Lecture 9

Computations on graphics processors

Ingemar Ragnemalm

Information Coding, ISY

Did you find it amazing to run on 8 cores in a single desktop?

Did you find it amazing to run on 8 cores in a single desktop?

How about doing that with 10240 cores?

This lecture:

Plan for this part of the course

GPU evolution

GPU architecture

A first intro to general computing solutions with GPUs

My part of the course:

5 lectures

1 lesson

3 labs

Local sub-page: http://computer-graphics.se/TDDD56/

Lectures:

9. GPU evolution and architecture

10. Intro to CUDA

11. CUDA memory, threads, synchronization

12. More CUDA, sorting on GPU

13. Intro to OpenCL. Computing with shaders

Labs:

4. CUDA

5. Image filter with CUDA

6. OpenCL

No lab reports,
demonstrations during the lab

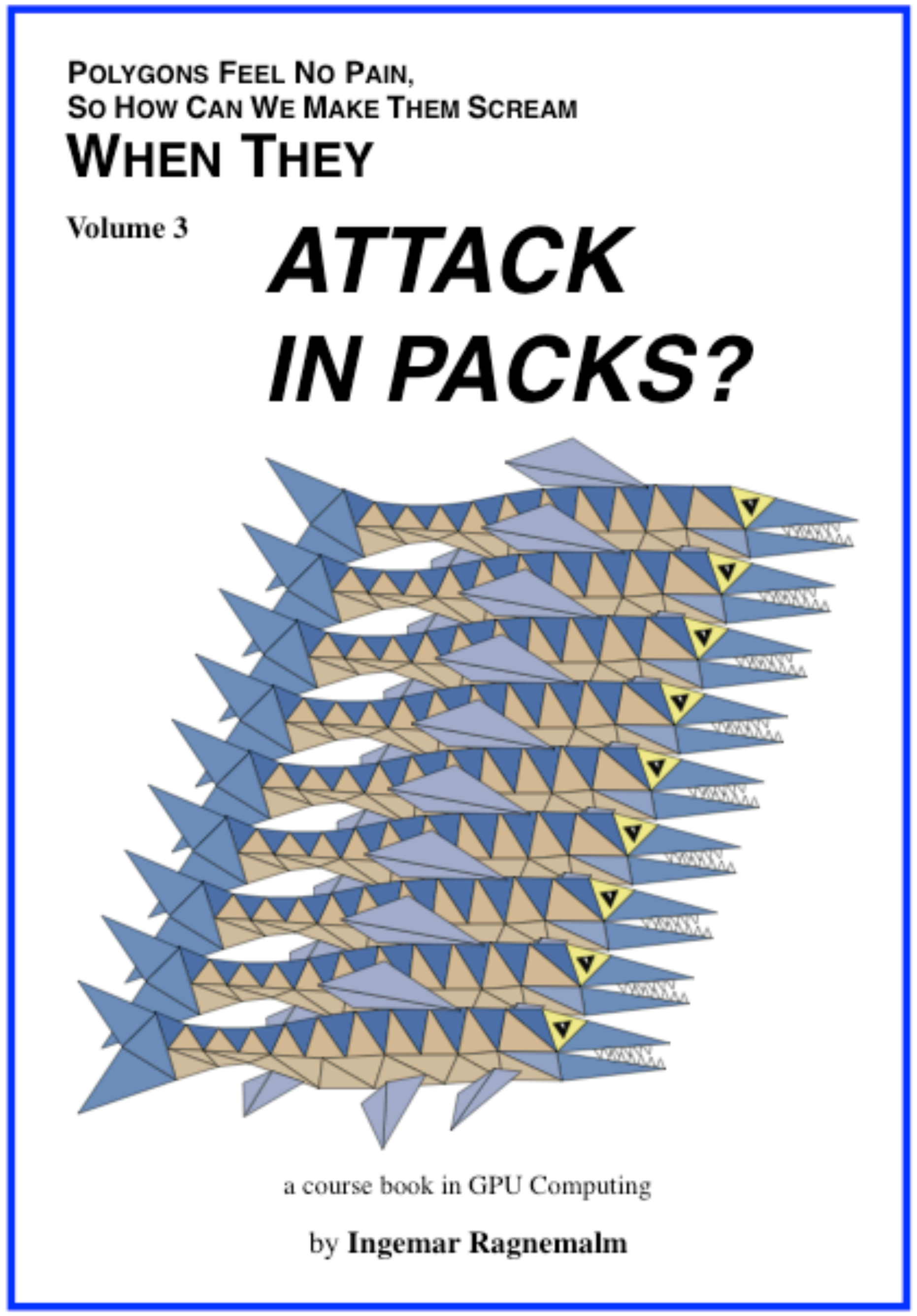
Literature for this part:

ATTACK IN PACKS

Available at Bokakademin

Inexpensive!

Also on-line (free!)



POLYGONS FEEL NO PAIN,
SO HOW CAN WE MAKE THEM SCREAM
WHEN THEY

Volume 3

ATTACK
IN PACKS?

a course book in GPU Computing

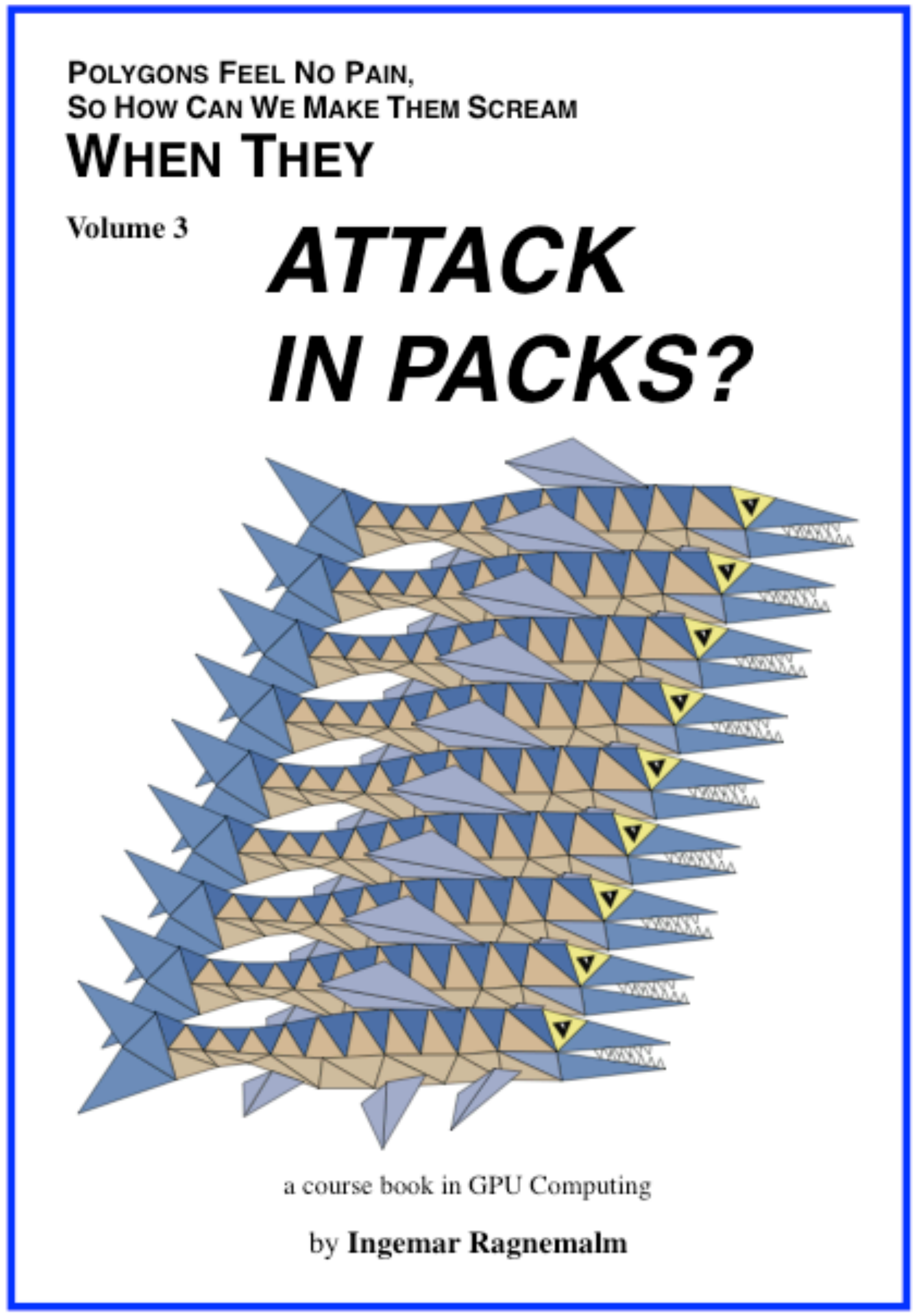
by **Ingemar Ragnemalm**

Printed version: 100kr

Online version here:

http://computer-graphics.se/TDDD56/

You decide what you need!



POLYGONS FEEL NO PAIN,
SO HOW CAN WE MAKE THEM SCREAM
WHEN THEY
Volume 3
ATTACK
IN PACKS?

a course book in GPU Computing

by **Ingemar Ragnemalm**

Questions

1. How can a GPU be much faster than a CPU?

2. Why is the G80 so much faster than the previous GPUs (e.g. 7000 series)?

3. A texturing unit provides access to texture memory. What more is it than just another memory?

4. What current trend is driven by the GPU evolution?

The decline of CPU evolution

Three ”walls”:

The decline of CPU evolution

Three ”walls”:

Tenessee Waltz

Max Wall

Wall-E

The decline of CPU evolution

Three ”walls”:

The decline of CPU evolution

Three ”walls”:

Power wall

Memory wall

ILP wall

The decline of CPU evolution

Three ”walls”:

Power wall

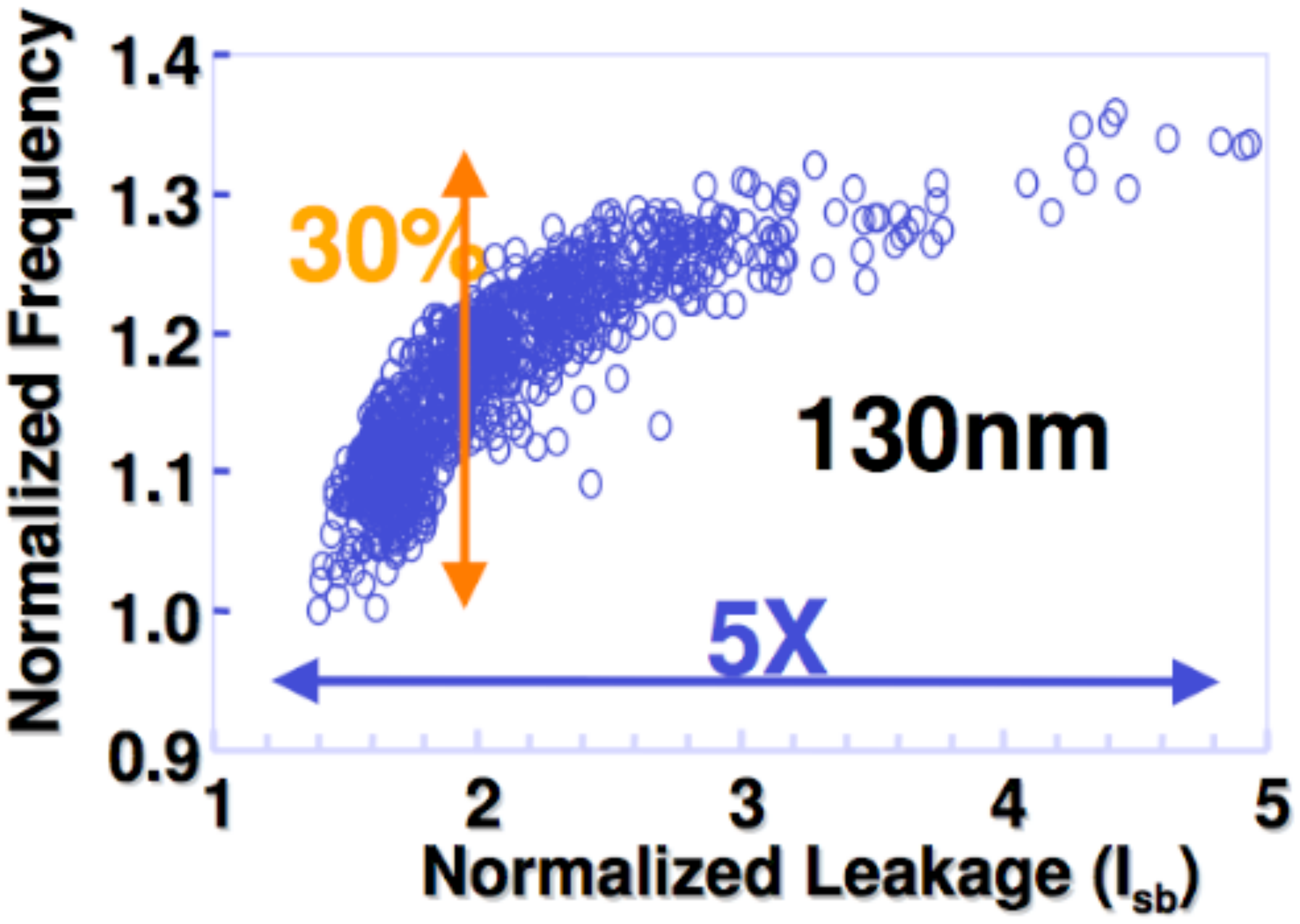
Memory wall

ILP wall

- **Clock frequency can no longer go up**
- **The memory architecture is insufficient**
- **Attempts to parallelize have failed**

Power wall

13% higher frequency = 73% more (almost double) double power consumption!



Power wall

Reverse reasoning: Lower frequency a little, win much power.

Replace one high-frequency CPU with two slightly slower - for the same cost!

Works nicely for two CPUs.

Intel promises 80 cores in a few years

BUT

this will run into the ”memory wall”

Memory wall

Already, the memory is slower than the CPU.

With more and more CPUs fighting for accessing the same RAM and caches, efficiency will degrade!

Memory bandwidth helps - if we can get it.

ILP wall

Instruction level parallelism

Writing parallel code is complicated.

Many problems are sequential by nature - or traditionally expressed as such.

ILP wall

Instruction level parallelism

Writing parallel code is complicated.

Many problems are sequential by nature - or traditionally expressed as such.

Solutions:

- **Explore algorithms in search of parallel solutions**
- **Learn how to code in parallel**
- **New programming paradigms, not optimizing for the programmer but for the computer!**

Timeline for CPUs

80’s: CPU and system same speed. Zero wait states.

1993: CPUs faster than the rest of the system. Rapid raise of frequency.

Late 90’s to present: Multi-CPU systems, multi-core CPUs.

CPUs are still improving, but going for higher frequency is not as obvious as before.

Meanwhile, at the graphics dept

80’s: Hardware sprites. Push pixels with low-level code.

1993: Textured 3D games: Wolf3D, Doom.

Early 90’s: Professional 3D boards.

1996: 3dfx Voodoo1!

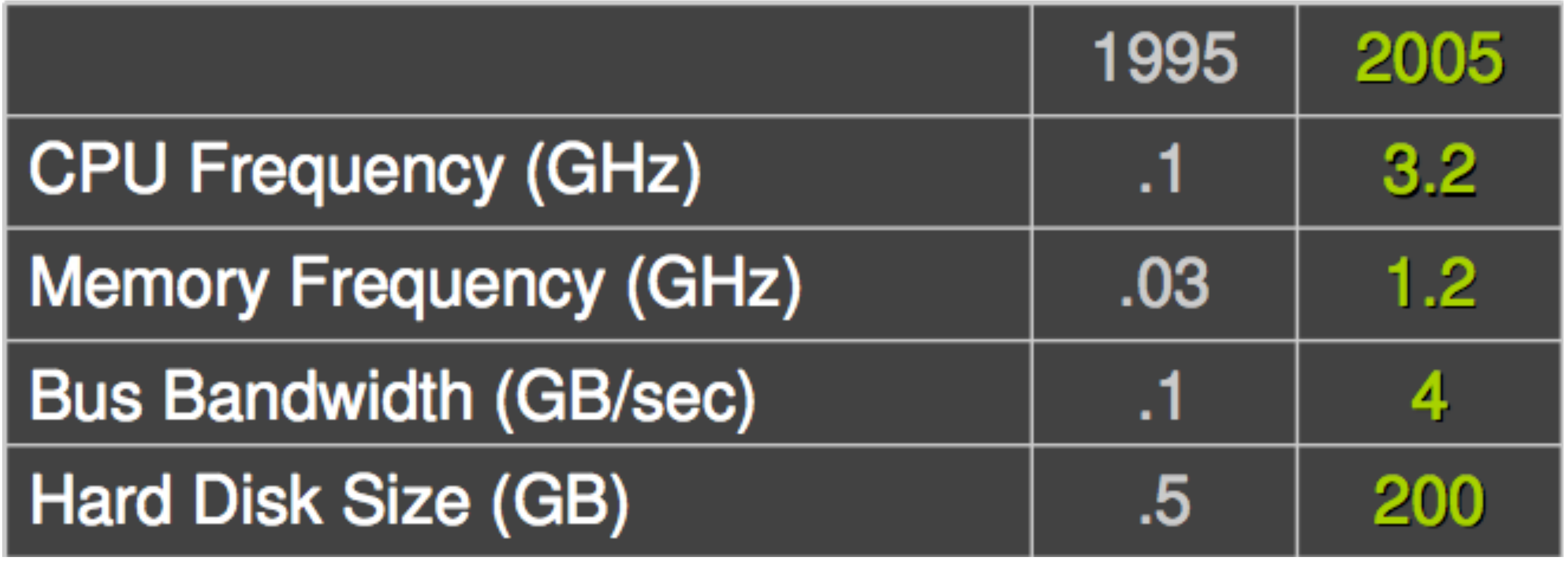
2001: Programmable shaders.

2006: G80, unified architecture. CUDA.

2009: OpenCL.

2010: Fermi architecture

2012-2021: Kepler, Maxwell, Pascal, Turing, Ampère...



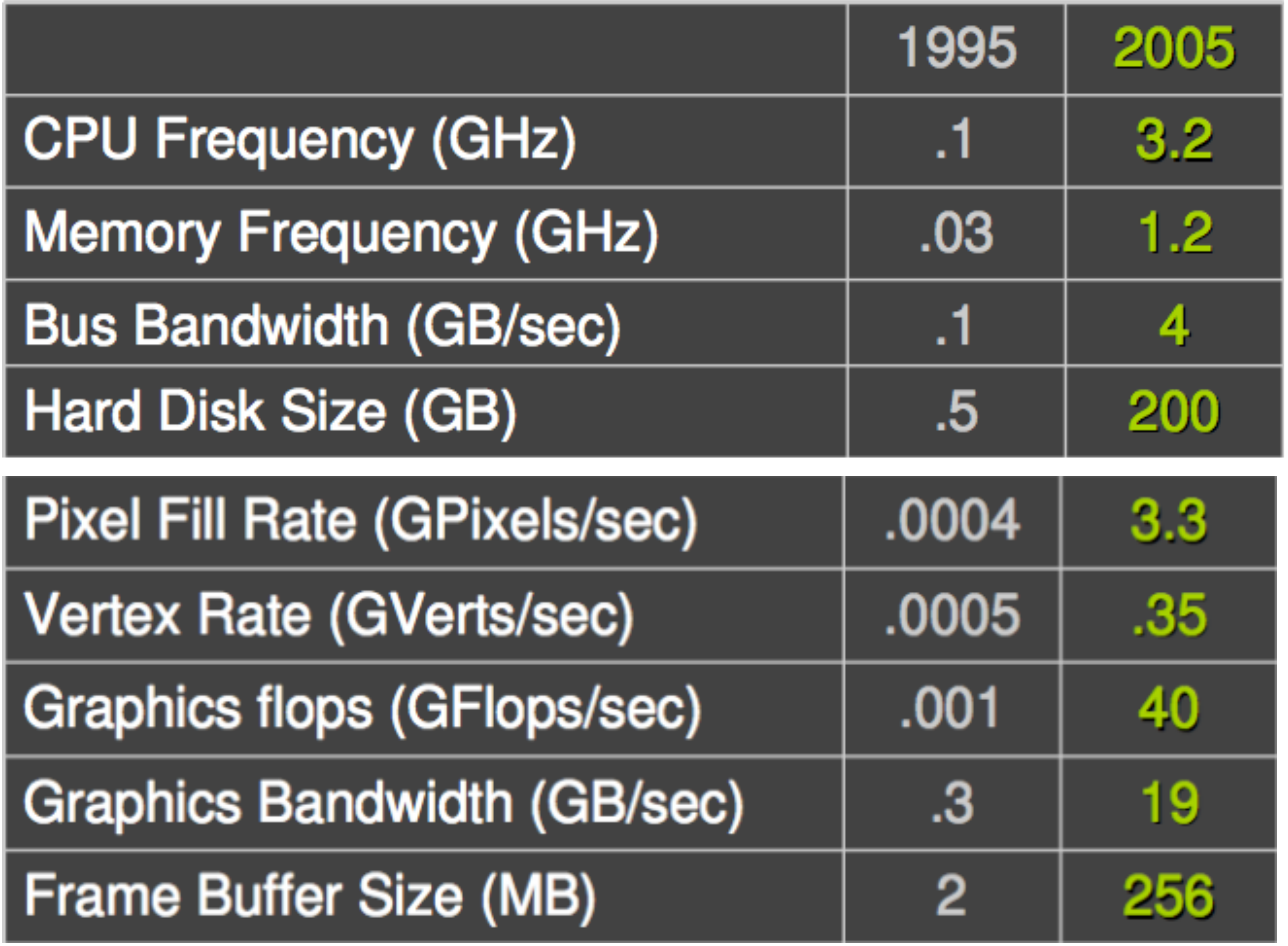
	1995	2005
CPU Frequency (GHz)	.1	3.2
Memory Frequency (GHz)	.03	1.2
Bus Bandwidth (GB/sec)	.1	4
Hard Disk Size (GB)	.5	200

32x

40x

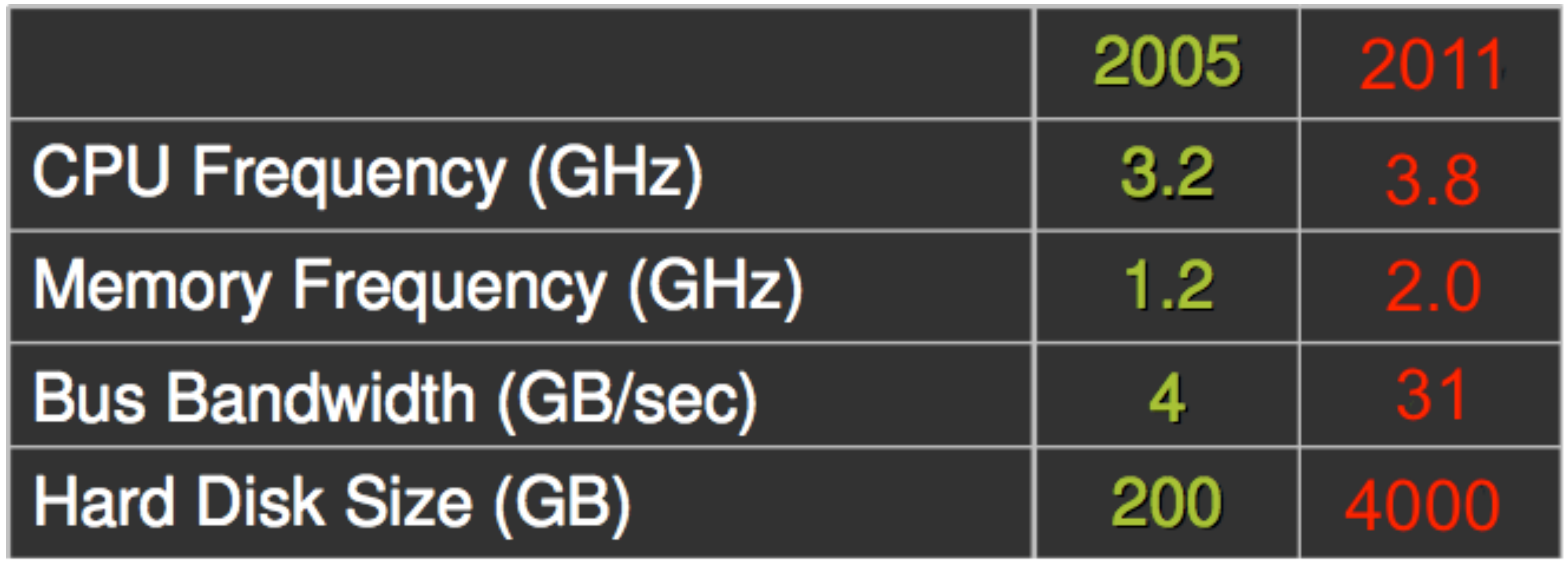
40x

400x

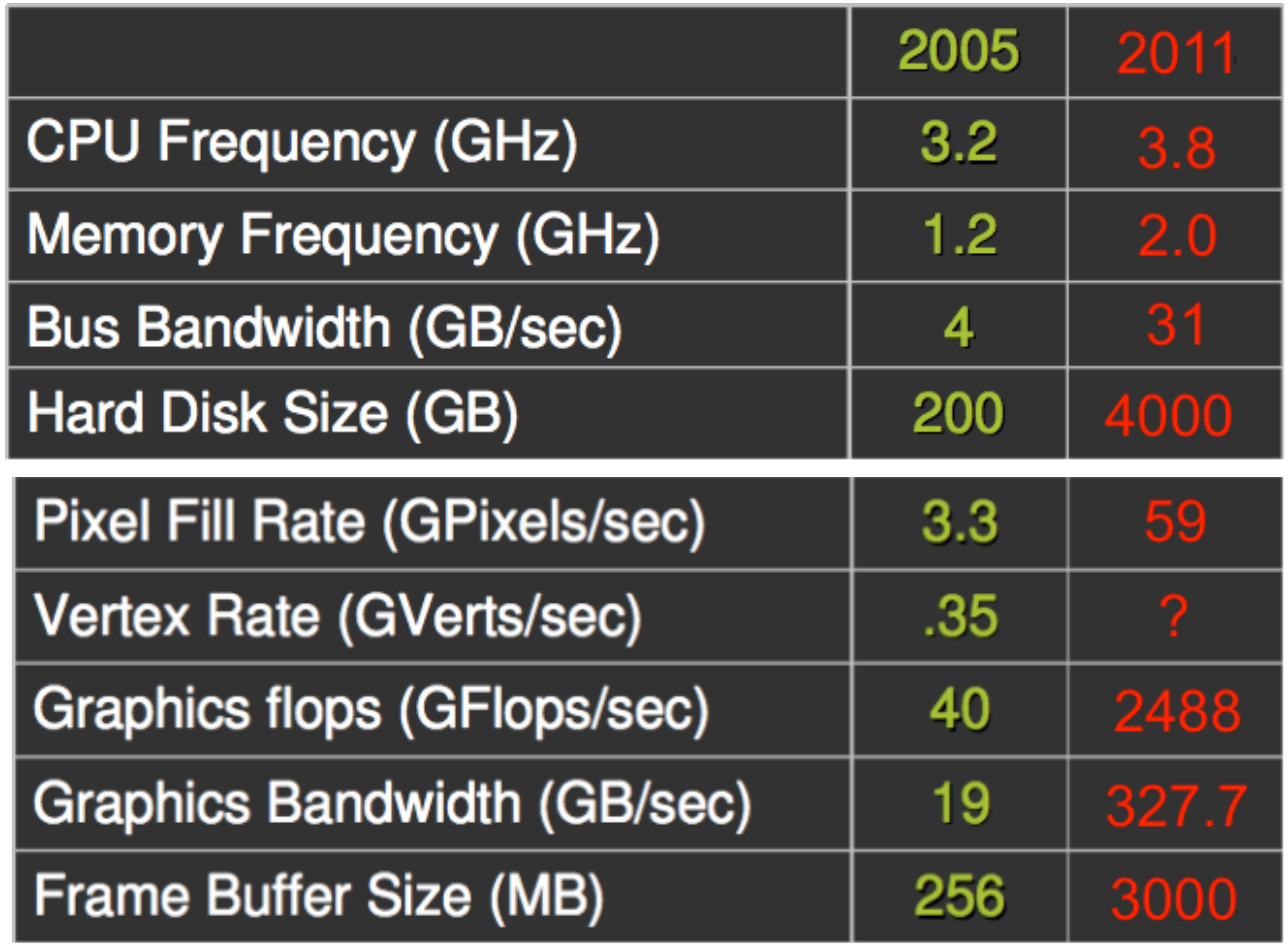


	1995	2005	
CPU Frequency (GHz)	.1	3.2	32x
Memory Frequency (GHz)	.03	1.2	40x
Bus Bandwidth (GB/sec)	.1	4	40x
Hard Disk Size (GB)	.5	200	400x
Pixel Fill Rate (GPixels/sec)	.0004	3.3	8250x
Vertex Rate (GVerts/sec)	.0005	.35	700x
Graphics flops (GFlops/sec)	.001	40	40000x
Graphics Bandwidth (GB/sec)	.3	19	63x
Frame Buffer Size (MB)	2	256	128x

How about 2005-2019?



	2005	2011		2019	
CPU Frequency (GHz)	3.2	3.8	1.18x x cores?	5.0	1.56x
Memory Frequency (GHz)	1.2	2.0	1.67x	4.27	3.56x
Bus Bandwidth (GB/sec)	4	31	7.75x	128	32x
Hard Disk Size (GB)	200	4000	20x	16000	80x



	2005	2011		2019	
CPU Frequency (GHz)	3.2	3.8	1.18x x cores?	5.0	1.56x
Memory Frequency (GHz)	1.2	2.0	1.67x	4.27	3.56x
Bus Bandwidth (GB/sec)	4	31	7.75x	128	32x
Hard Disk Size (GB)	200	4000	20x	16000	80x
Pixel Fill Rate (GPixels/sec)	3.3	59	18x	170	51x
Vertex Rate (GVerts/sec)	.35	?	?	?	?
Graphics flops (GFlops/sec)	40	2488	62x	16312*	408x
Graphics Bandwidth (GB/sec)	19	327.7	17x	672	35x
Frame Buffer Size (MB)	256	3000	12x	24000	94x

* single precision

How about 2021?

GPU (NVidia 3080Ti):

Pixel rate 186.5 GPixel/s (a bit up)
Graphics FLOP: 34 TFLOPS (double from 2019)
10240 cores!

CPU (AMD Threadripper 3990X):

256-1.5 TFLOPS
64 cores!

But is this a fair comparison? Let us compare apples with apples: GFLOPS for both!

	GPU	CPU
1995:	0.001	0.09
2005:	40	5.6
2011:	2488	91
2015:	7000	176
2016:	16380	400-700*
2021:	34000	1500

* Theoretical, 16 cores

Gets complicated here:
CUDA vs tensor cores

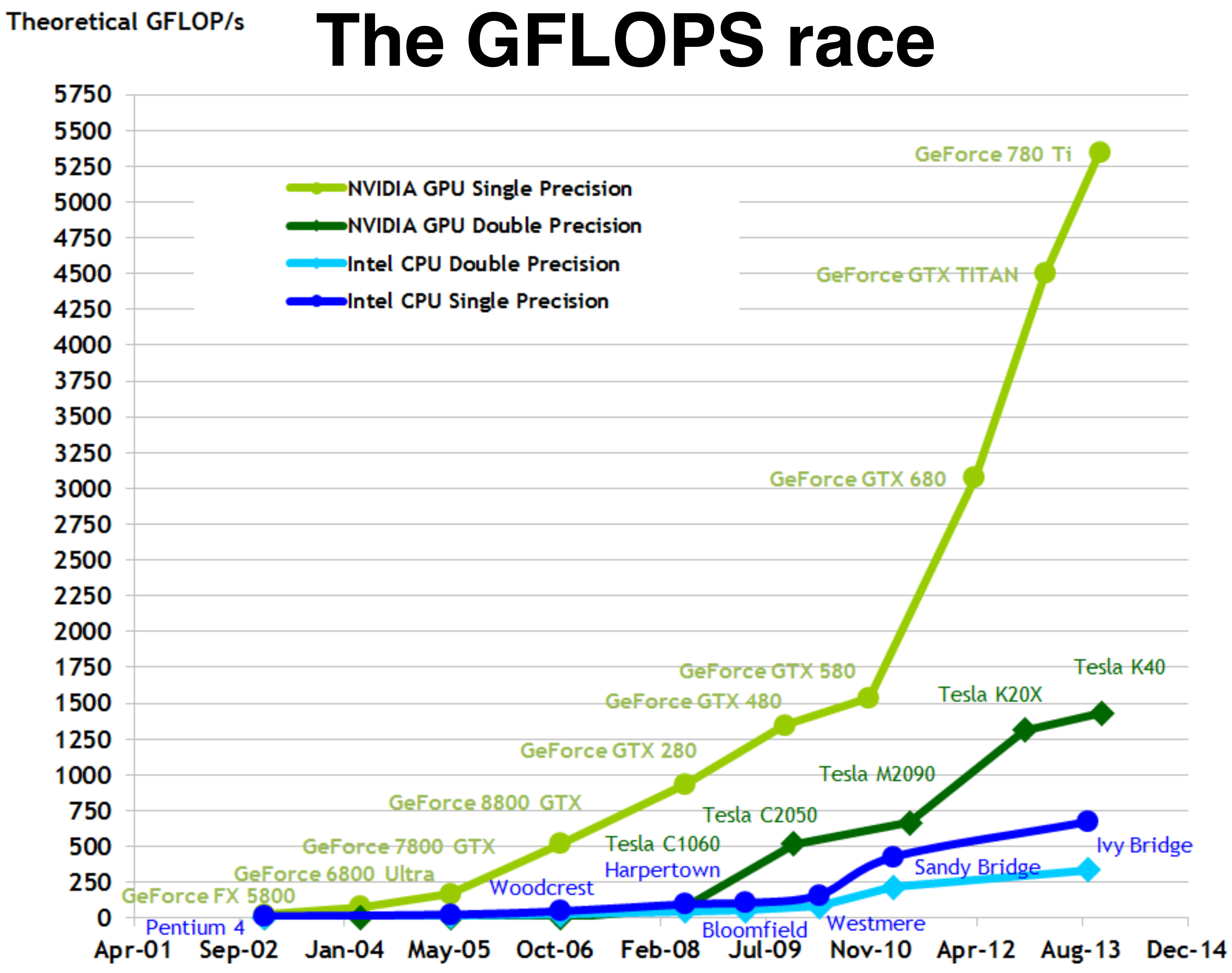
(Various sources)

How about economy: dollar per GFLOPS?

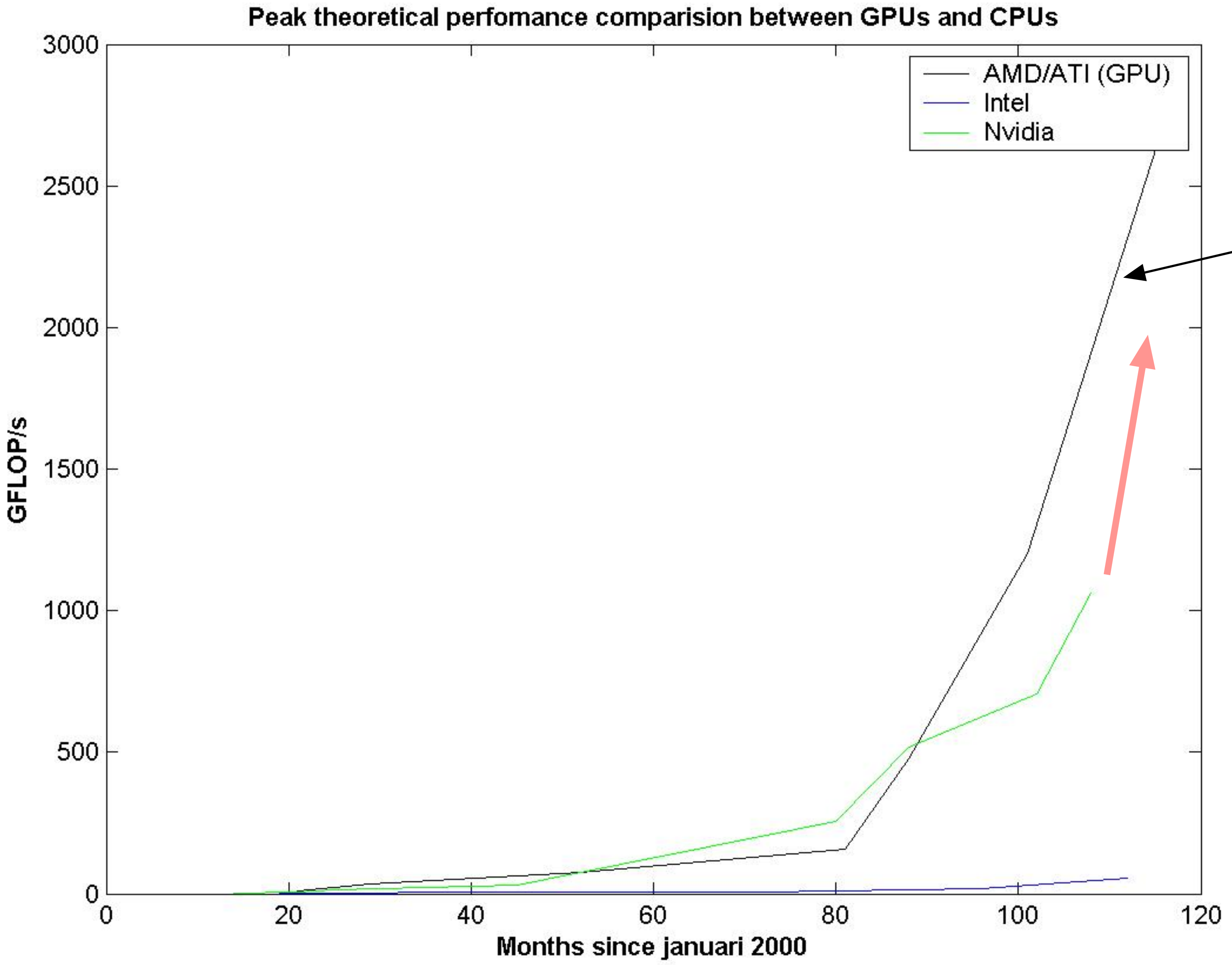
1961:	**8.3 trillion**
1984:	**42 million**
1997:	**42000 (CPU cluster)**
2000:	**836-1300**
2007:	**52**
2012:	**0.73 (AMD 7970)**
2013:	**0.22 (PS4)**
2015:	**0.08 (Radeon R9 295)**
2021:	**0.03 (probably 3080Ti?)**

(Wikipedia)

The GFLOPS race

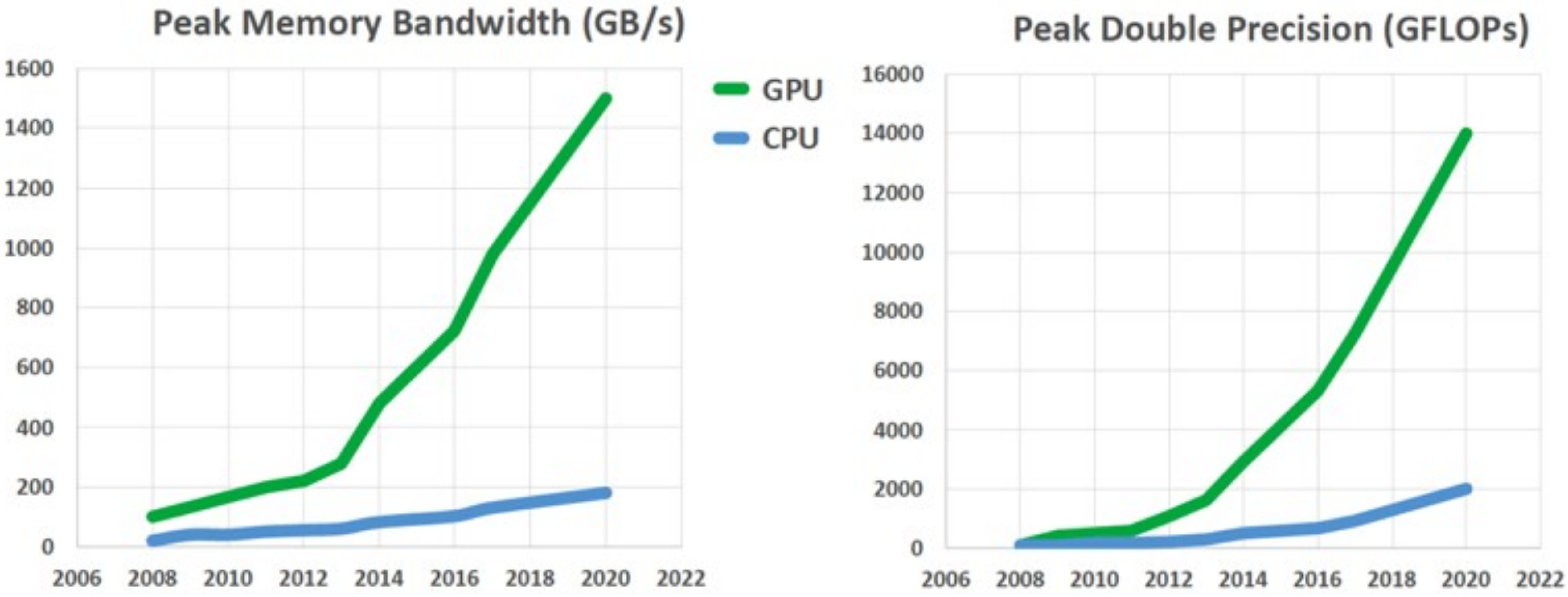


Another graph, including ATI/AMD



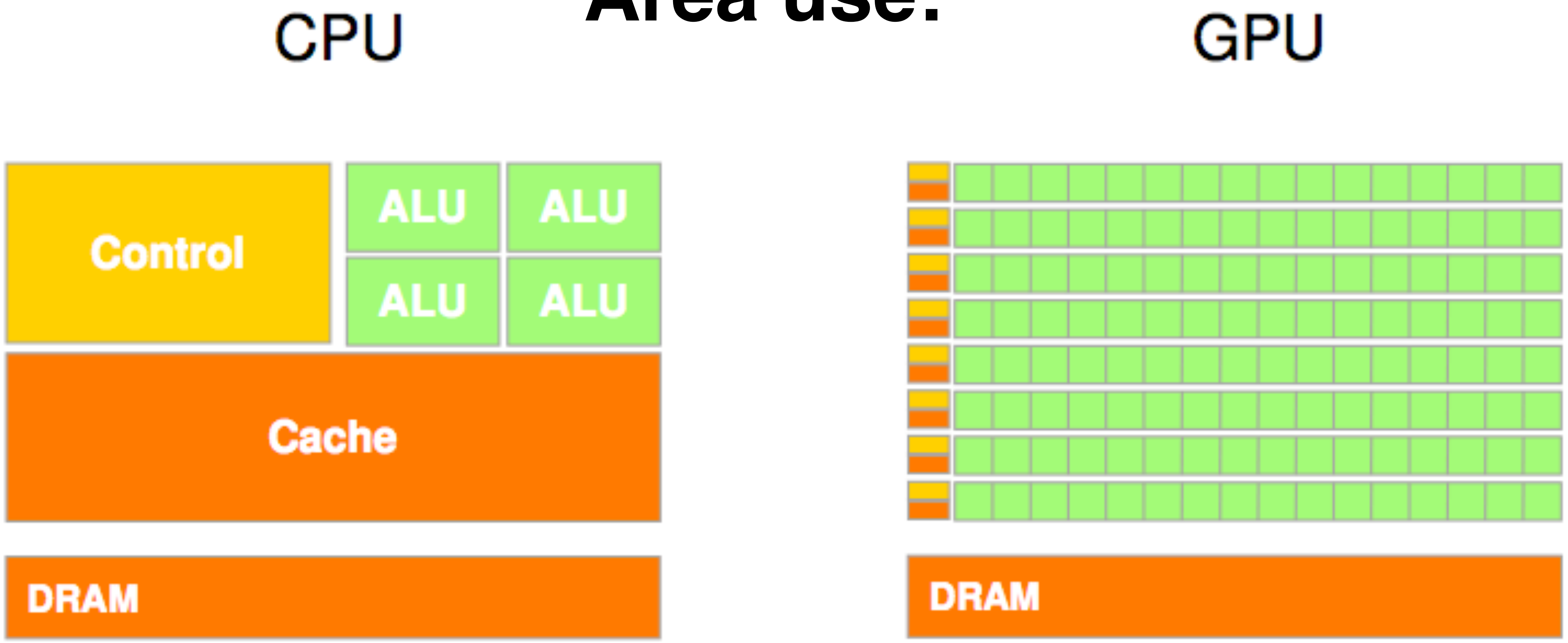
AMD took the lead in single precision while NVidia was chasing for double with Fermi

...up to today.


Peak Memory Bandwidth (GB/s)
1600
1400
1200
1000
800
600
400
200
0
2006
2008
2010
2012
2014
2016
2018
2020
2022
GPU
CPU
Peak Double Precision (GFLOPs)
16000
14000
12000
10000
8000
6000
4000
2000
0
2006
2008
2010
2012
2014
2016
2018
2020
2022

How is this possible?

Area use:



But in particular: SIMD architecture

Flynn's taxonomy

SISD Single instruction, single data Old single-core systems	MISD Multiple instruction, single data Multiple for redundance
SIMD Single instruction, multiple data GPUs, vector processors	MIMD Multiple instruction, multiple data Multi-core CPUs

SIMT, single instruction, multiple threads ≈ SIMD

SIMD

Single instruction, multiple data

Simplifies instruction handling. All cores get the same instruction.

Excellent for operations where one operation must be made on many data elements.

Is that so common? Yes!

Data best in stored arrays.

Data Oriented Programming

DOP optimizes for performance.

Data structures selected to fit the computations, instead of the programmer!

Optimize for the end user instead for the programmer!

Popular in the game industry - why not elsewhere?

SIMT - Single Instruction, Multiple Thread

A variant of SIMD.

Parallelism expressed as threads.

A programming model, but also demands that the hardware can handle threads very fast.

Threads dependent - executed in a SIMD processor!

So, why does SIMT fit a graphics processor so well?

Is this important?

- Extra hardware needed
- Different programming
- Only benefits big problems with good parallellization possibilities

but

+ Great for all image processing problems
+ Good for many other problems (sorting, FFT...)
+ Key component in the current deep learning revolution!

Deep learning

Learning systems based on very large neural networks.

Good problem for GPUs!

Remarkable results! Big trend in computer vision and other fields.

GPUs opened the door!

Why did GPUs get so much performance?

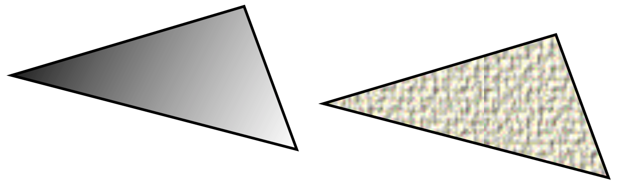
Early problem with large amounts of data. (Complex geometry, millions of output pixels.)

Graphics pipeline designed for parallelism!

Hiding memory latency by parallelism

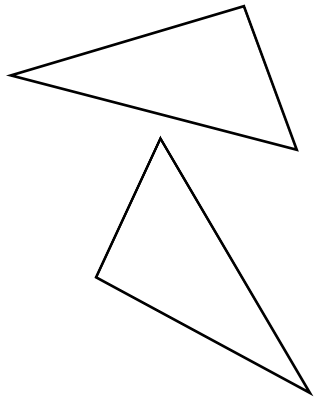
Volume. 3D graphics boards central component in game industry. Everybody wants one!

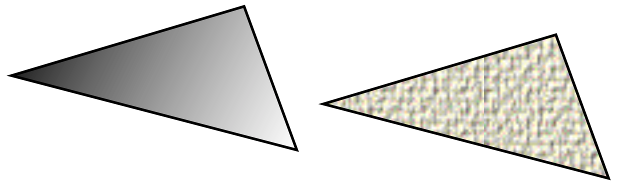
New games need new impressive features. Many important advancements started as game features.



Must process many pixels fast!

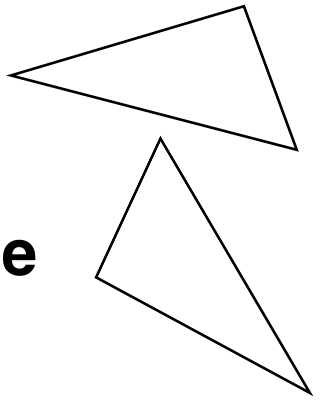
Early GPUs could draw textured, shaded triangles much faster than the CPU.





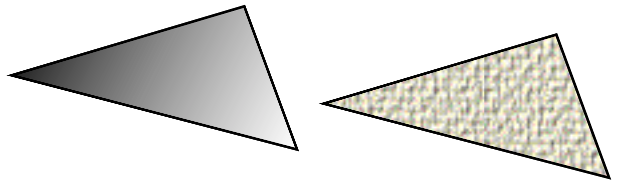
Must process many pixels fast!

Early GPUs could draw textured, shaded triangles much faster than the CPU.



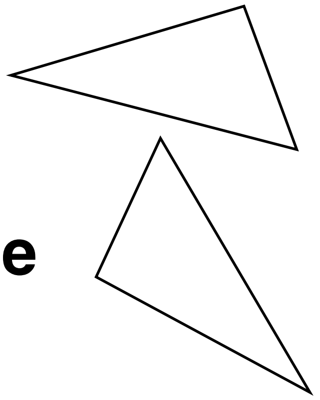
Must do matrix multiplication and divisions fast.

Next generation could transform vertices and normalize vectors.



Must process many pixels fast!

Early GPUs could draw textured, shaded triangles much faster than the CPU.

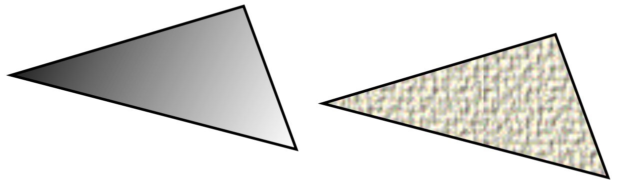


Must do matrix multiplication and divisions fast.

Next generation could transform vertices and normalize vectors.

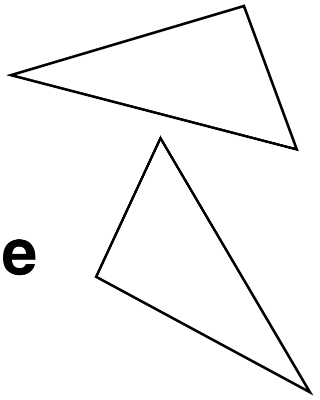
Must have programmable parts.

This was added to make Phong shading and bump mapping.



Must process many pixels fast!

Early GPUs could draw textured, shaded triangles much faster than the CPU.



Must do matrix multiplication and divisions fast.

Next generation could transform vertices and normalize vectors.

Must have programmable parts.

This was added to make Phong shading and bump mapping.

Must work in floating-point!

This was for light effects, HDR.

So a GPU should

- process vertices, many in parallel, applying the same transformations on each

- process pixels (fragments) in parallel, applying the same color/light/texture calculations on each

SIMD friendly problem!

Less control, control many calculations instead of one

A different kind of threads

SIMD threads, all run the same program

Group-wise, they execute in parallel, SIMD-style

Made for graphics operations: Shader threads calculate one pixel or one vertex

CUDA/OpenCL threads may calculate anything, but typically one part of the output - in order

A look at the GPU architecture

Back to the timeline, big changes:

Pre-G80: Separate vertex and fragment processors.

Hard-wired for graphics. Load balance problems.

G80: Unified architecture. More suited for GPGPU. Higher performance due to better load balancing.

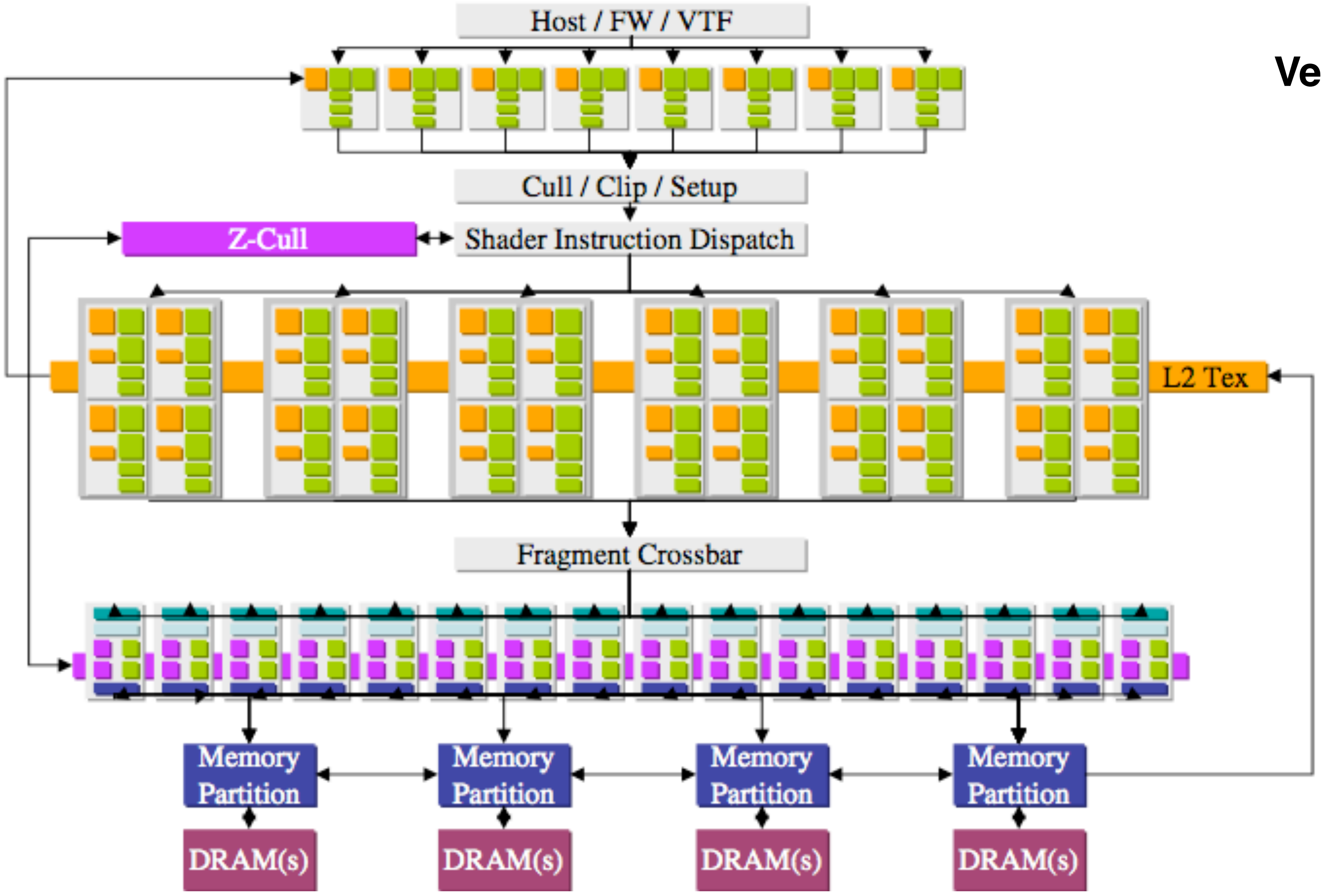
G92: Similar to G80, more cores, more cores per group.

GT100: Much more double precision

TU102: Tensor & RT cores

(Similar track for AMD)

7800: High-end GPU before G80

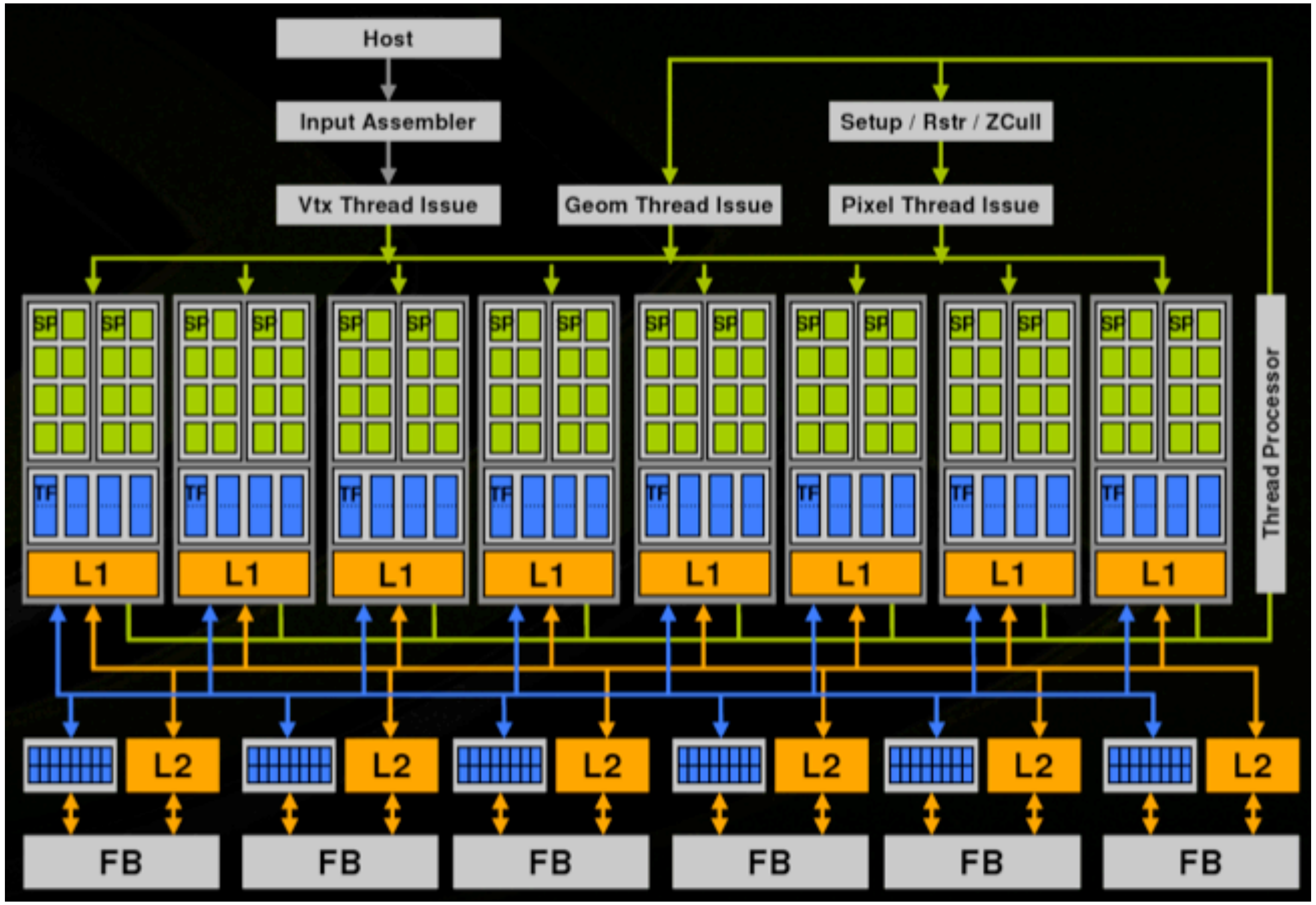


Vertex processors

Fragment processors

Framebuffer operations

G80



Hardware formerly between vertex and fragment processors

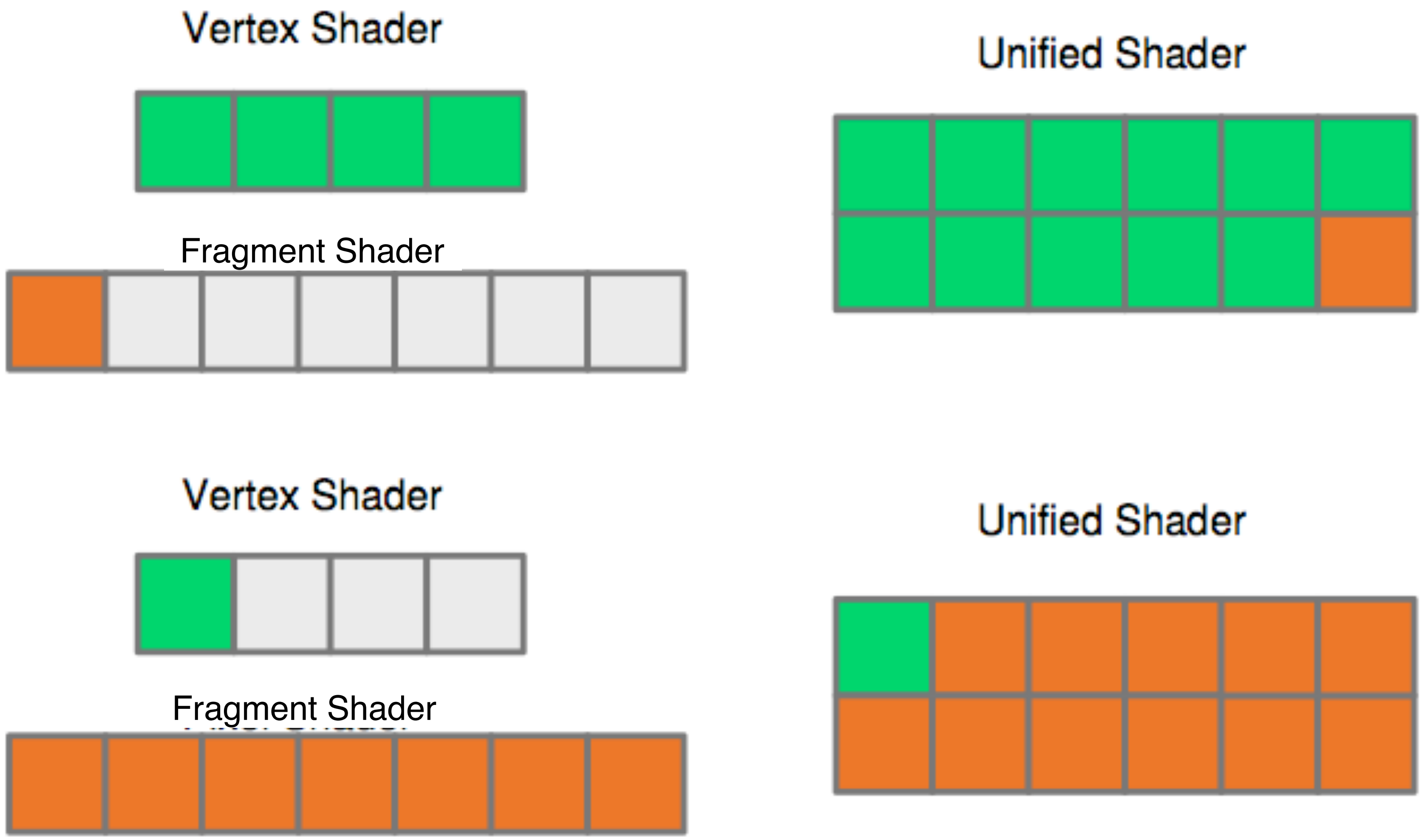
Unified processors!

Framebuffer operations

G80: A question of *load balance*!

Separate vertex and fragment processors

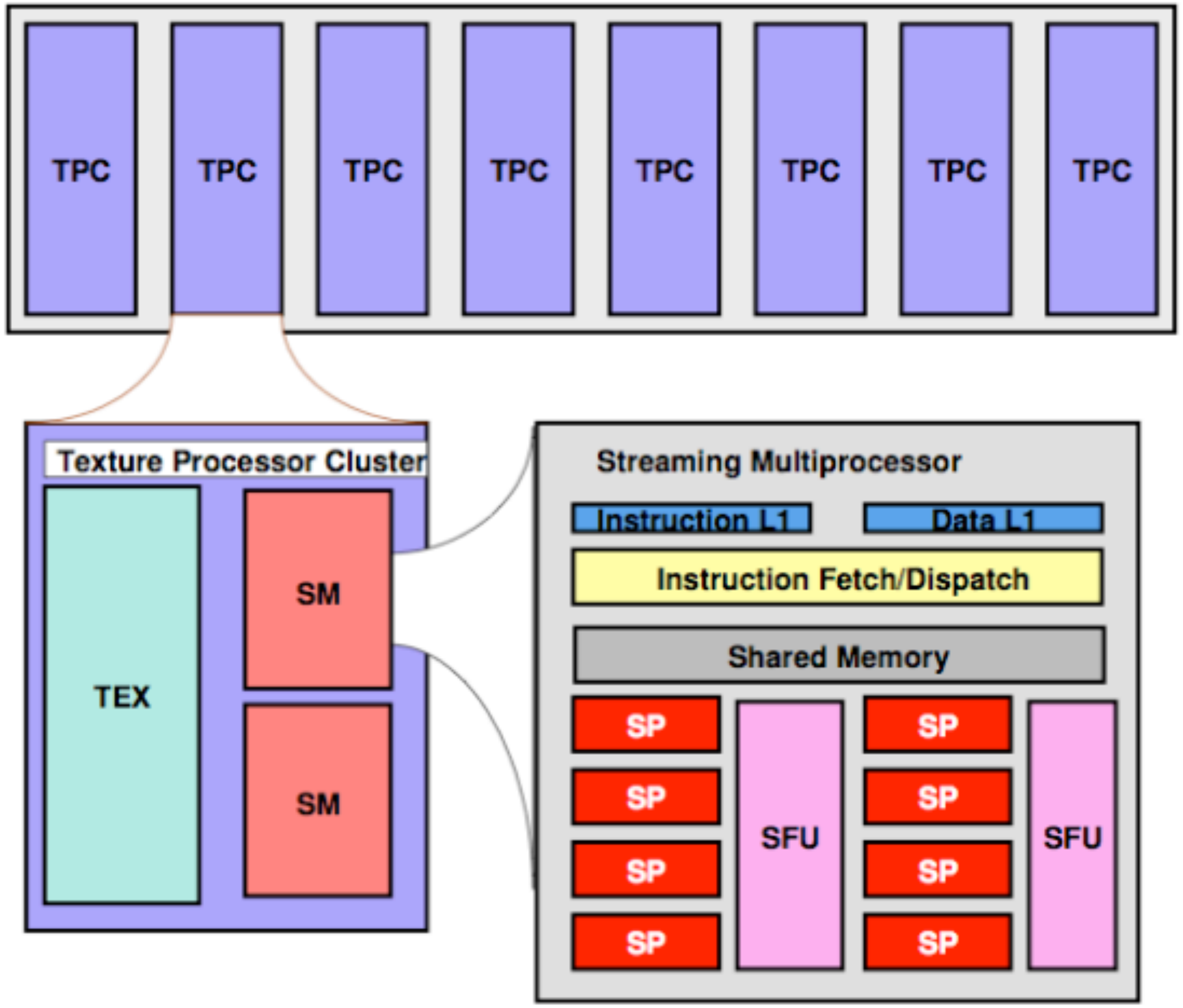
Unified processors



Vertex problem (e.g. complex geometry)

Fragment problem (e.g. advanced rendering)

G80 processor hierarchy

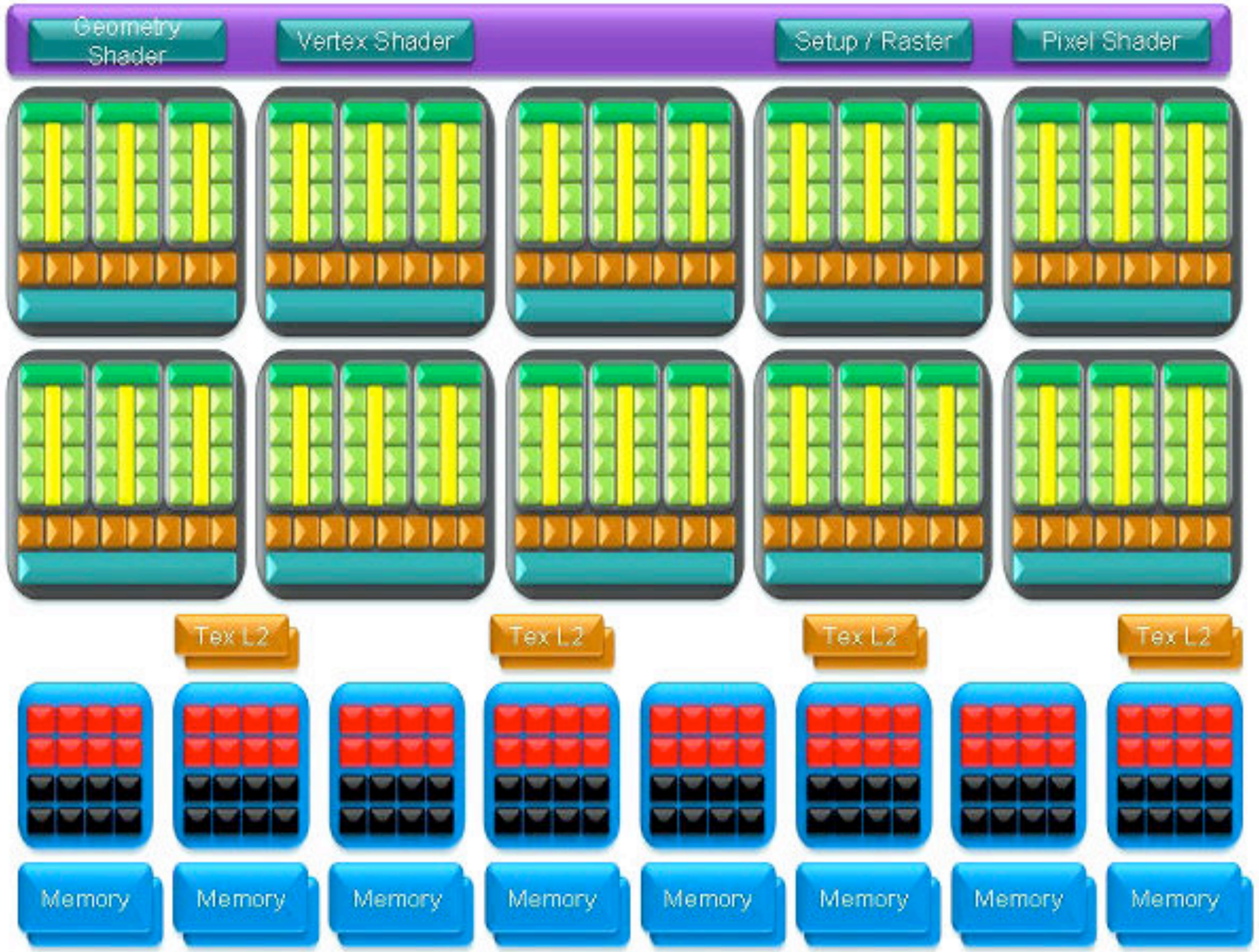


8 top-level groups of TPCs

SM = Streaming Multiprocessor

SM is a group of 8 SIMD cores

GT200

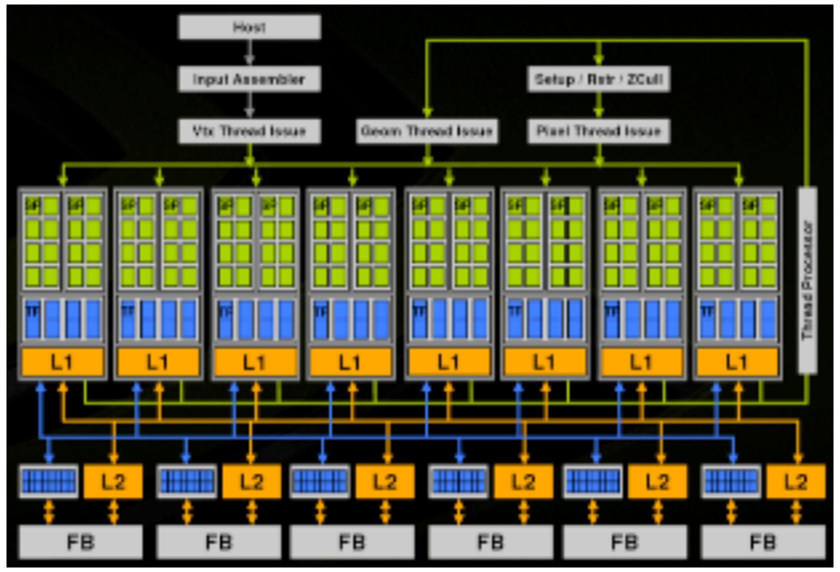


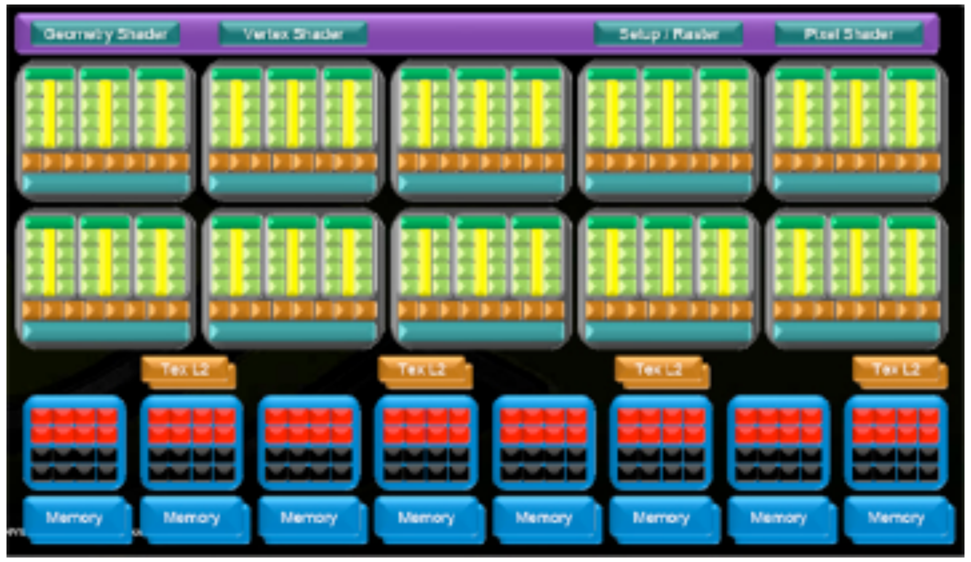
Many updates are just this:

Similar but with a bit more of everything

G80 vs GT200 in numbers:

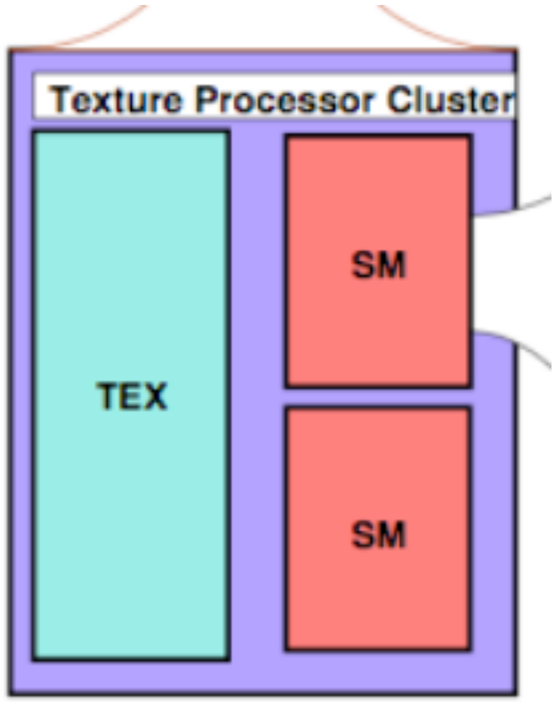
8 cores per SM 10 cores per SM
2 SMs per cluster 3 SMs per cluster
8 clusters 10 clusters





8 was *not* a magic number - more cores per SM

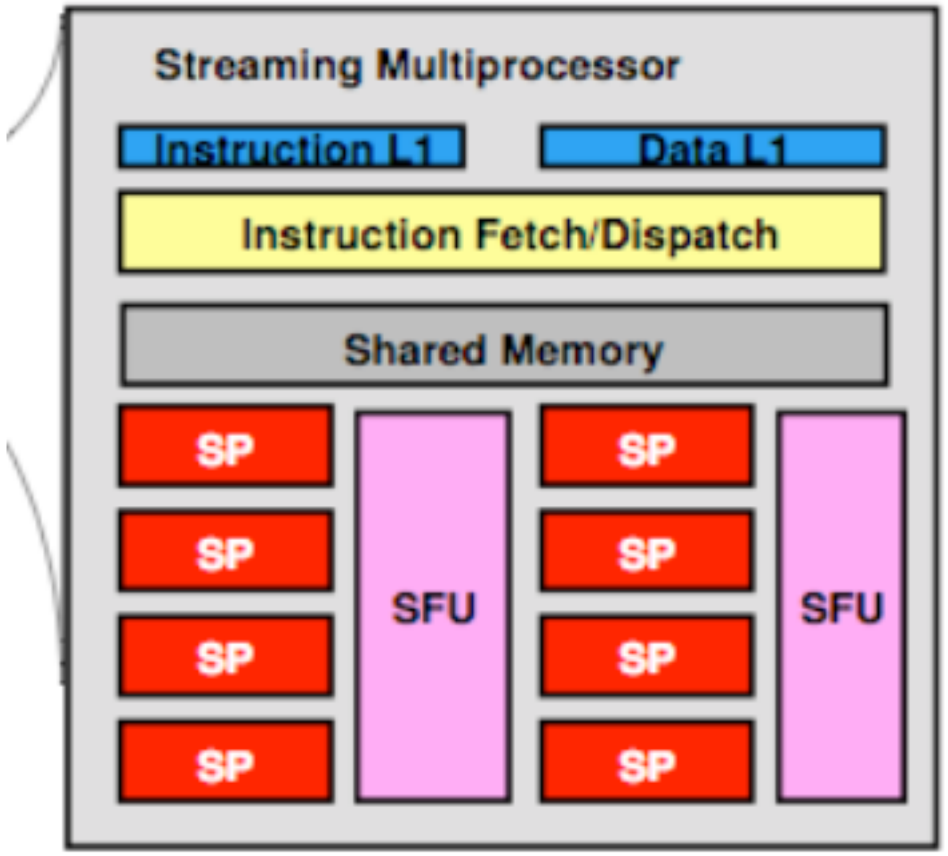
Vital components



Texture processor cluster: 2 or 3 SMs and a *texturing unit*

A texturing unit will provide texturing access with automatic interpolation - vital component for graphics

Vital components



SM: 8 cores

but also

SFU: Special functions unit

Shared memory

Register memory in each core

Instruction handling/thread management

How much architecture details do we need to know?

Shaders: The architecture is mostly invisible

Cuda/OpenCL: Less so, but number of cores more or less ignored - as long as we provide more parallelism in our algorithm than the architecture has!

Memory usage is specified by the programming languages. More about that later.

2010: Fermi (GT100)

16 SMs

32 cores per SM

Important change:

Much area for L2 cache!

More on Fermi

4x performance for double (64-bit FP)

More silicon space for cache! More like a CPU.

CGPU = Computing Graphics Processing Unit

=> NVidia aims for GPGPU with Fermi!

2012: Kepler (GK104, GK110)
2014: Maxwell (GM107, GM204)

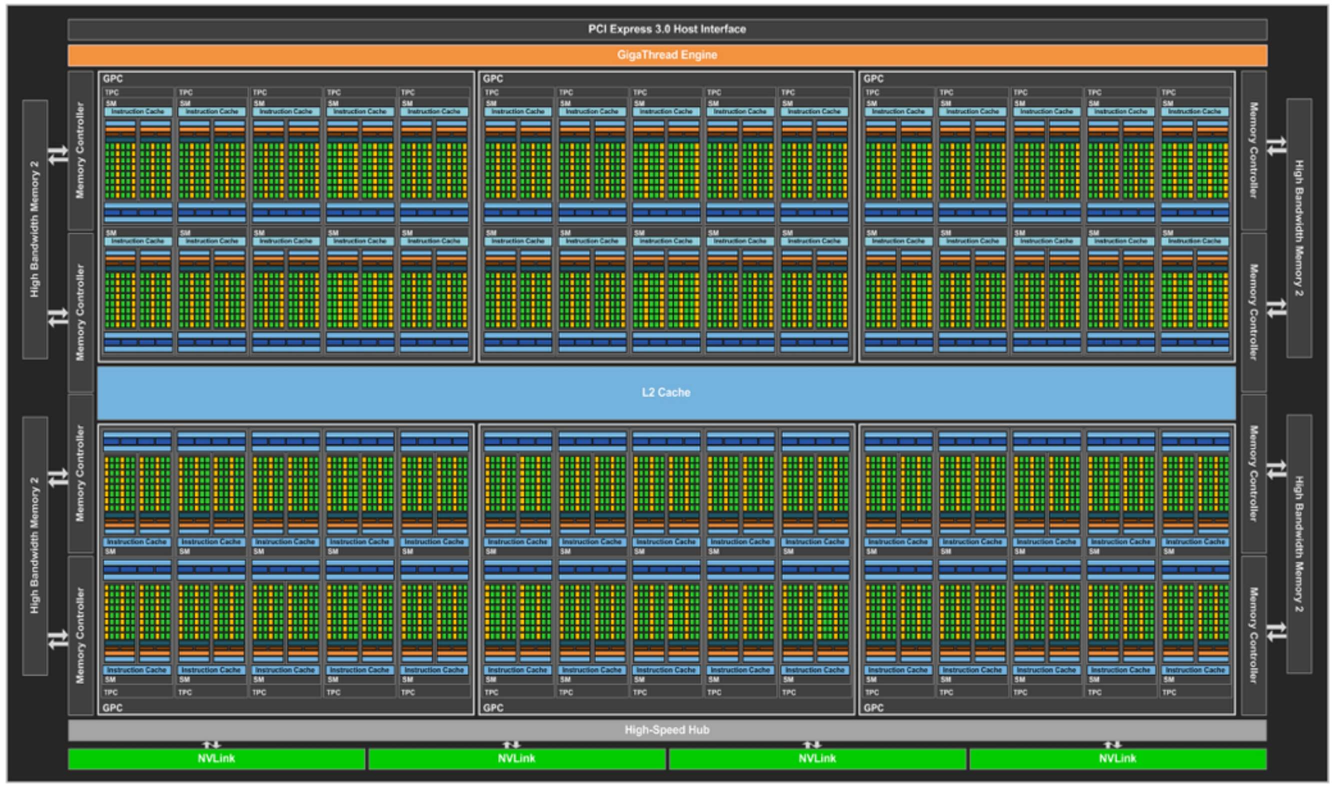
Back to graphics focus, strikes back against AMD.

Fewer SMs, double performance lagging behind.

AMD taking the lead in GPU computing with the R9 series!

2016: Pascal (GP102-107)

Good double performance is back!



2018: Turing
2020: Ampère

Big change towards specialized parts

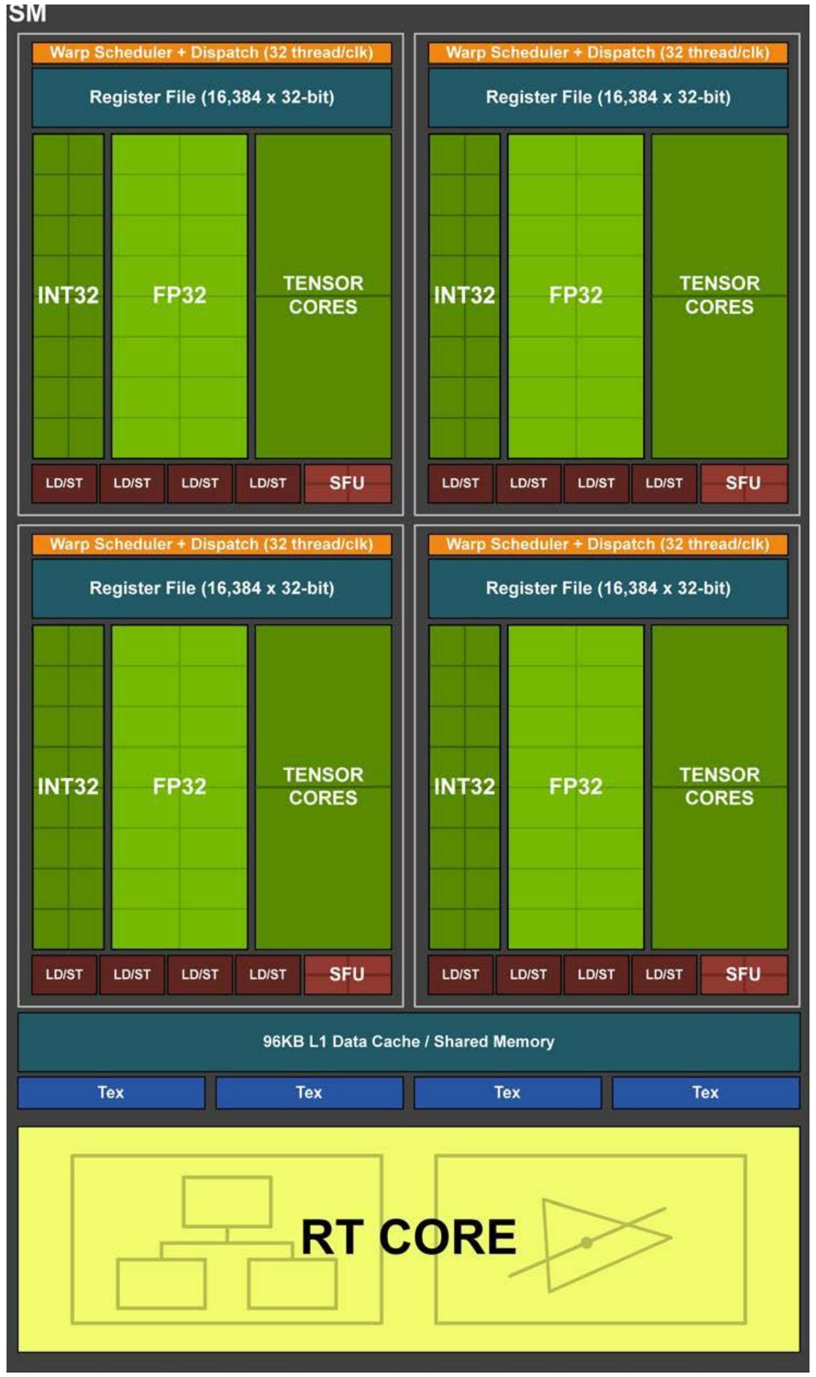
- **Tensor cores**
- **RT cores**
- **Focus on raytracing and learning**

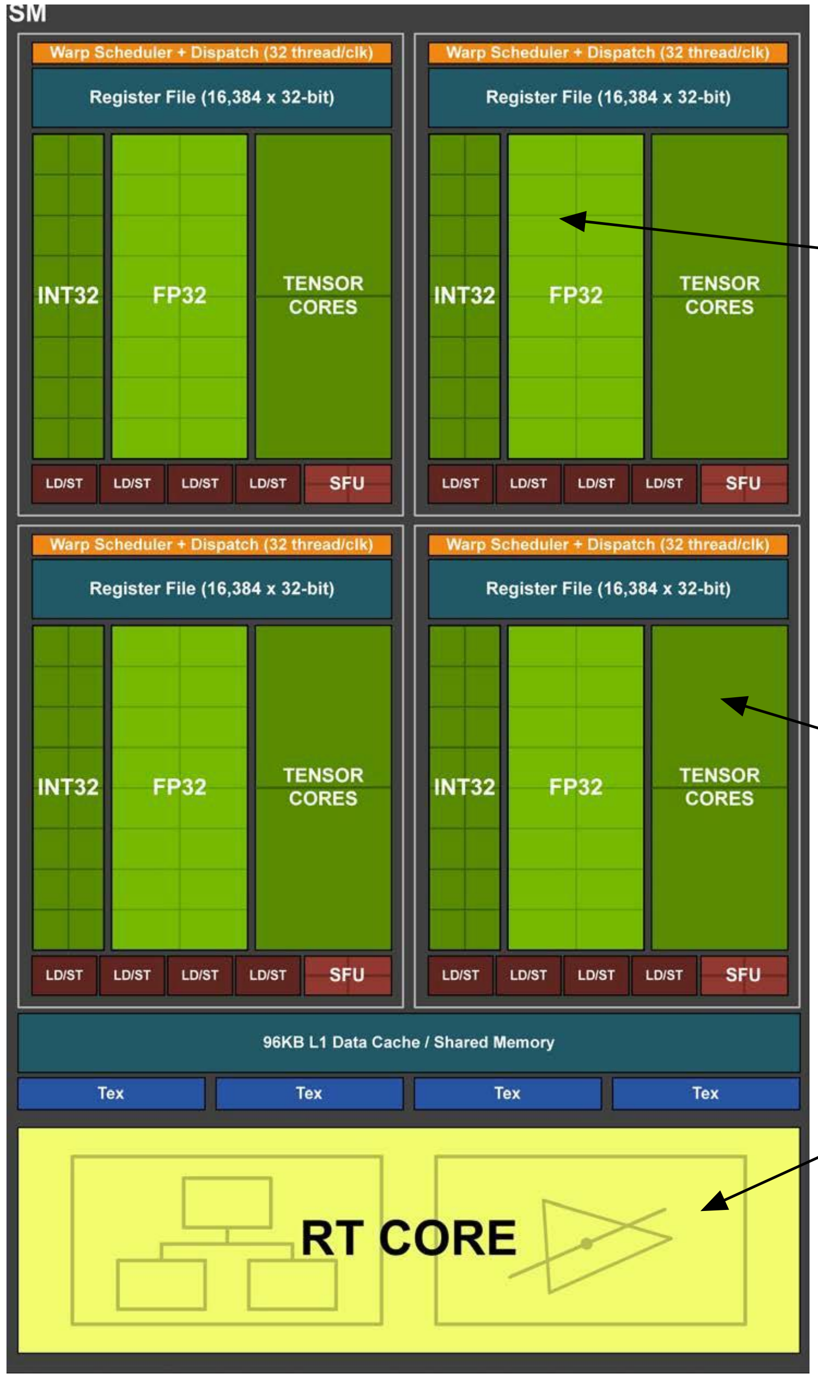
Turing vs G80

G80 = unification, only one kind of cores = better use of hardware

Turing = separation, three kinds of cores... meaning what?

Contradiction! Will this last?





General purpose hardware

Our focus!

Special purpose hardware

Questionable usability for general purpose computations

Related parallelization efforts

IBM Cell (next generation canceled!)

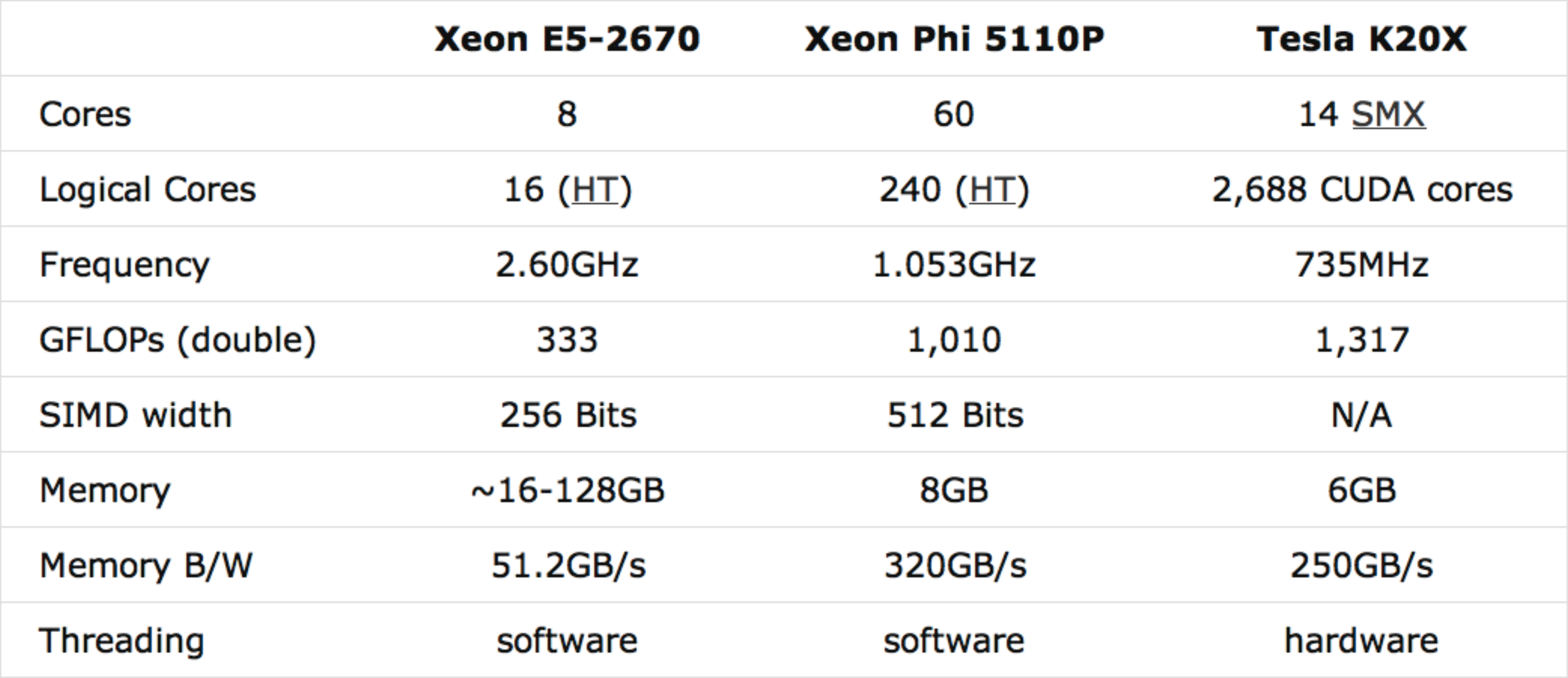
Intel Larabee (”put on ice” - dead)

GPUs are the clear winners so far!

But never count out Intel...

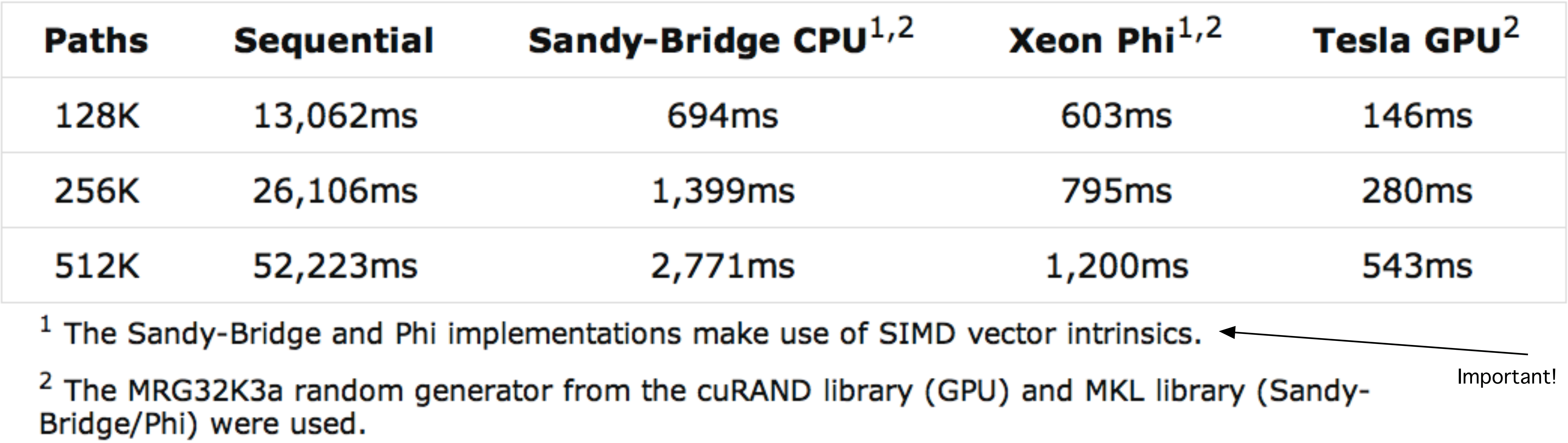
how about the more recent Xeon Phi?
(Follow-up on Larabee)

How does it compare?



	Xeon E5-2670	**Xeon Phi 5110P**	**Tesla K20X**
Cores	8	60	14 SMX
Logical Cores	16 (HT)	240 (HT)	2,688 CUDA cores
Frequency	2.60GHz	1.053GHz	735MHz
GFLOPs (double)	333	1,010	1,317
SIMD width	256 Bits	512 Bits	N/A
Memory	~16-128GB	8GB	6GB
Memory B/W	51.2GB/s	320GB/s	250GB/s
Threading	software	software	hardware

Test: Does it compete?



Paths	Sequential	Sandy-Bridge CPU[1,2]	Xeon Phi[1,2]	Tesla GPU[2]
128K	13,062ms	694ms	603ms	146ms
256K	26,106ms	1,399ms	795ms	280ms
512K	52,223ms	2,771ms	1,200ms	543ms

[1] The Sandy-Bridge and Phi implementations make use of SIMD vector intrinsics.

[2] The MRG32K3a random generator from the cuRAND library (GPU) and MKL library (Sandy-Bridge/Phi) were used.

Important!

The GPU still wins! (Even over other SIMD!)

Conclusion comparison
SB - Xeon Phi - GPU

Even the CPU performed pretty well.

All use SIMD (at least partially) for best performance!

All require you to code in parallel!

And this brought us to:

GPGPU/GPU Computing

General Purpose computation on Graphics Processing Units

Mark Harris, 2002

Perform demanding calculations on the GPU instead of the CPU!

At first, appeared to be a wild idea, but is now a very serious technology! Results were highly varied in the early years, but the GPU advantage has grown bigger and bigger.

Key components starting the GPGPU trend

High processing power in parallel

Programmability: Introduction of shader programs, much more flexible, programmable for any problem.

Floating-point buffers: Vital! Initially with poor precision. 32-bit floating-point decent... but not really impressive.

GPGPU approaches

- Using fixed pipeline graphics
- Shader programs
- CUDA
- OpenCL
- Compute shaders

Fixed pipeline GPGPU

Reformulate a problem to something that can be done by standard graphics operations.

Limited success 1999/2000. Not of any practical interest!

Example: Jörgen Ahlberg, face tracking

Fragment (pixel) shader based GPGPU

Portable! All GPUs can use shaders, no need for extra software, run using standard software/drivers.

All modern shader languages (GLSL, Cg, HLSL) are similar and easy to program in.

Requires a re-mapping of data to textures.

Very good results already in 2005: 8x speedups overall reported!

CUDA-based GPGPU

Only works on NVidia hardware.

Requires extra software - which isn't very elegant.

Nice integration of CPU and GPU code in the same program.

Excellent results! 100x speedups are common - before optimizing! Even low-end GPUs give significant boosts.

OpenCL-based GPGPU

Works on various hardware - not only GPUs.

Developed by Khronos Group, pushed by Apple.

Harder to get started, software looks pretty much like programming shaders.

OpenGL Compute shaders

Built into OpenGL

Similar to OpenCL

Good portability

Direct Compute Compute shaders

Built into DirectX

Similar to OpenCL

MS only

Vulkan

The "new OpenGL", arrived 2016.

"Bleeding edge".

Future main generic GPU platform for both graphics and computing?

Same compute shaders as OpenGL.

Metal

Apples "Vulkan".

Apple has deprecated everything else - including OpenCL

"Metal Performance Shaders".

Apple only.

Use the source, Luke!

Four trivial examples:

Hello World! for CUDA

Hello World! for OpenCL

Hello World for GLSL

Hello World for Compute Shaders

In Olympen

GTX1080

Pascal GPUs!

In Asgård

GTX2060

Turing GPUs!

Pretty fresh and good performance.

That’s all, folks!

Next time: Introduction to CUDA