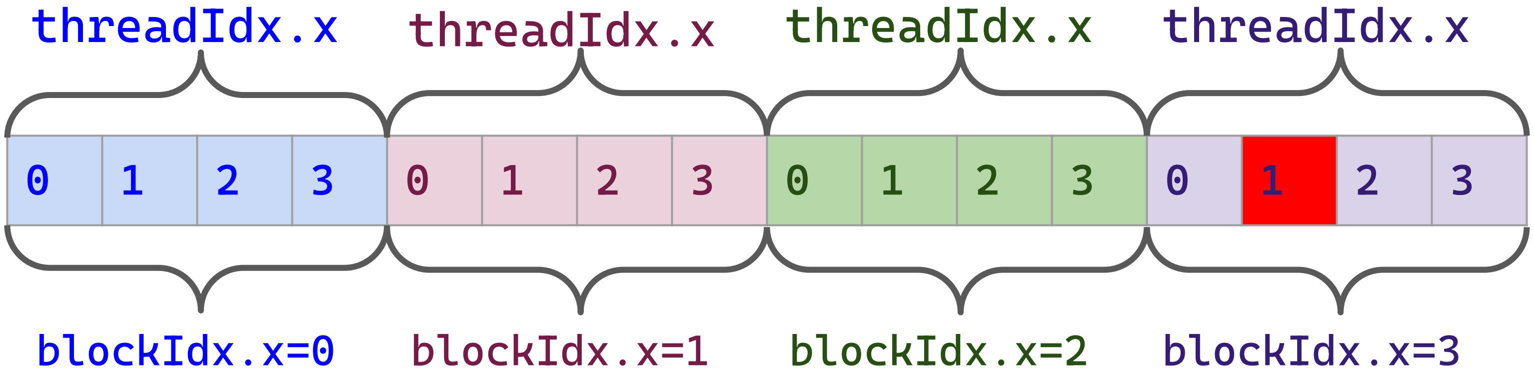
Thread Indexing (global vs local, again)



Given M threads per block, a unique index is:

```
int idx = blockIdx.x * M + threadIdx.x
```

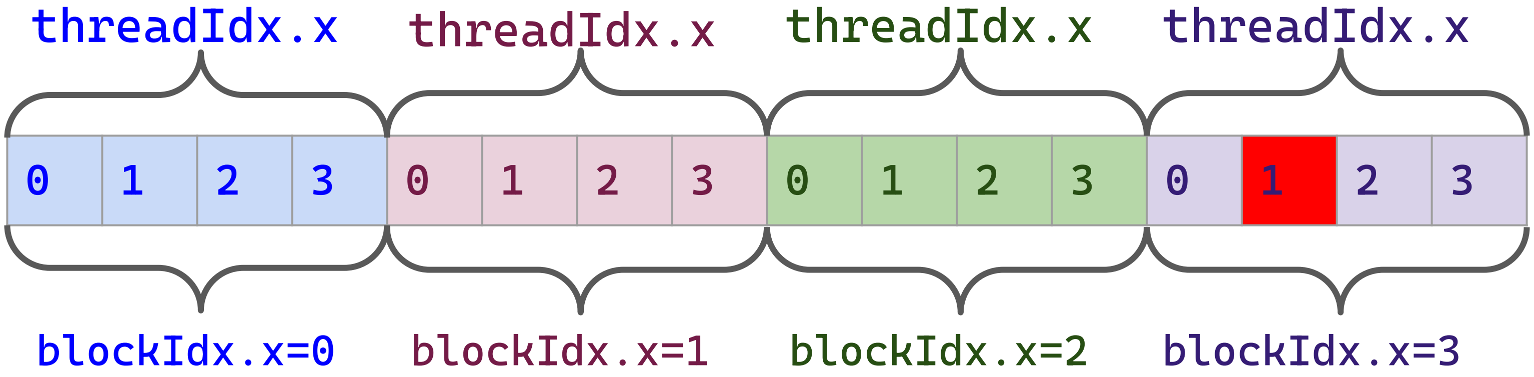
Thread Indexing (global vs local, again)



Given M threads per block, a unique index is:

$$\texttt{int idx = }\underbrace{\texttt{blockIdx.x}}_{3} * \underbrace{\texttt{M}}_{4} + \underbrace{\texttt{threadIdx.x}}_{1}$$

Thread Indexing (global vs local, again)



Given M threads per block, a unique index is:

```
int idx = blockIdx.x * M + threadIdx.x
```

Multidimensional thread indexing follows the same spirit

Thread Indexing: add kernel example

```
__global__ void add(int* A, int* B, int* C, int N)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}
```

Blocks/threads parallelism

```
#define N 2048*2048
#define T 1024 //threads per block
int main(){
  int dim; int s=sizeof(int)*dim
  int * h_a = (int *) malloc(h_a,s);
  int * h_b = (int *) malloc(h_b,s);
  int * h_c = (int *) malloc(h_c,s);
  int *d_a,*d_b,*d_c;
  for(int i=0; i<dim;i++){
      h_a[i]=1; h_b[i]=2;
  }
  cudaMalloc((void **)&d_a,s);
  cudaMalloc((void **)&d_b,s);
  cudaMalloc((void **)&d_c,s);

  cudaMemCpy(d_a,h_a,s,CudaMemcpyHostToDevice)
  cudaMemCpy(d_b,h_b,s,CudaMemcpyHostToDevice)

  add<<(int)ceil(N/T),T>>(d_a,d_b,d_c,dim);

  cudaMemCpy(h_c,d_c,s,CudaMemcpyDeviceToHost)
  free(h_a); free(h_b); free(h_c);
  cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
```

- The max number of threads/block, blocks/grid…are codified in the compute capabilities (cc)
- Leonardo: A100 cc=80

Blocks/threads parallelism

- Finding the number of blocks/threads to maximize the GPU occupancy is not easy

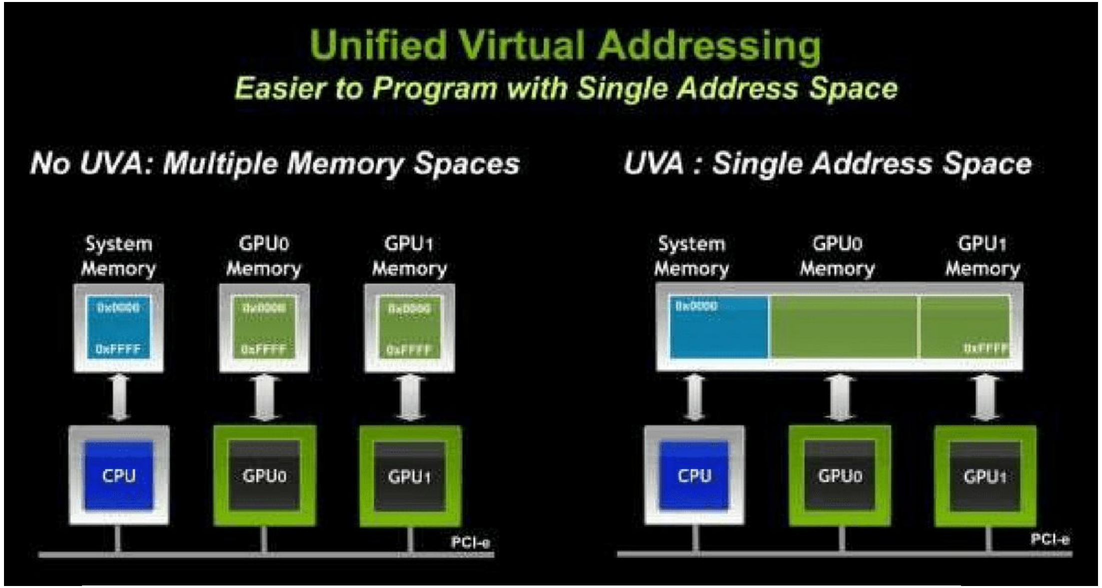
- Threads are executed in warps, following the .x direction

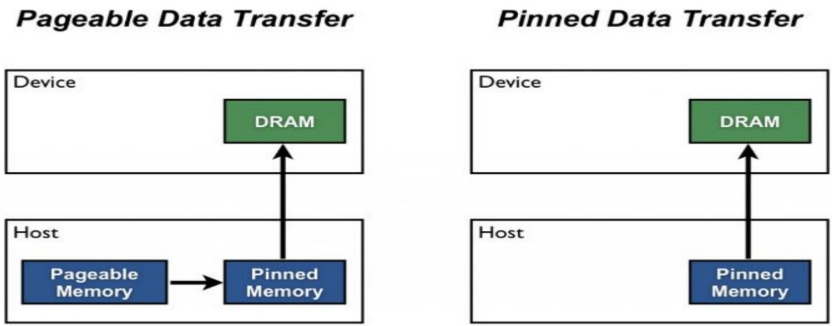
- If the number of threads per block is **not** multiple of 32, **a partially empty warp will be scheduled**, hurting performance

```
Device 0 has compute capability 8.0.
Device 0 has  maxThreadsPerBlock 1024
Device 0 has  warpSize 32
Device 0 has  maxThreadsPerMultiProcessor  2048
Device 0 has   maxThreadsDim[3] (1024,1024,64)
Device 0 has   maxGridSize[3]   (2147483647,65535,65535)
```

[CudaGetDeviceProperties]()

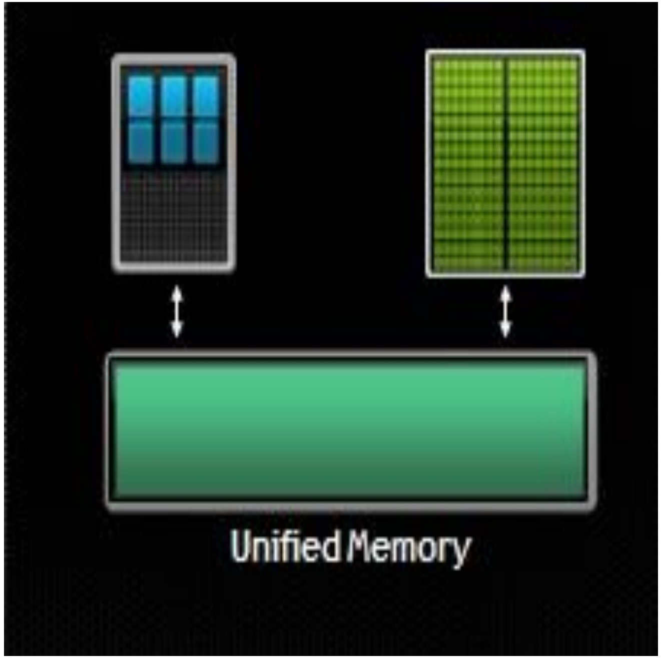
Unified Virtual Address





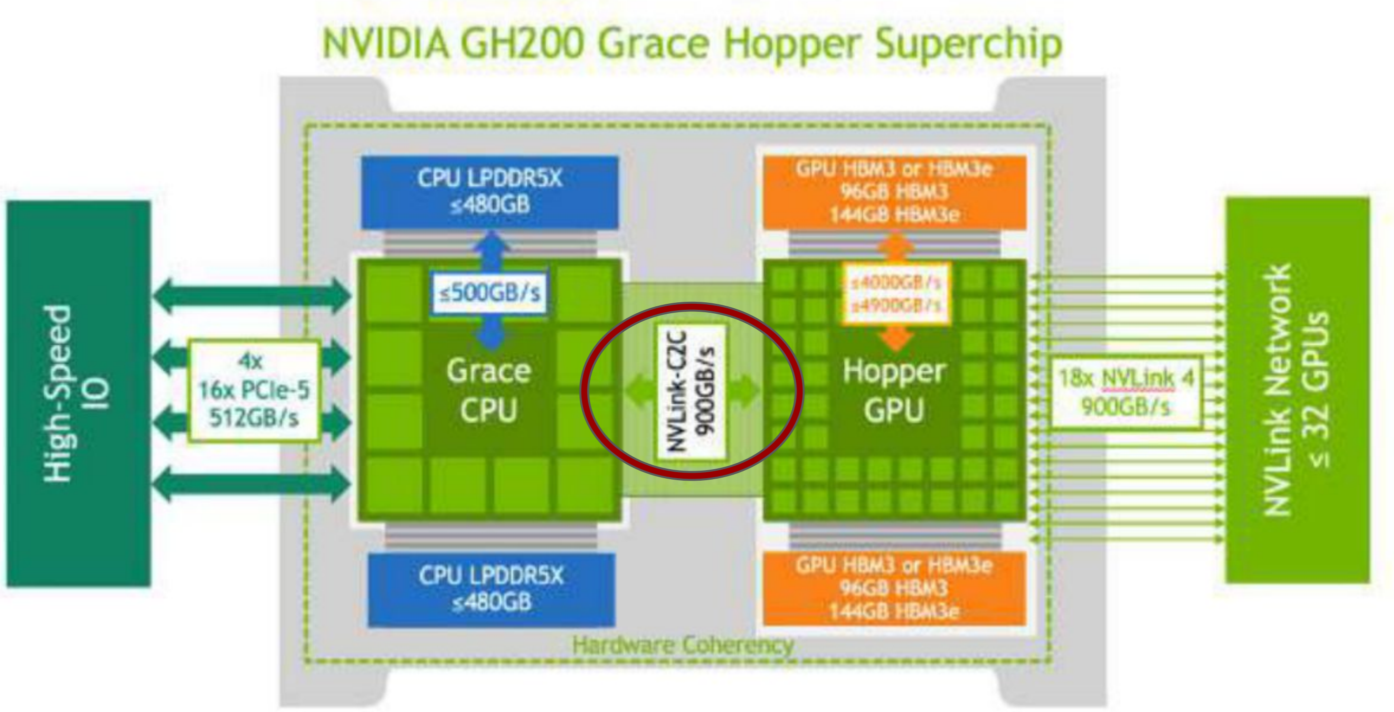
- Unified shared virtual address space for host/device
- Enables zero-copy memory access (but requires **pinned** memory)
- When you copy data between host/device you don’t need to specify the direction

Unified Memory (pre Grace-Hopper)

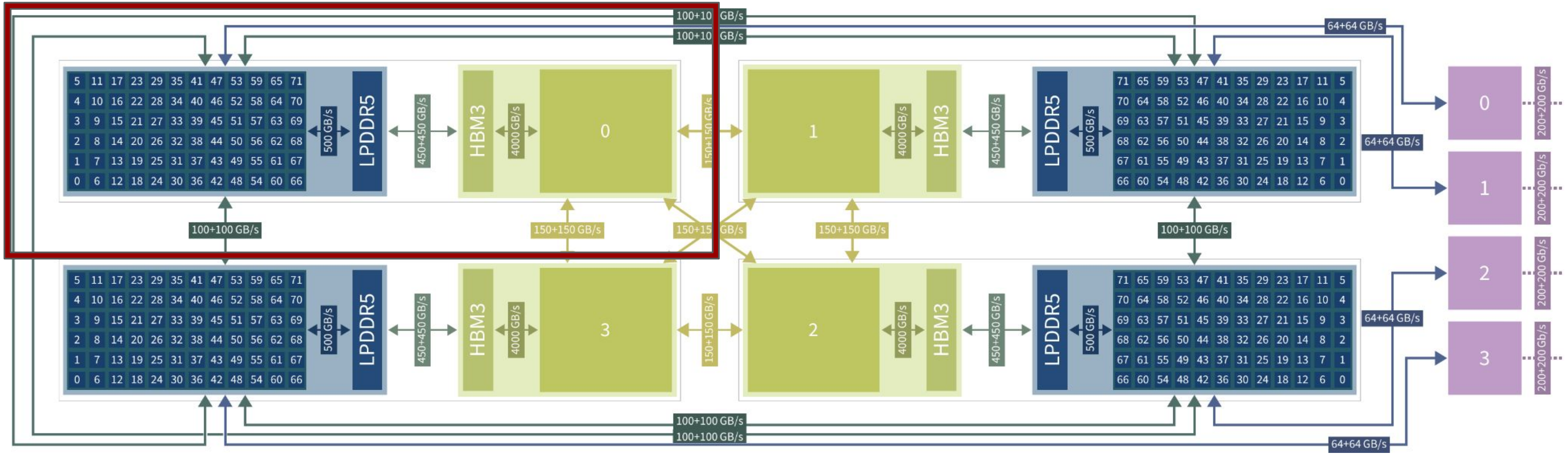


- Managed memory accessible in the same way from host/device
- Data are automatically copied to CPU/GPU as needed (there is also a prefetching API)
- Easier programming, but your own implementation could be faster

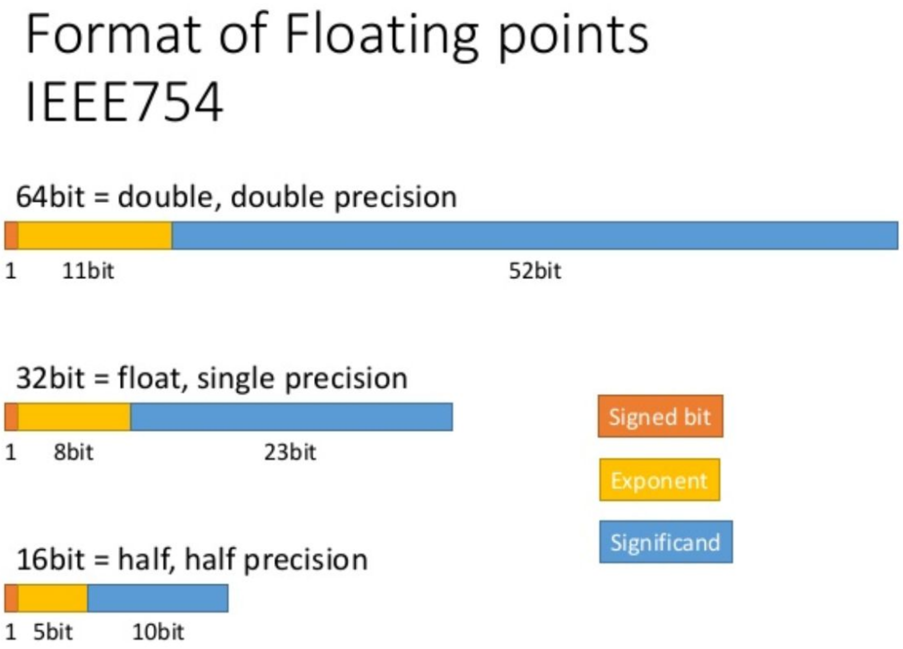
Unified memory: Grace–Hopper configuration



Unified memory: Grace–Hopper configuration



Half-precision data (FP16 since Pascal)

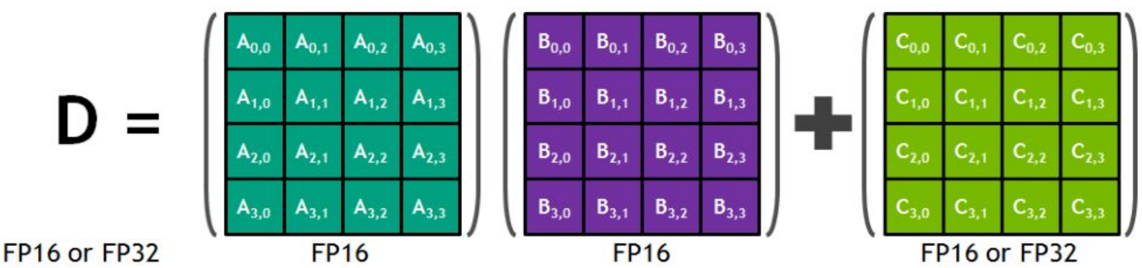


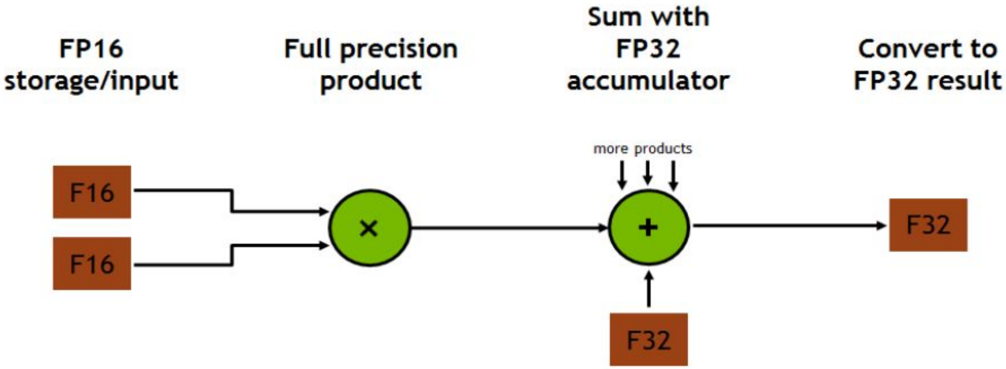
- GPU support low precision types
- ML does not require full FP64 precision (LLM, Image recognition..)
- …many more types over the years (int4, int8, fp4…)

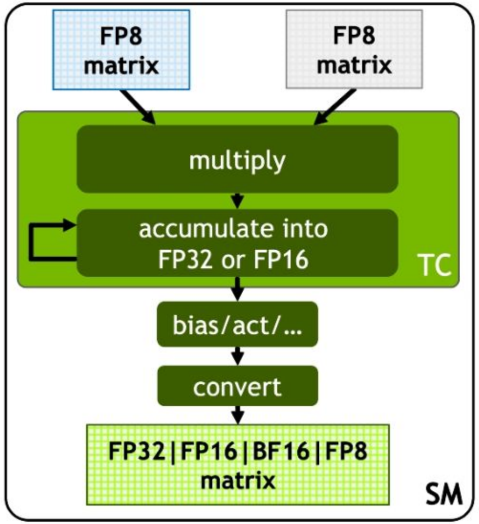
….with half precision you move twice the variables using the same amount of bytes!

Tensor cores and matrix operations

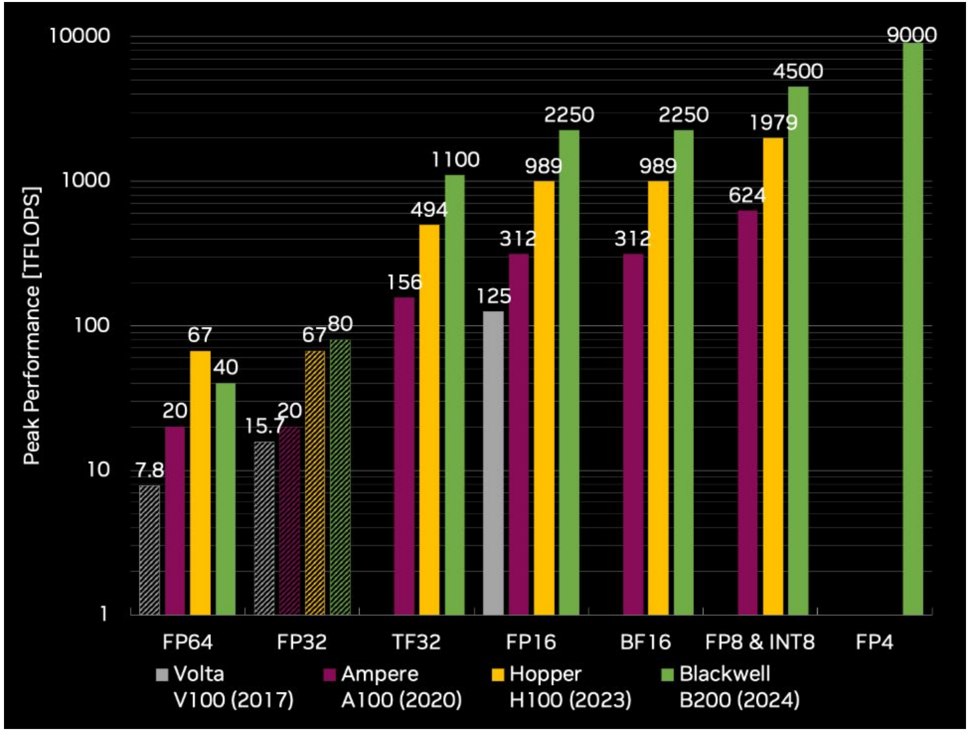
Matrix Multiplication: fundamental operation in DL







Tensor cores and matrix operations



Precision	Operation	Energy per FLOP (Matrix Multiply)
FP64	FMA	2.5x
FP32	**FMA**	**1.0x**
FP16	FMA	0.5x
FP64	Tensor Core MMA	1.5x
FP16	Tensor Core MMA	0.12x
FP8	Tensor Core MMA	0.06x
INT8	Tensor Core MMA	0.04x

THANK YOU FOR YOUR ATTENTION!