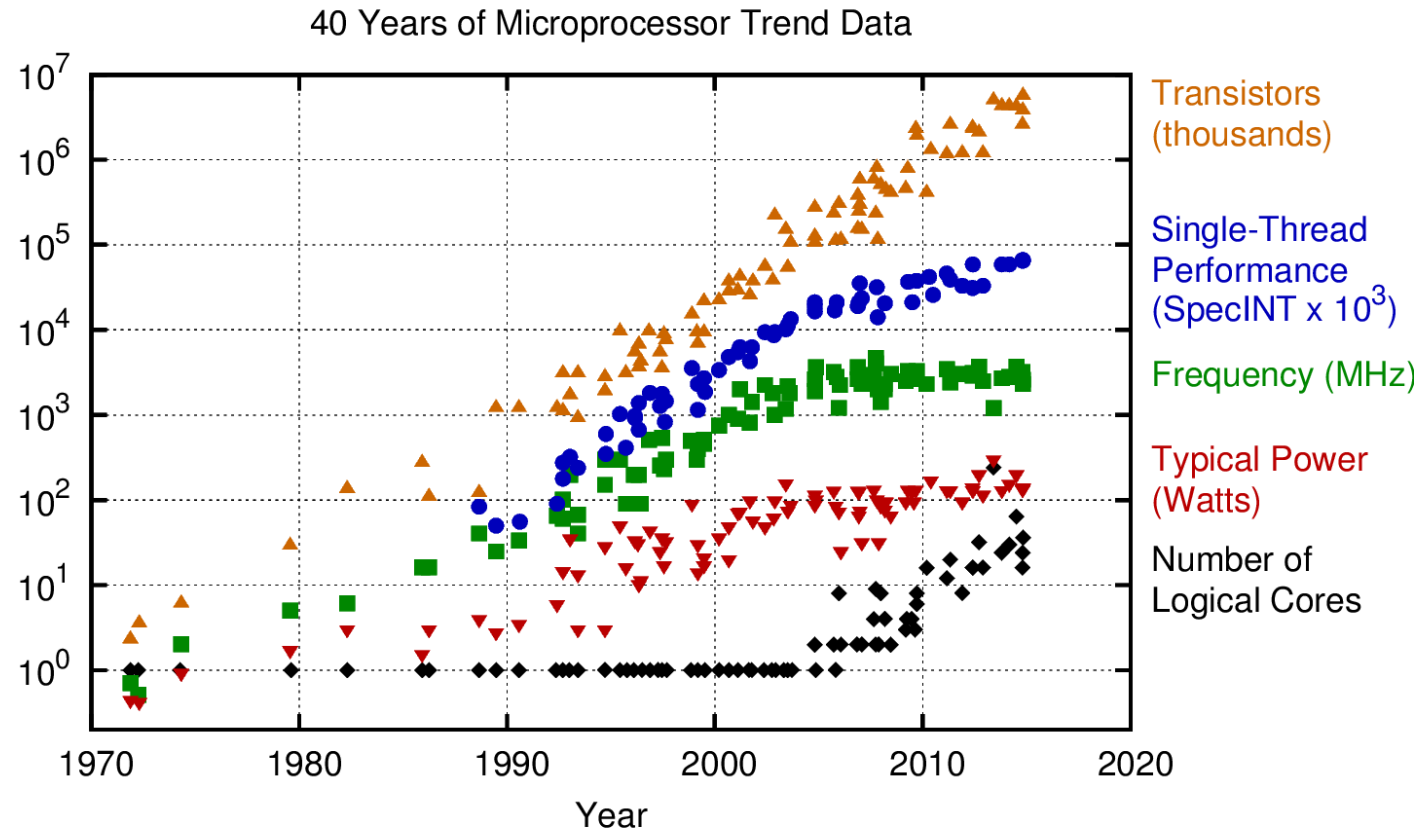


Overview of modern HW

Igor Sfiligoi – San Diego Supercomputing Center (UCSD)

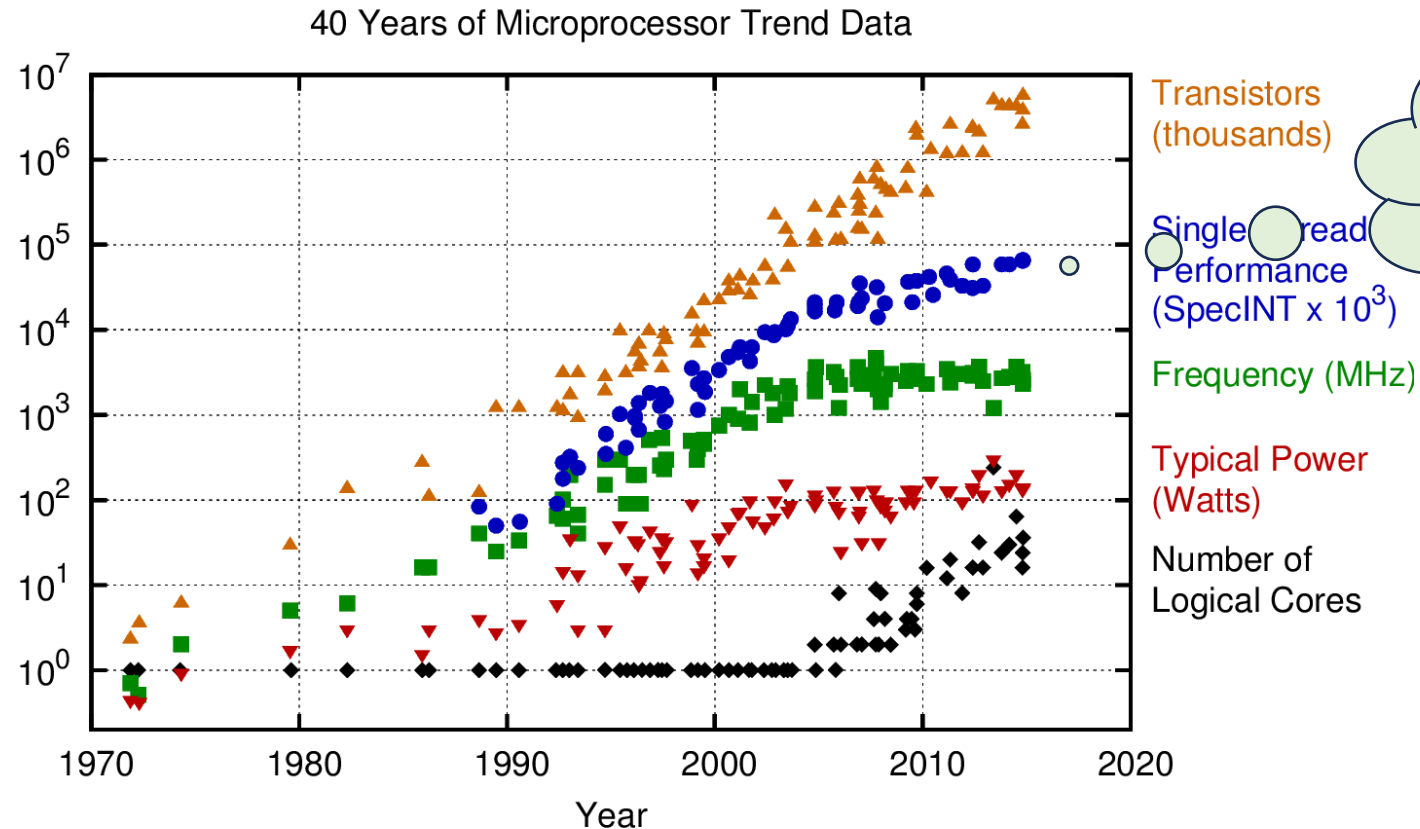
CPU cores stopped scaling long ago



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 2010-2015 by K. Rupp

Credits: <https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/>

CPU cores mostly stopped scaling long ago



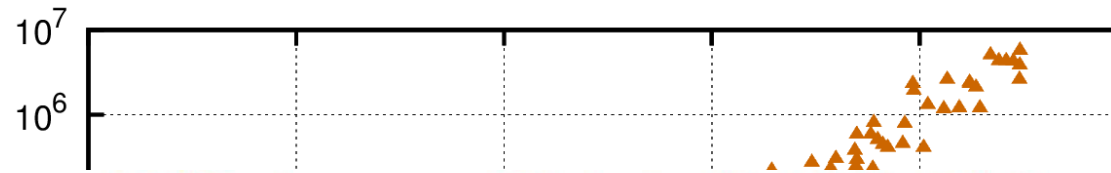
Not completely,
but progress slow

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 2010-2015 by K. Rupp

Credits: <https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/>

CPU cores mostly stopped scaling long ago

40 Years of Microprocessor Trend Data



Transistors (thousands)

Single thread performance (SpecINT x 10³)

Frequency (MHz)

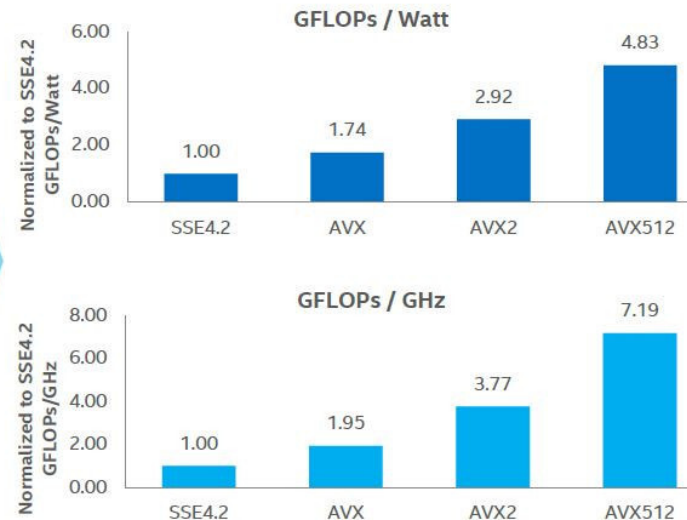
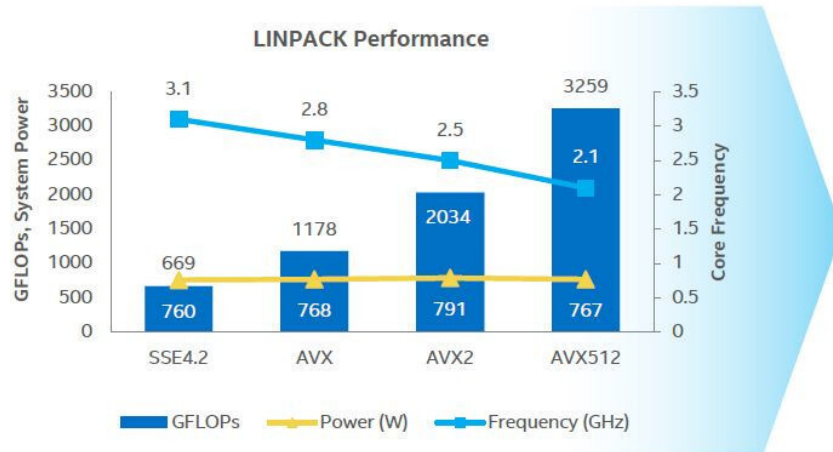
Typical Power (Watts)

Number of Logical Cores

Not completely, but progress slow

Most of that speedup due to vectorization

Performance and Efficiency with Intel® AVX-512

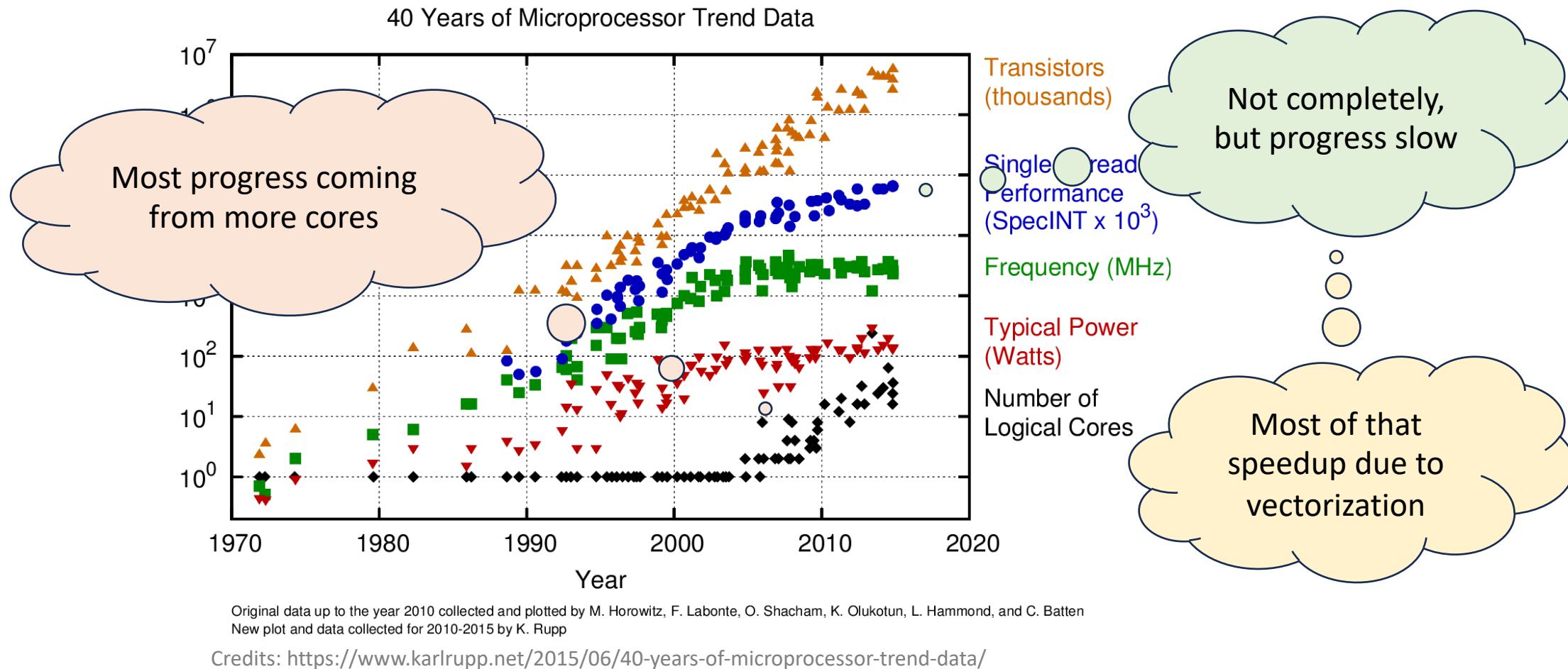


l, and C. Batten

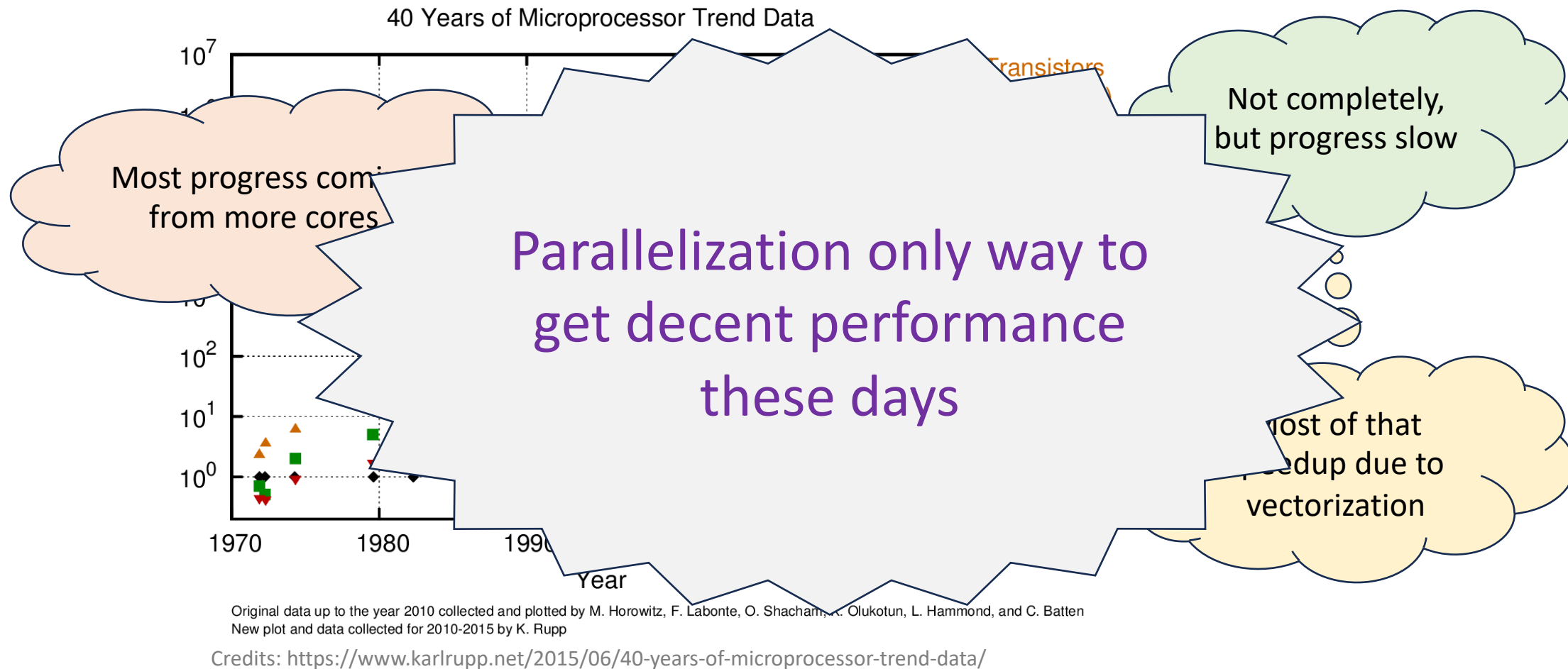
INTEL® AVX-512 DELIVERS SIGNIFICANT PERFORMANCE AND EFFICIENCY GAINS

Credits: <https://www.techpowerup.com/269770/linus-torvalds-finds-avx-512-an-intel-gimmick-to-invent-and-win-at-benchmarks>

But CPUs still scaling



But CPUs still scaling



GPUs becoming the preferred HPC processor

- Already in 2012
 - #1 Top500 ORNL Titan used NVIDIA K20X GPUs
 - Achieving 27 PFLOPs
 - Using 18.6k GPUs
 - #7 Top500 Stampede used INTEL Xeon E5-2680 CPUs
 - Achieved only 9.6 TFLOPS
 - Using 65k CPUs (522k cores)
- Today most Top500 systems GPU-based
 - #1 ORNL Frontier – AMD MI250X
 - 1700 PFLOPS over 136k GPUs
 - #2 ARNL Aurora – INTEL MAX
 - 2000 PFLOPS over 92k GPUs
 - #3 MS Eagle – NVIDIA H100
 - 850 PFLOS over 23.4k GPUs
 - CPU-based systems have lots of cores
 - #4 RIKEN Fugaku – Fujitsu A64FX
 - 530 PFLOPS over 160k CPUs (7.6M cores)

GPUs becoming the preferred HPC processor

- Already in 2012
 - #1 Top500 ORNL Titan used NVIDIA K20X GPUs
 - Achieving 27 PFLOPs
 - Using 18.6k GPUs
 - #7 Top500 Stampede used INTEL Xeon E5-2680 CPU
 - Achieved only 9.6 TFLOPS
 - Using 65k CPUs (522k cores)
- Today most Top500 systems GPU-based
 - #1 ORNL Frontier – AMD MI250X
 - 1600 PFLOPS over 136k GPUs
 - #2 ORNL Aurora – INTEL MAX
 - 1000 PFLOPS over 92k GPUs
 - #3 MS Eagle – NVIDIA H100
 - 840 PFLOS over 23.4k GPUs
 - CPU-based systems have lots of cores
 - #4 RIKEN Fugaku – Fujitsu A64FX
 - 530 PFLOPS over 160k CPUs (7.6M cores)

Why GPUs?

GPUs becoming the preferred HPC processor

- Already in 2012
 - #1 Top500 ORNL Titan used NVIDIA K20X GPUs
 - Achieving 27 PFLOPs
 - Using 18.6k GPUs
- Today most Top500 systems GPU-based
 - #1 ORNL Frontier – AMD MI250X
 - 1600 PFLOPS over 136k GPUs
 - #2 ORNL Aurora – INTEL MAX
 - 1000 PFLOPS over 92k GPUs
 - #3 MS Eagle – NVIDIA H100
 - 840 PFLOS over 23.4k GPUs
 - CPU-based systems have lots of cores
 - #4 RIKEN Fugaku – Fujitsu A64FX
 - 530 PFLOPS over 160k CPUs (7.6M cores)

Why GPUs?

An AMD Genoa CPU
has 96x512bit throughput
($96 \times 16 = 1.5\text{k}$ fp32 par. ops)

Many more but
simpler compute cores

A NVIDIA H100
has 16k CUDA cores

GPUs becoming the preferred HPC processor

- Already in 2012
- Today most Top500 systems GPU-based

- #1 Top500 ORNL Titan used NVIDIA K20X GPUs

- Achieving 27 PFLOPs
- Using 18.6k GPUs

An AMD Genoa CPU
has 96x512bit throughput
($96 \times 16 = 1.5\text{k}$ fp32 par. ops)

Many more but
simpler compute cores

Why GPUs?

A NVIDIA H100
has 16k CUDA cores

- #1 ORNL Frontier – AMD MI250X

1600 PFLOPS over 136k GPUs

ORNL Aurora – INTEL MAX

1000 PFLOPS over 92k GPUs

- #3 MS Eagle – NVIDIA H100

- 840 PFLOPS over 23.4k GPUs
(340M CUDA cores)

- CPU-based systems have lots of cores

- #4 Riken Fugaku – Fujitsu A64FX

530 PFLOPS over 160k CPUs (7.6M cores)

GPUs becoming the preferred HPC processor

- Already
- Today most Top500 systems GPU-based

Parallelization only way
to get decent
performance these days

- Achieving
- Using 18.6k GPUs

An AMD Genoa CPU
has 96x512bit throughput
($96 \times 16 = 1.5\text{k}$ fp32 par. ops)

Many more but
simpler compute cores

Why GPUs?

A NVIDIA H100
has 16k CUDA cores

#1 ORNL Frontier – AMD MI250X

1600 PFLOPS over 136k GPUs

ORNL Aurora – INTEL MAX

1000 PFLOPS over 92k GPUs

#3 MS Eagle – NVIDIA H100

- 840 PFLOPS over 23.4k GPUs
(340M CUDA cores)

- CPU-based systems have lots of cores

• #4 Riken Fugaku – Fujitsu A64FX

530 PFLOPS over 160k CPUs (7.6M cores)

Compute vs memory access

- All the FLOPS numbers of previous slides are "peak FLOPS"
 - Assume the processor has the data and is ready to operate on it
- **But where is the data coming from?**

Compute vs memory access

- All the FLOPS numbers of previous slides are "peak FLOPS"
 - Assume the processor has the data and is ready to operate on it
- But where is the data coming from?
 - Most programming languages have "variables"
 - Which are either scalars or buffers/vectors/matrices
 - **Either way, they need to be stored "somewhere"**

Compute vs memory access

- All the FLOPS numbers of previous slides are "peak FLOPS"
 - Assume the processor has the data and is ready to operate on it
- But where is the data coming from?
 - Most programming languages have "variables"
 - Which are either scalars or buffers/vectors/matrices
 - Either way, they need to be stored "somewhere"
- Processors operate on registers
 - Most processors limited to $O(100)$ per thread
 - Which is too small for most problems!
 - Virtually all data thus resident in system memory

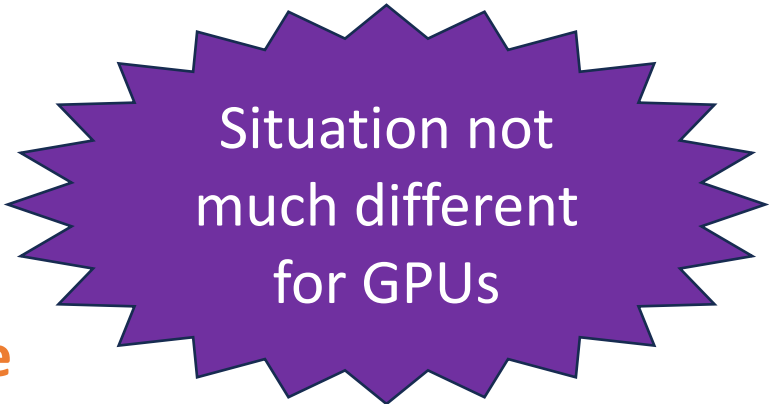
The cost of memory access

- Reading from memory takes a long time
 - **Over 300 clock cycles** on modern CPUs!
 - The CPU will be idle while waiting for the data
Unless you can schedule other compute alongside
- **Memory bandwidth into the CPU limited**, too
 - About 2 Tbps on modern CPUs (DDR5)
 - Would only allow 1kb of data to be read per cycle (at 2 GHz)
 - While the processor can compute a $96 \times 512 \text{b} = 50 \text{kb}$

50x

The cost of memory access

- Reading from memory takes a long time
 - **Over 400 clock cycles** on modern **GPUs**!
 - The **GPU** will be idle while waiting for the data
Unless you can schedule other compute alongside
- **Memory bandwidth into the GPU limited**, too
 - About 25 Tbps on modern GPUs (HBM3e)
 - Would only allow 16kb of data to be read per cycle (at 1.5 GHz)
 - While the processor can compute a $16k \times 32b = 512kb$



Situation not
much different
for GPUs

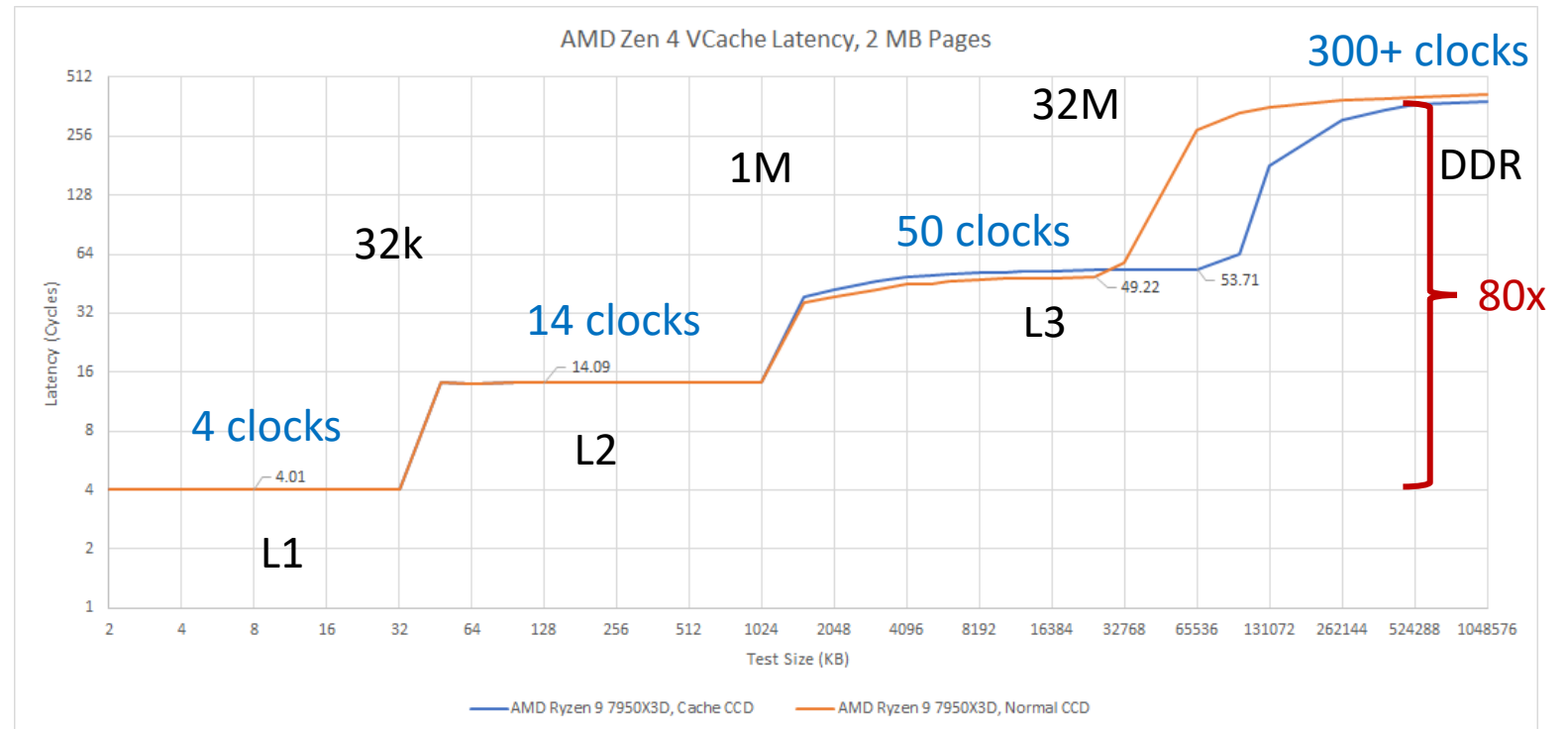
30x

The cost of memory access

- Reading from memory takes a long time
- Memory bandwidth into the CPU/GPU limited
- **Writing to memory is much slower**, still
 - You want to store those results, right?

Data caching to the rescue

- Most CPUs and GPUs have on-chip caches to compensate
 - If a "variable" is reused soon enough, it will be found in the cache
- Several levels
 - L1 - fast but small
 - L2+ - significantly larger, but slower
- Anything past L1 too slow for keeping all compute in use

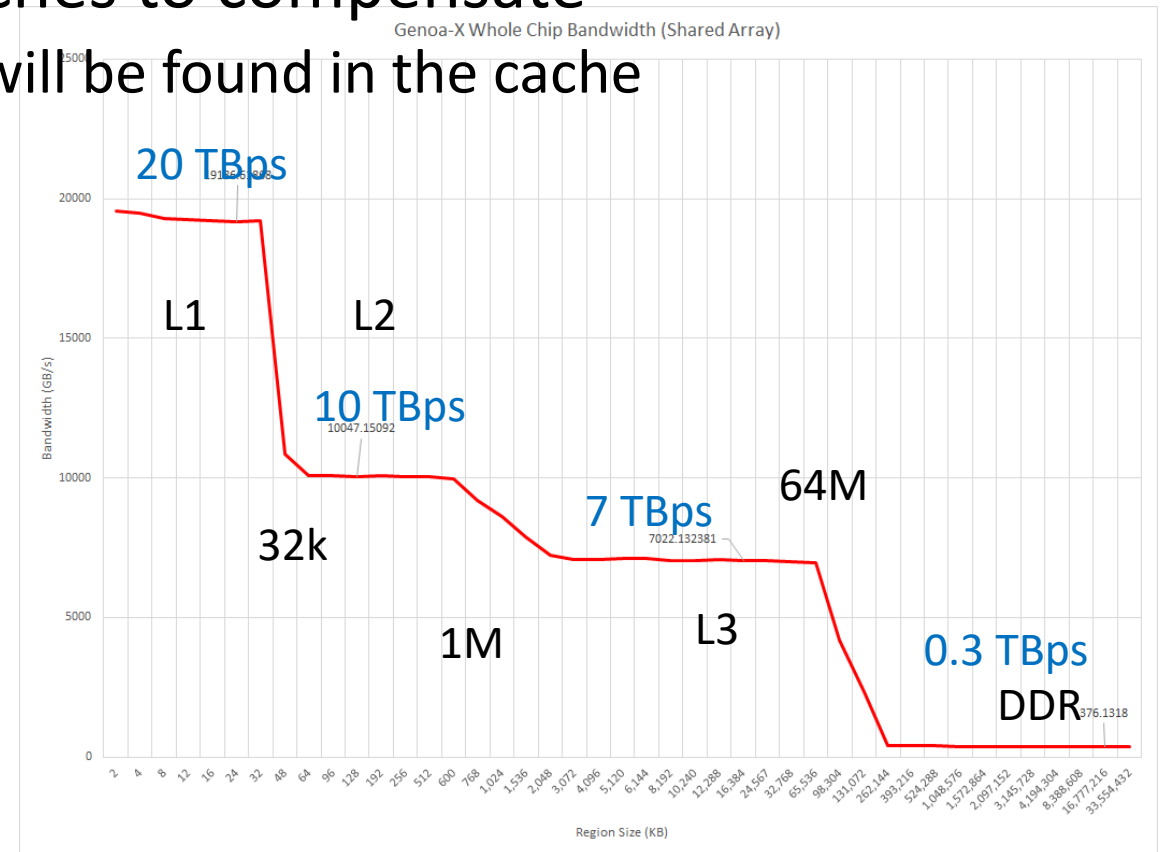


Credits <https://chipsandcheese.com/2023/07/17/genoa-x-server-v-cache-round-2/>

Data caching to the rescue

- Most CPUs and GPUs have on-chip caches to compensate
 - If a "variable" is reused soon enough, it will be found in the cache
- Several levels
 - L1 - fast but small
 - L2+ - significantly larger, but slower
- Anything past L1 too slow for keeping all compute in use

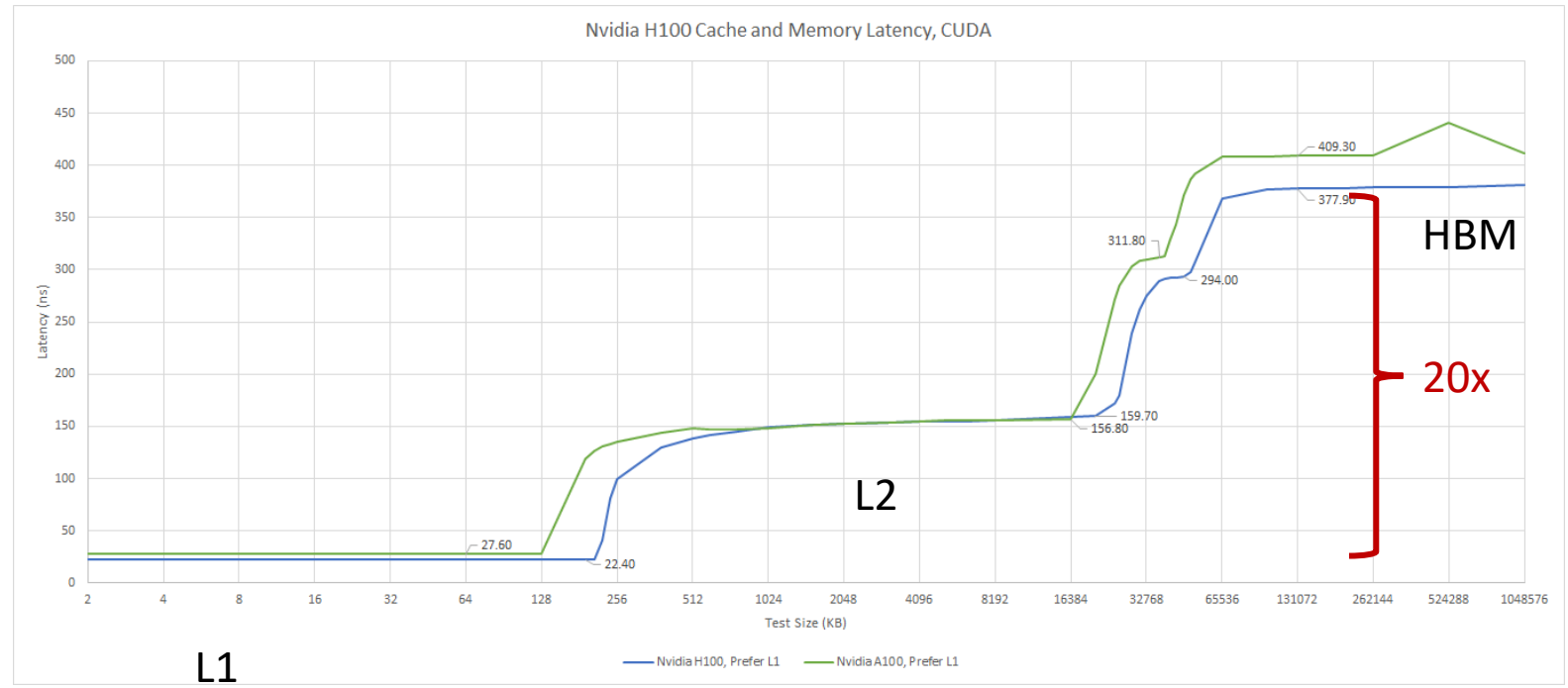
66x



Credits <https://chipsandcheese.com/2023/07/17/genoa-x-server-v-cache-round-2/>

Data caching to the rescue

- Most CPUs and GPUs have on-chip caches to compensate
 - If a "variable" is reused soon enough, it will be found in the cache
- Several levels
 - L1 - fast but small
 - L2+ - significantly larger, but slower
- Anything past L1 too slow for keeping all compute in use



Data caching to the rescue

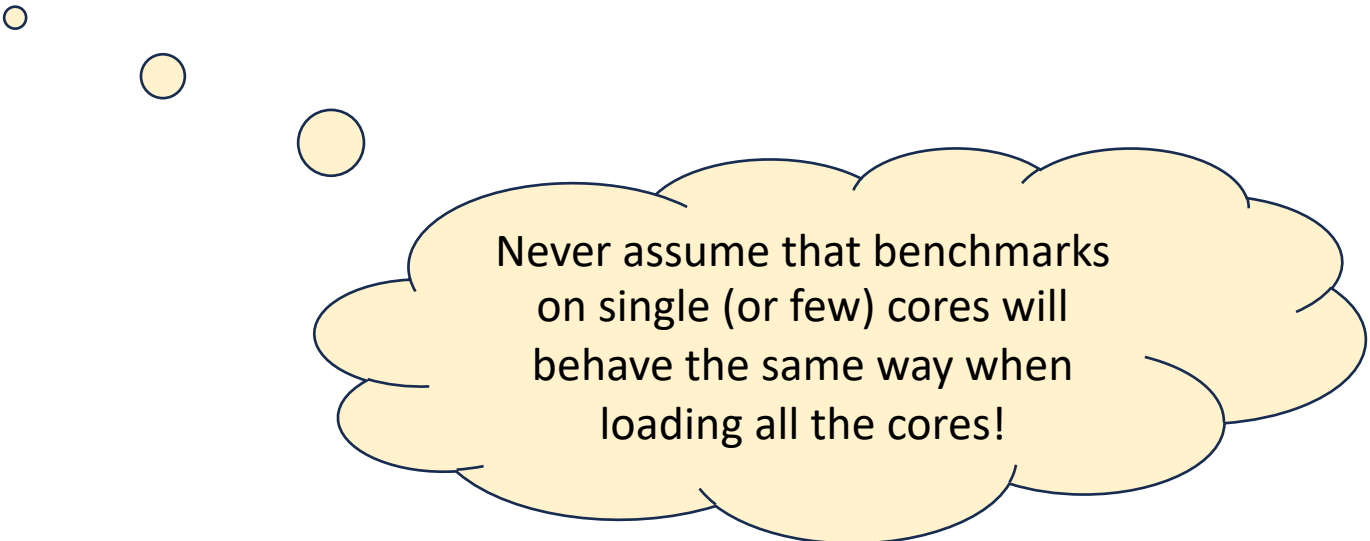
- Most CPUs and GPUs have on-chip caches to compensate
 - If a "variable" is reused soon enough, it will be found in the cache
- Several levels
 - L1 - fast but small
 - L2+ - significantly larger, but slower
- Anything past L1 too slow for keeping all compute in use

	NVIDIA H100	NVIDIA A100
L1 size	256 KB	192 KB
L1 bandwidth	100 TBps	40 TBps
L2 size	50 MB	40 MB
L2 bandwidth	5.5 TBps	4.8 TBps
HBM bandwidth	3 TBps	1.5TBps

Credits <https://chipsandcheese.com/2023/07/02/nvidias-h100-funny-l2-and-tons-of-bandwidth/>
<https://www.nextplatform.com/2022/03/31/deep-dive-into-nvidias-hopper-gpu-architecture/>

Per-core and global throughput

- L1 caches local to the “core”, aka compute unit
 - If all compute can be done on data in L1, no interference between cores
- Once we need data from outside the per-core cache(s)
 - Cores start to compete for memory bandwidth



Never assume that benchmarks
on single (or few) cores will
behave the same way when
loading all the cores!

Parallelism necessary for speed

- All modern processors require massive parallelism for speed
 - A modern CPU can have 48k bits of parallelism (e.g., 1.5k fp32 ops)
 - A modern GPU can have 512k bits of parallelism (e.g., 16k fp32 ops)
- **You need to structure your algorithms to be inherently parallel**

Parallelism necessary for speed

- All modern processors require massive parallelism for speed
 - A modern CPU can have 48k bits of parallelism (e.g., 1.5k fp32 ops)
 - A modern GPU can have 512k bits of parallelism (e.g., 16k fp32 ops)
- You need to structure your algorithms to be inherently parallel
- **Large buffers/vectors/arrays/meshes should make that trivial,** as long as
 - a) You reuse elements while in cache (else you are just blocked on memory)
 - b) There are no dependencies between operations
 - c) There is no logic (i.e. if statements) involved

Parallelism necessary for speed

- All modern processors require massive parallelism for speed
 - A modern CPU can have 48k bits of parallelism (e.g., 1.5k fp32 ops)
 - A modern GPU can have 512k bits of parallelism
- You need to structure your algorithm
- Large buffers/vectors/arrays/meshes in memory as long as

- a) You reuse elements while in cache (else you are just blocked on memory)
- b) There are no dependencies between operations
- c) There is no logic (i.e. if statements) involved



OK, not trivial at all!

(but still doable for many problems)

Not all compute is the same (when you run parallel)

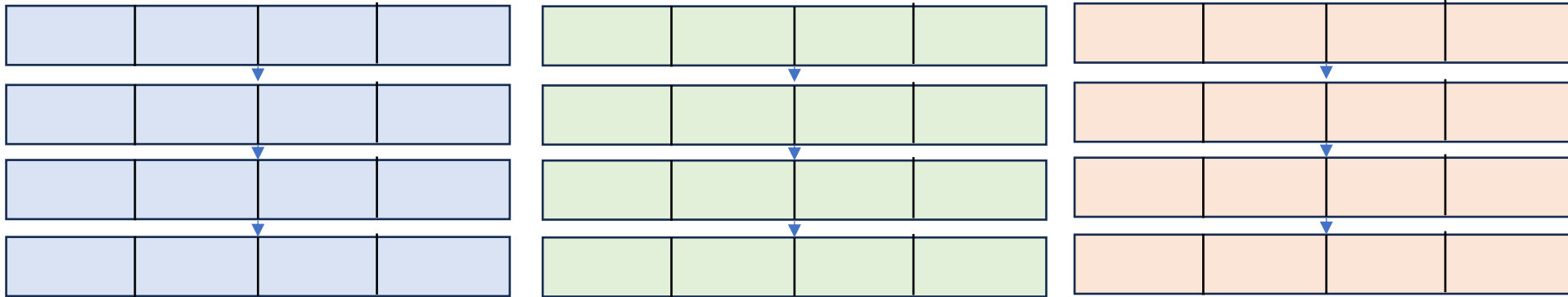
- Most modern processors have specialized compute units for various compute types
 - Do you really need max fp64 precision?
 - Can you fit the problem as a series of tensor/matrix operations?
- True for both CPU and GPU compute
 - CPU vectors fixed bit size – **8x64** vs **16x32**
 - Most GPUs optimized for lower precision
 - Also, some consumer GPUs virtually lack fp64 compute

Form Factor	H100 SXM	
FP64	34 teraFLOPS	30x
FP64 Tensor Core	67 teraFLOPS	
FP32	67 teraFLOPS	
TF32 Tensor Core	989 teraFLOPS ²	120x
BFLOAT16 Tensor Core	1,979 teraFLOPS ²	
FP16 Tensor Core	1,979 teraFLOPS ²	
FP8 Tensor Core	3,958 teraFLOPS ²	
INT8 Tensor Core	3,958 TOPS ²	

Credits: <https://www.nvidia.com/en-us/data-center/h100/>

CPU and GPU parallelism different

- In an idealized world, they are both **many independent compute units processing vector operations**



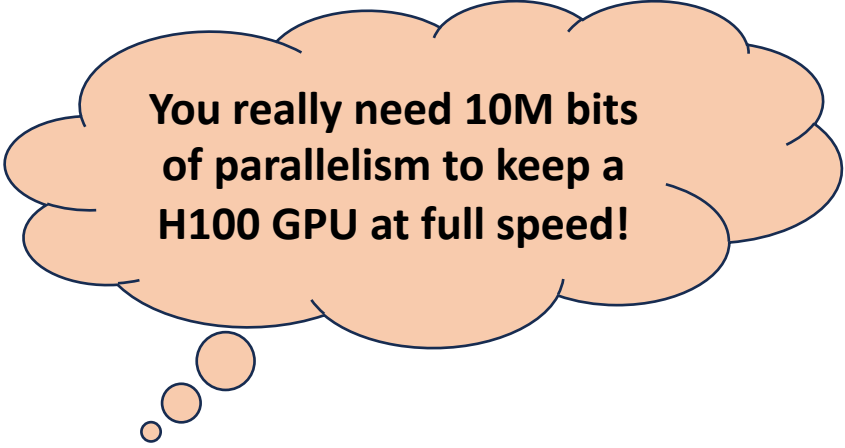
- But real life is rarely like that

CPU and GPU parallelism different

- In an idealized world, they are both **many independent compute units processing vector operations**
- To deal with real-life complexity:
 - CPUs are optimized for logic
 - Sophisticated branch prediction logic
 - Speculative execution of branches, too
 - Variable vector width
 - GPUs are optimized for massive parallelism
 - Almost-free context switching
 - Huge register count
 - Hardware sub-vector branching

CPU and GPU parallelism different

- In an idealized world, they are both **many independent compute units processing vector operations**
- To deal with real-life complexity:
 - CPUs are optimized for logic
 - Sophisticated branch prediction logic
 - Speculative execution of branches, too
 - Variable vector width
 - GPUs are optimized for massive parallelism
 - Almost-free context switching
 - Huge register count
 - Hardware sub-vector branching



You really need 10M bits of parallelism to keep a H100 GPU at full speed!

Assumes 10-100 vectors per compute unit ready to be executed at any time

Disclaimer: Other GPUs are similar, just picking one as an example.

Execution units

CPU

- Composed of “**cores**” (up to 96)
- Each with a sequential “thread”
- Each step one instruction:
 - Scalar (add, bool, fp)
 - Logic (if, jump, call)
 - Vector (up to 512 bits)
 - Memory (load, store, 8-512 bits)
- Hyperthreading = parallel thread
 - Independent, but same HW

GPU

- Composed of “streaming multiprocessors” (**SMs**) (up to 144)
- Each can schedule hundreds of “**wavefronts**” (~threads)
 - Picks one that has data ready
- Each step executes a vector instr.
 - For all the ops (add, if, load, etc.)
- HW splits vector on diverging logic
 - With auto-merge when converging

Execution units

CPU

- Composed of “**cores**” (processors)
- Each with a sequential execution unit
- Each step one instruction
 - Scalar (add, bool, fp)
 - Logic (if, jump, call)
 - Vector (up to 512 bits)
 - Memory (load, store, 8-512 bits)
- Hyperthreading = parallel thread
 - Independent, but same HW

GPU

Approximately
the same concept

- Composed of “streaming processors” (**SMs**)
- Can schedule hundreds of “**wavefronts**” (~threads)
 - Picks one that has data ready
- Each step executes a vector instr.
 - For all the ops (add, if, load, etc.)
- HW splits vector on diverging logic
 - With auto-merge when converging

But CPUs support
only a subset of
scalar operations
as vectors

(compilers often fall back
on scalar compute)

- Each one instruction
 - Scalar (i, d, bool, fp)
 - Logic (l, jump, call)
 - Vector (up to 512 bits)
 - Memory (load, store, 8-512 bits)
- Hyperthreading = parallel thread
 - Independent, but same HW

GPU

Approximately
the same concept

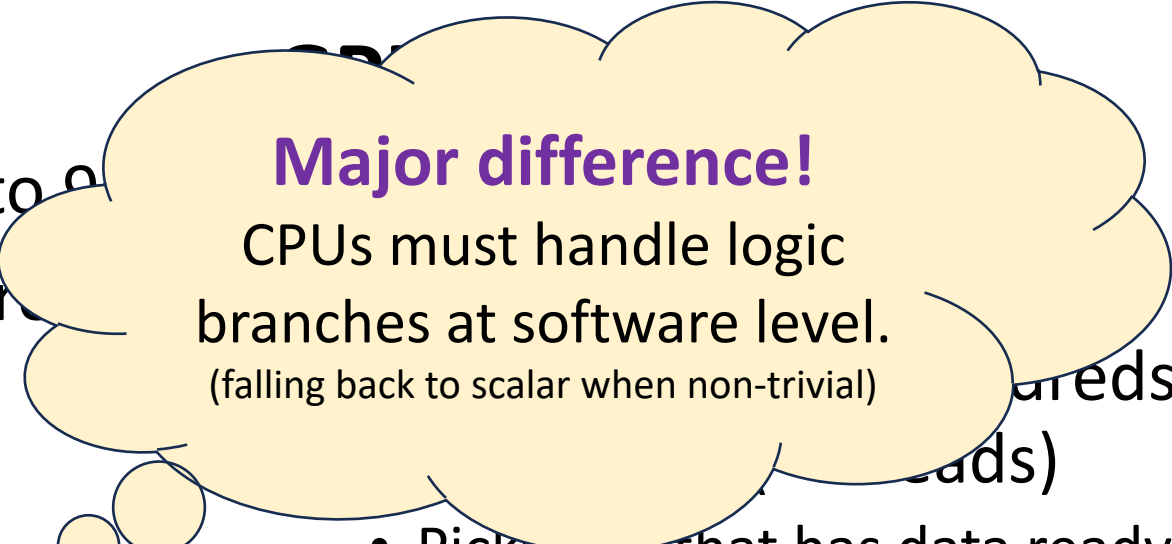
made of “streaming
processors” (**SMs**)

can schedule hundreds of
wavefronts (~threads)

- Picks one that has data ready
- Each step executes a vector instr.
 - For all the ops (add, if, load, etc.)
- HW splits vector on diverging logic
 - With auto-merge when converge

Execution units

CPU

- Composed of “cores” (up to 96)
 - Each with a sequential “thread”
 - Each step one instruction:
 - Scalar (add, bool, fp)
 - Logic (if, jump, call)
 - Vector (up to 512 bits)
 - Memory (load, store, 8-512 bits)
 - Hyperthreading = parallel thread
 - Independent, but same HW
- 
- Major difference!**
CPUs must handle logic branches at software level.
(falling back to scalar when non-trivial)
- Picks one that has data ready
 - Each step executes a vector instr.
 - For all the ops (add, if, load, etc.)
 - HW splits vector on diverging logic
 - With auto-merge when converge

Execution units

CPU

- Composed of “cores” (up to 100)
- Each with a sequential “thread”
- Each step one instruction:
 - Scalar (add, bool, fp)
 - Logic (if, jump, call)
 - Vector (load, store, add, sub, mul, div)

GPUs more flexible,
but can get slow with
nested logic.

Major difference!

CPU must handle logic
branches at software level.
(falling back to scalar when non-trivial)

- Picks one that has data ready
- Each step executes a vector instr.
 - For all the ops (add, if, load, etc.)
- HW splits vector on diverging logic
 - With auto-merge when converge

Execution units

CPU

- Composed of “**cores**” (up to 96)
- Each with a sequential “thread”
- Each step...

Major difference!

GPUs hide memory latency, although they require much higher parallelism.

- Hyperthreading = parallel thread
 - Independent, but same HW

GPU

- Composed of “streaming multiprocessors” (**SMs**)
- Each can schedule hundreds of “**wavefronts**” (~threads)
 - Picks one that has data ready
- Each step executes a vector instr.
 - For all the ops (add, if, load, etc.)
- HW splits vector on diverging logic
 - With auto-merge when converge

Execution units

CPU

- Composed of “**cores**” (up to 96)
- Each with a sequential “thread”
- Each stream of instructions

Major difference!

GPUs hide memory latency, although they require much higher parallelism.

- Hyperthreading = parallel thread
 - Independent, but same HW

GPU

- Composed of “streaming multiprocessors” (**SMs**)
- Each can schedule hundreds of “**wavefronts**” (~threads)
 - Picks one that has data ready
- Each SM can execute many threads

CPU Hyperthreading a step in the same direction. But not used much.

Memory coherency

CPU

- Goes to great length to provide memory coherency
 - Everywhere from L1 cache to main memory
- Guaranteeing access order just a software problem
- Low-cost atomic operations
 - Between cores in a CPU
 - Often between CPUs, too

GPU

- Uses a very relaxed coherency philosophy
 - Each L1 cache independent
 - Must be explicitly flushed to propagate writes to other SMs and main memory
- Atomics cheap only inside a SM
 - Expensive across whole GPU

CPU always in charge

- GPUs do not have an operating system on their own
 - Not at this time, anyway
- GPU execution is thus orchestrated by CPU processes
 - Equivalent to launching "a complex function" – called a "**GPU kernel**"
 - Significant overhead at each invocation (2-10 microseconds, >1k cycles)
- Mixing CPU and GPU code must be done carefully
 - Ensure GPU kernel execution time >> overhead
- CPU does not need to wait for kernel completion
 - Wait on kernel completion is explicit (although some SW may hide it)
 - May launch many kernels without waiting
 - May do CPU-based compute while GPU computes

} HW fully async

Partitioned memory space

- In most systems, CPU and GPU **memory completely separate**
 - May not be able to directly access CPU memory from GPU (or the other way)
 - **Users must explicitly move data between the two (no coherency!)**
 - And that data movement is slow (60 GBps vs 2TBps of HBM or 300 GBps of DDR)

Memory virtualization

- In most systems, CPU and GPU memory completely separate
 - Users must explicitly move data between the two (no coherency!)
 - And that data movement is slow (60 GBps vs 2TBps of HBM or 300 GBps of DDR)
- Many modern systems now allow for “**managed memory**”
 - Users see system and GPU memory as being the same
 - Under the hood, **uses page-fault logic**
 - **Either** CPU or GPU owns the logical memory page (typically 4kB)
 - HW will move it to appropriate CPU or GPU physical memory
 - Still limited by CPU-GPU memory-bus bandwidth
 - **Can result in trashing similar to disk-based swap space**

True shared memory

- In most systems, CPU and GPU memory completely separate
 - Users must explicitly move data between the two (no coherency!)
- Many modern systems now allow for “**managed memory**”
 - Users see system and GPU memory as being the same (swap-like)
- New class of hybrid CPU-GPUs becoming available (e.g. NVIDIA Grace-Hopper and AMD MI300A)
 - CPU and GPU **have a common virtual memory space**
 - CPU can access GPU memory and the way around, NUMA-like

True shared memory

- In most systems, CPU and GPU memory are separate
 - Users must explicitly move data between them
- Many modern systems now have unified memory
 - Users see system and GPU memory as a single pool
- New class of hybrid CPU-GPUs becoming available (e.g. NVIDIA Grace-Hopper and AMD MI300A)
 - CPU and GPU **have a common virtual memory space**
 - CPU can access GPU memory and the way around, NUMA-like

Extremely promising!

But I would not yet optimize for them.

That's all for high level overview