Introduction to SASS & GPU Microarchitecture

(aka advanced optimisation for when those pesky regulators don’t approve your permit for a Dyson Sphere to power your GPUs)

Arun Demeure, 23/11/2024

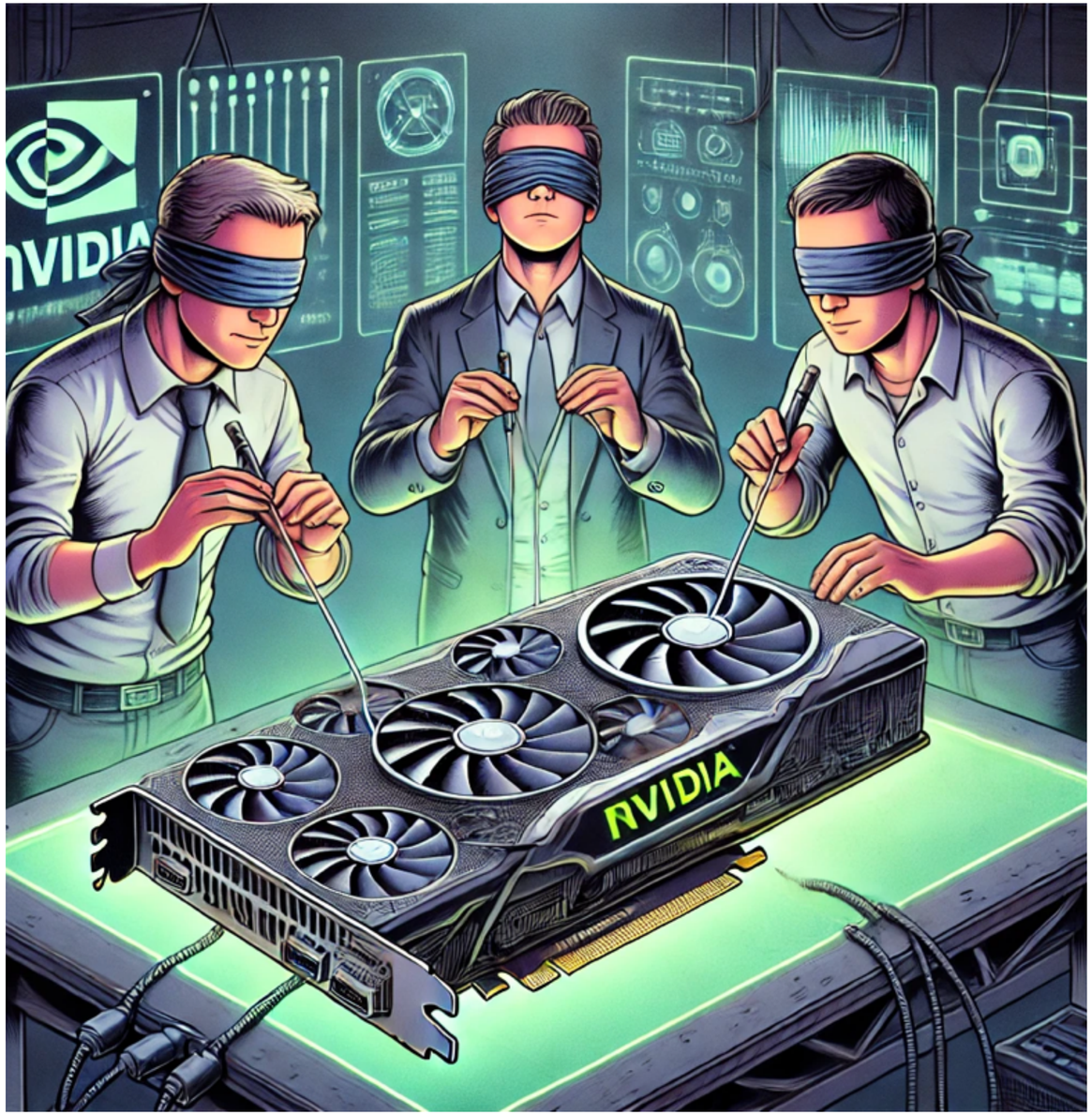
The Blind Men & The GPU

How successful does a chip need to be for someone to make an open source assembler?

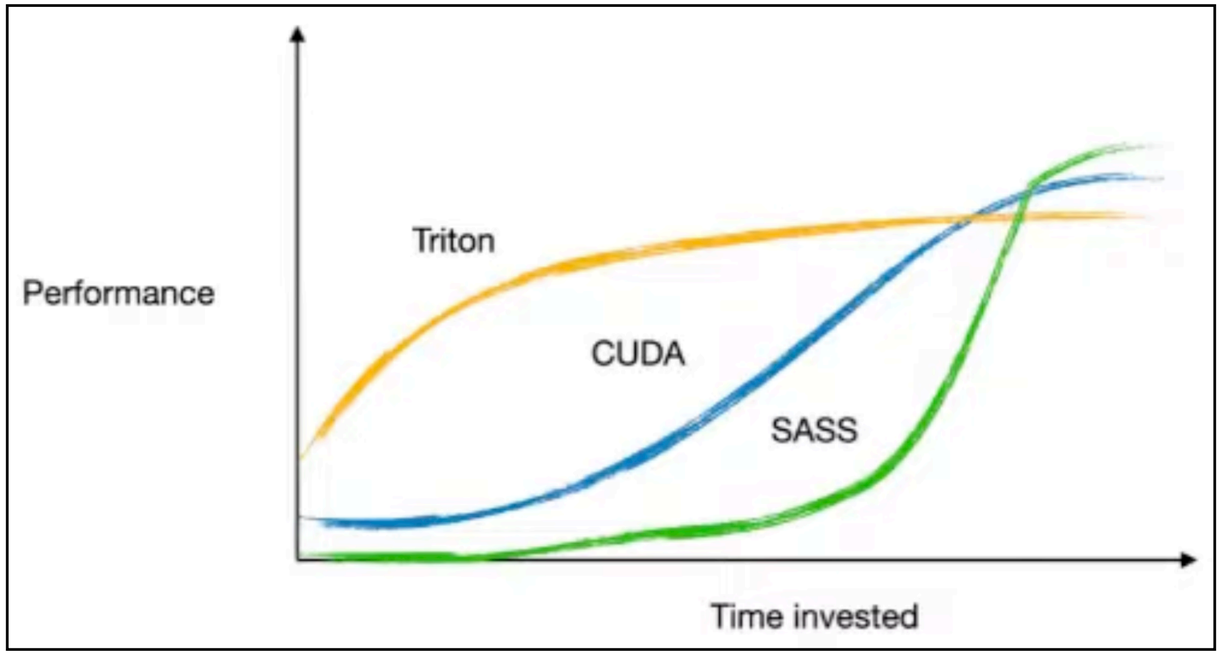
Strictly less than $100B in revenue, apparently! They exist for most GPUs but not H100 SASS.

What's the point of studying GPU assembly if you can't change it?

What's the point of understanding your DNA if you can't change it?



Great compilers beat average programmers…

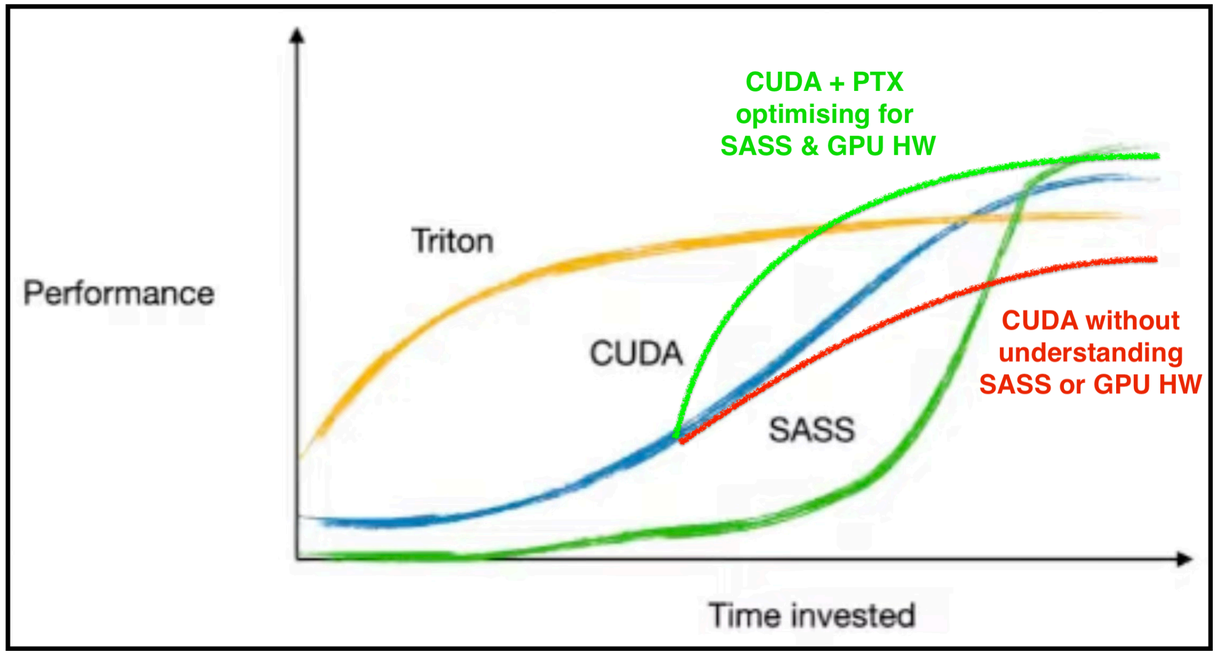


Block-based GPU Programming with Triton (Philippe Tillet, OpenAI)

https://www.youtube.com/watch?v=v_q2JTIqE20

Great compilers beat average programmers…

… but great programmers bend compilers to their will.

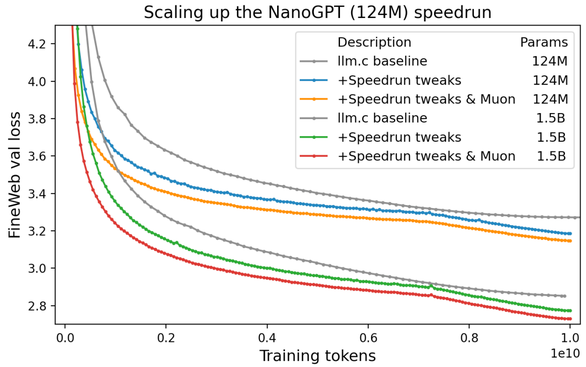


You don’t need to write assembly!

You “just” need to write CUDA (/PTX/CUTLASS/MLIR/Triton) that generates efficient SASS.

Iteration Time > Performance

In research, performance is really just one factor affecting the researcher's iteration time.
In production, performance is really just one factor affecting the end user's iteration time.



- modded-nanogpt vs llm.c
  - It would have taken longer to experiment with these ideas in llm.c
  - Probably still room for improvement with low-level optimisation!
  - And ironically torch.compile takes longer than the run itself (?!)
  - Todo: profile modded-nanogpt on H100/H200 :)

- Throughput matters *because* it impacts latency!
  - ~10% performance on ~2 months pre-training would save ~1 week of iteration time
  - You could always just wait *even longer* for your runs to complete (you won't, but you could)
  - PyTorch/Triton/etc. are very productive frameworks and likely to remain dominant for that reason

- Some low-level optimisations are simple and low maintenance, you just *need to know they're possible.*
  - But you still need to optimise algorithms & Big-O complexity first (FlashAttention for example!)
  - Key for compiler engineers (makes everyone's job easier, except mine trying to beat PyTorch)
  - Understanding HW helps you make sense of Nsight Compute profiling of Triton/PyTorch too

- **Potential for max optimisation for the largest training runs & hyperscaler inference?**
  - **O(N) is the wrong metric for hyperscalers these days…**

O(NR)

Optimisation (per Nuclear Reactor)

- Assuming ~50MW per SMR
  - Kairos is ~75MW per reactor (Google buying 6 or 7 by 2035)
  - Oklo Aurora is 15MW to 50MW

- 100K H100 Cluster = 70MW for GPUs alone (e.g. xAI Colossus p1)
  - 1% power reduction = 0.7MW = **~1.4% of a nuclear reactor**

- 100K GB200s: NVL72 = 132kW —> 183MW just for compute!
  - +1% performance at same power = **~2% of a nuclear reactor**

- But wait, how many GPUs is NVIDIA selling again?! (CEO Math style!)
  - ~$37.5B/quarter today, assume $200B+ over Blackwell lifetime
  - ~$3M per NVL72 —> 67K+ NVL72 / 5M+ GB200 equivalent

More than 10 gigawatts of power worldwide (conservatively?!)

<u>1% Perf/W = 2 Small Nuclear Reactors</u>

Advanced optimisation has a high maintenance cost, but so do nuclear reactors.

But more importantly… it's fun!

and builds intuition so you'll hopefully spend *less* time in the profiler

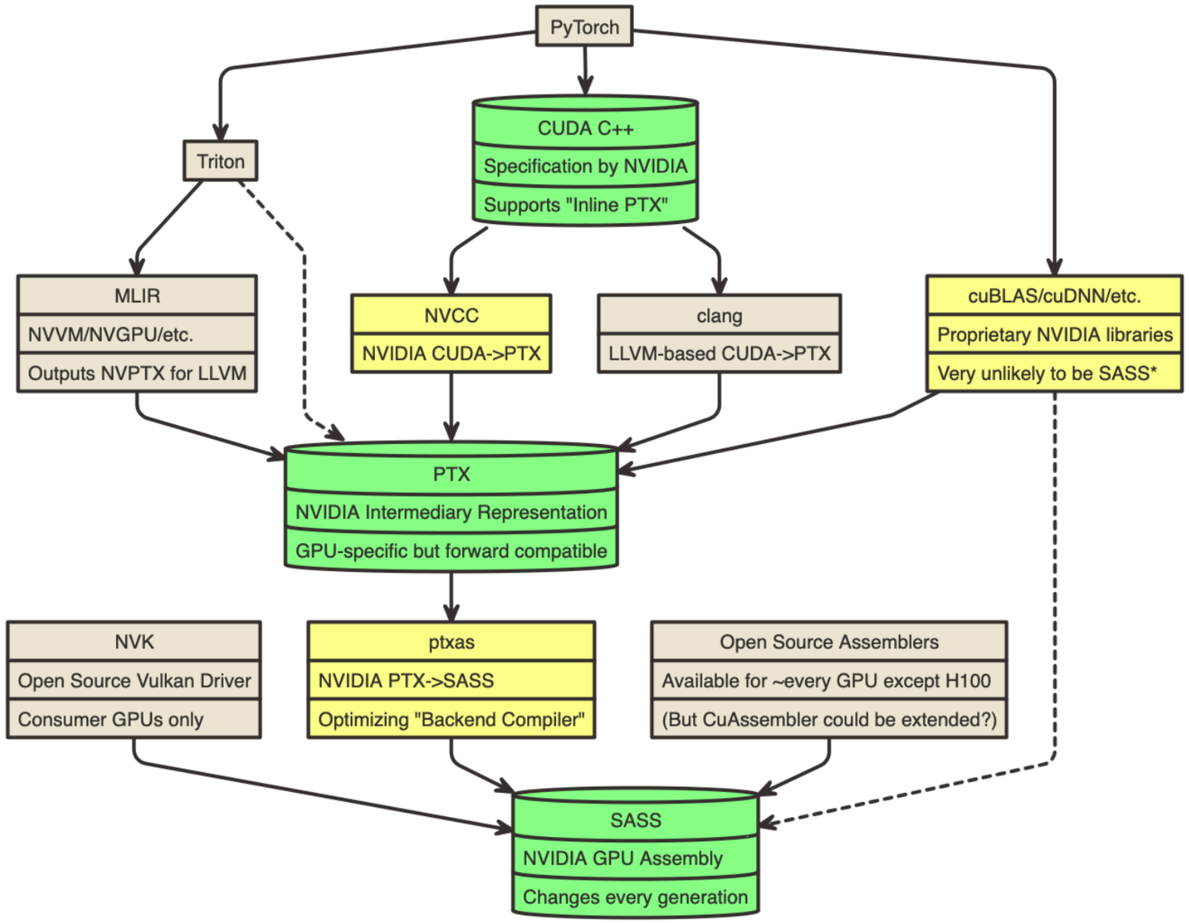
- Advanced Optimisation may result in large dopamine hit and risk of addiction.
  - Symptoms may include:
    - Premature optimisation
    - Losing all sense of time
    - **Becoming a GPU Ninja**

- But may also result in unintentional dopamine detox…
  - Quickest way to write fast code is to start with very fast code :(
  - Many optimisations fail or take longer than expected, that's okay! (hopefully)

- This won't be able to cover very much, but hopefully enough to get you started!
  - Tensor Cores and TMA/DSMEM/etc. are their own completely separate topics
  - Appendix & slides includes references a number of presentations & papers
  - But in the end, the only way to really learn many of this is to do it yourself!

Generating SASS

One black box at a time



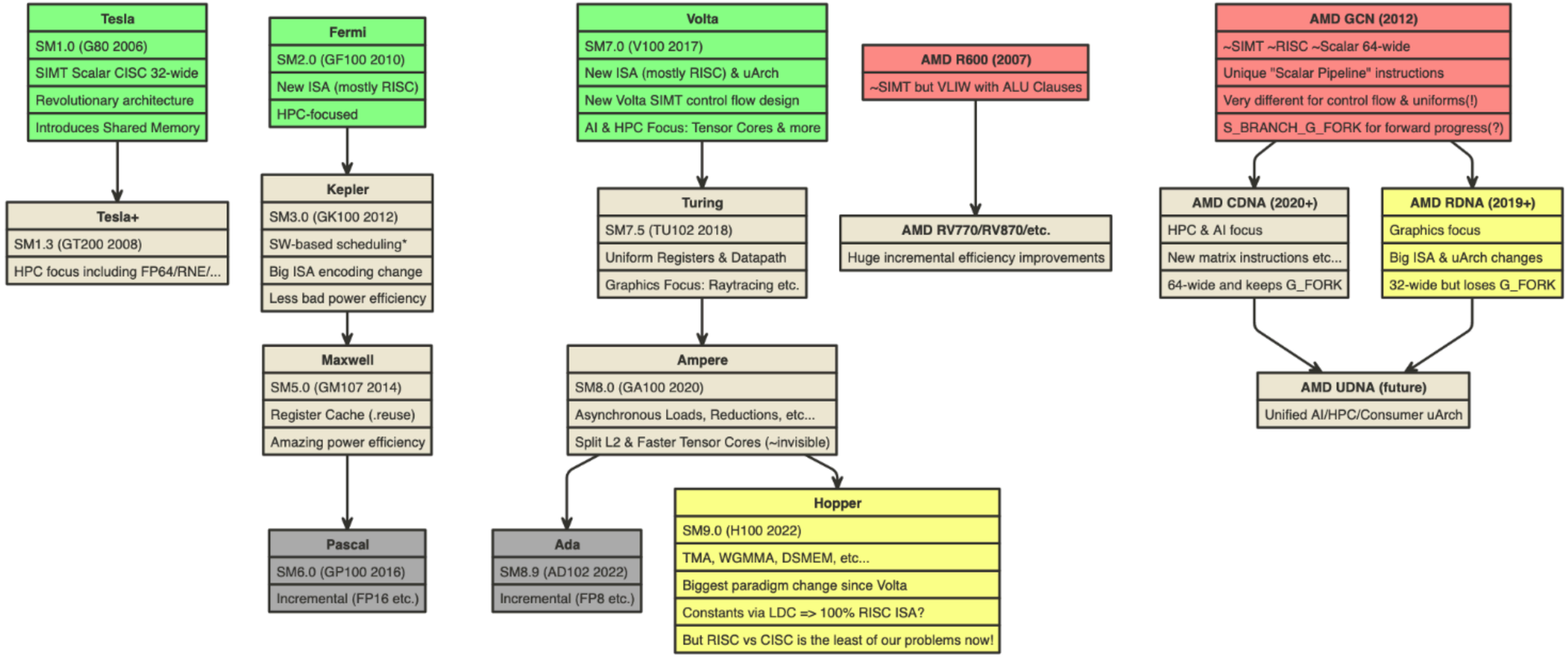
List of assemblers & related tools

You will probably never use any of them, but it's cool they exist!

- CuAssembler: Most up to date assembler that supports everything from Pascal to Ampere, extremely flexible in theory but doesn't fully work out of the box for Hopper/H100 because of desc[…] and other new functionality (not actively maintained).
- maxas by Scott Gray: Most widely referenced & complete assembler (Maxwell/Pascal). Written in Perl by Scott Gray to beat NVIDIA's own SGEMM kernels (SGEMM optimisation walkthrough has lots of good info but careful it doesn't all apply to Volta). He later introduced efficient Winograd kernels for convolutions before moving to OpenAI. It seems his kernels were better for some sizes, but not across the board, which highlights the difficulty of handwriting SASS.
- cudasm/decuda for G80: 1st SASS assembler (and 1st disassembler before cuobjdump was released by NVIDIA!)
- Other assemblers on GitHub: TuringAs, KeplerAs, AsFermi, etc…
- Other interesting tools that aren't assemblers:
  - nv_isa_solver: Parser that documents instruction encoding in-depth for H100/Ada (discussed more later!)
  - DocumentSASS: Clever "man in the middle" tool that intercepts strings from nvdisasm including instruction latencies etc.
  - NVBit: Official NVIDIA tool that intercepts SASS and lets you replace it by PTX/CUDA (but not custom SASS!) - useful for very advanced debugging, e.g. getting a full memory trace (NVIDIA's profiling of memory bank conflicts in Nsight Compute is apparently based on a similar principle rather than HW counters!)

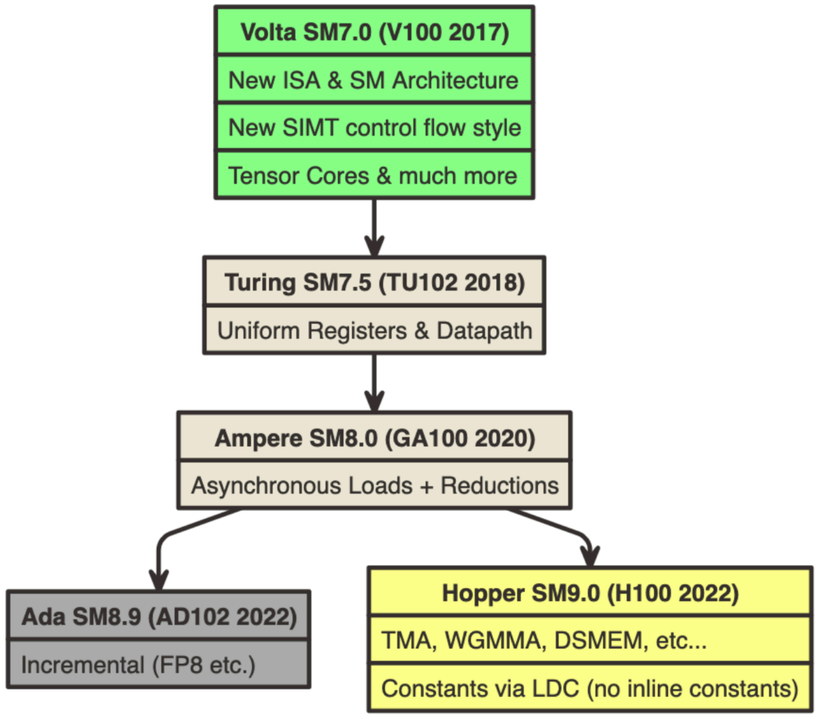
Will the real SASS please stand up?

A story of ~~love~~ CISC and RISC ~~betrayal~~
(and the ~~friends~~ **SIMT variants** we made along the way)



Will the real SASS please stand up?

If you’re “lucky", you only need to care about Hopper (and Blackwell in the future)



Looking at PTX & SASS

Godbolt / Nsight Compute / cuobjdump

- Live Demo
- Explanation of PTX & SASS for a simple RELU kernel on H100 & V100
- Looking through the user interfaces, and NVIDIA’s binary utility reference page

PTX ~= SASS for trivial cases

Inline PTX is (sometimes) your friend

- Live Demo: Integer addition
- Live Demo: A100 reduction instructions

But PTX != SASS

Inefficient SASS might blow up in your face if you don't check!

- Live Demo: Integer addition of multiple numbers
- Example: H100 INT4 Matrix Multiplication

- Forward compatibility is hard!
- Lots of non-obvious low-level optimisations (& HW bug workarounds?)
- GPUs would be a *lot* less efficient if they had to support old assembly! (hi x86)

Controlling Long-Distance Optimisation

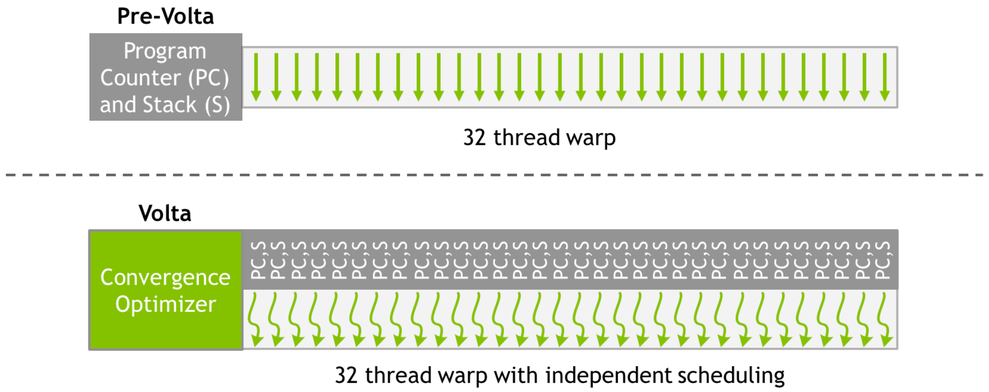
for simpler performance analysis (that’s actually useful)

- Live Demo using Godbolt
  - nanosleep, noinline, launchbounds, etc.

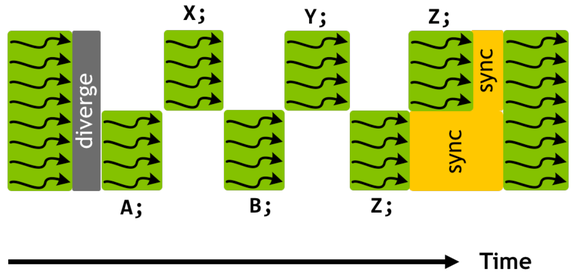
The core of the GPU: SIMT

Volta's Independent Thread Scheduling (& why you probably shouldn't care?)

- Live Demo
  - SETP & Predication
  - BSSY, BRA, BSYNC
  - BREAK, BMOV, B2R
- For a lot more background information, see Olivier Giroux's "Synchronisation is bad, but if you must… (S9329)"




```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
__syncwarp()
```



2nd diagram is potentially misleading as this is not the typical case.

(A & B would run back-to-back for 1 thread unless something happens between them; e.g. atomic/sync/…)

Profiling with Nsight Compute

SASS doesn’t show everything!

- Live Demo
- Example: Vector reads & writes of consecutive registers => liveness/rw tool
- Example: Long latency dependencies merged => stalls only on some reads

Volta & Hopper ISA Encoding

128-Bit Fixed Length (surprisingly simple & low density!)

- 128-bit Fixed Length Encoding (very low code density compared to CPUs or even other GPUs, partly offset by very aggressive prefetching).
- Votla ISA was first analysed in the famous “Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking” paper.
- There is a nice visual representation of the Turing ISA in the “Optimizing Batched Winograd Convolution on GPUs” paper below.
- Kuter Dinel’s excellent nv_isa_solver is the 1st exhaustive documentation of H100’s ISA (including corner cases): https://github.com/kuterd/nv_isa_solver

PPoPP ’20, February 22–26, 2020, San Diego, CA, USA Da Yan, Wei Wang, and Xiaowen Chu

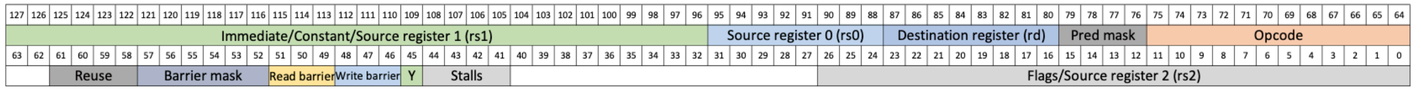
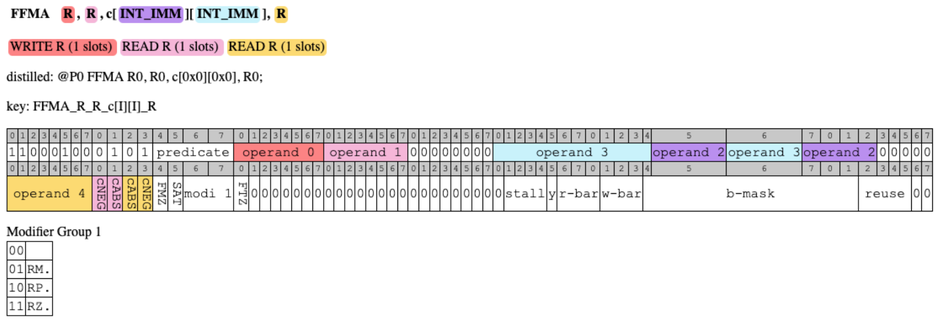


Figure 6. Instruction encoding on Nvidia Volta and Turing. White parts are left unused and are filled with 0.



Don’t trust everything you read about GPUs

(NVIDIA’s forums are pretty great though)

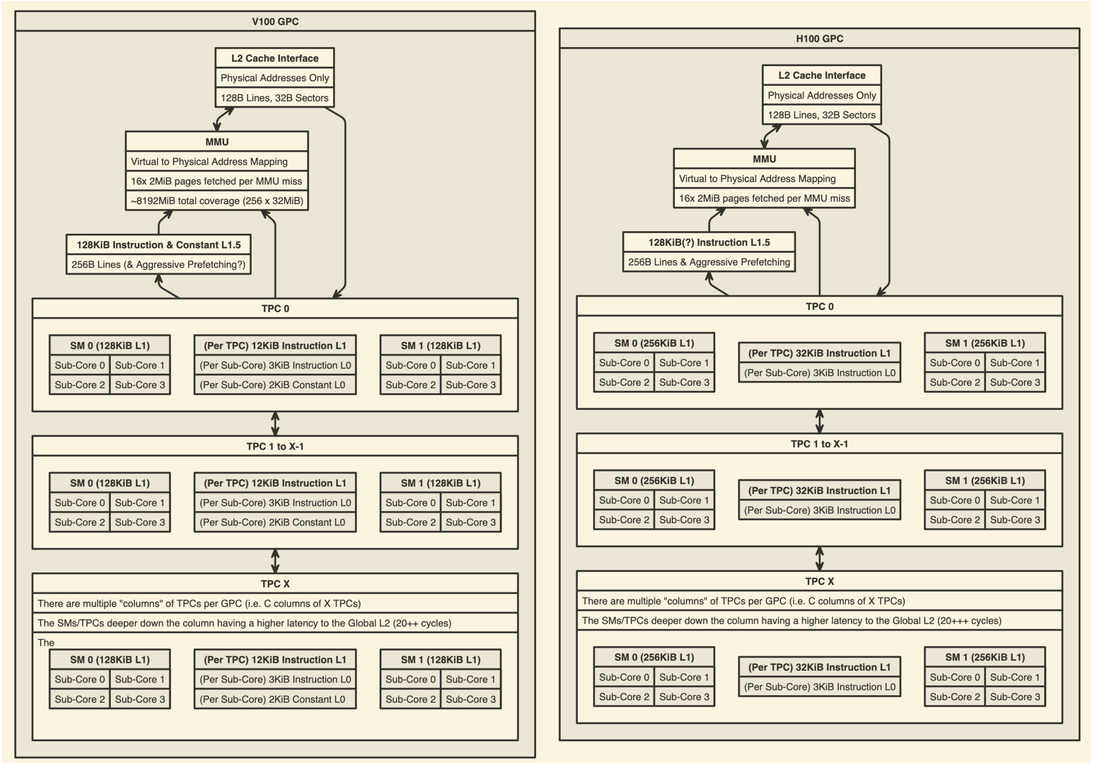
- There’s a lot of great official & unofficial presentations/papers(/forum posts!) on NVIDIA GPUs **(see Appendix)**
  - But… it’s impossible for any 3rd party to get all the details right with so many variables
  - Internal HW specifications *try* to be perfectly accurate, but *nothing* is ever both complete & perfect :(

My attempt at the GPC memory hierarchy:

(also wrong! e.g. H100 has rows & columns of TPCs)

(you probably shouldn’t care about any of this)

(3KiB L0 I$ is beautiful in how irrelevant it is!)



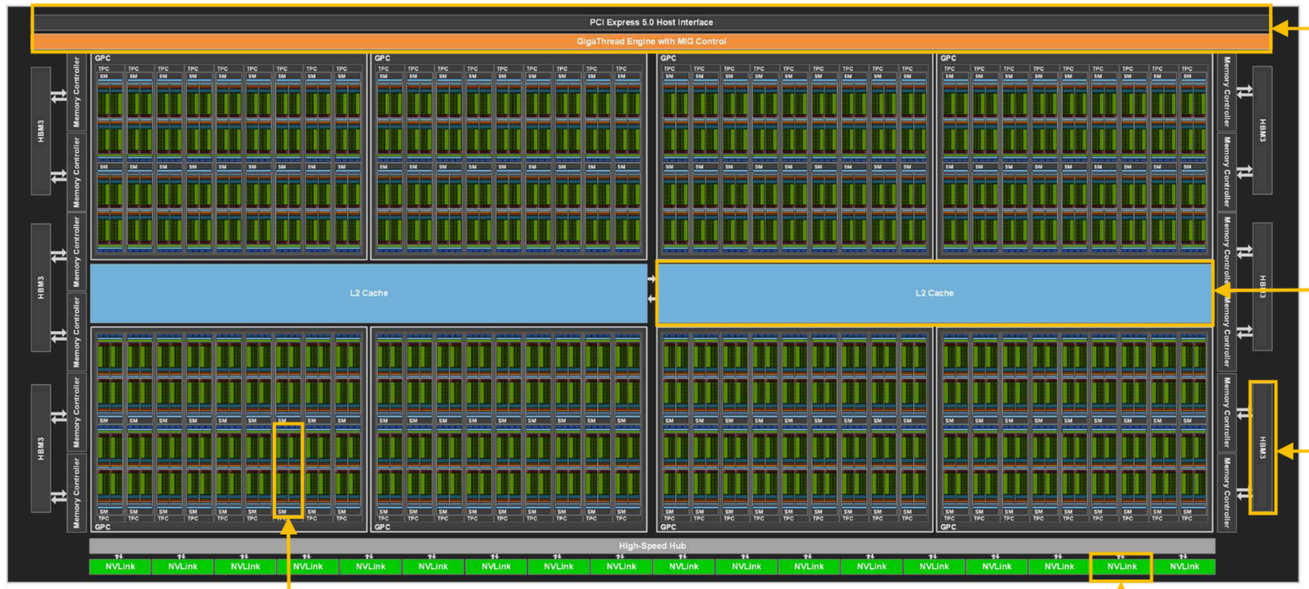
Let’s try to dig deep…

http://hc34.hotchips.org/assets/program/conference/day1/GPU%20HPC/HC2022.NVIDIA.Choquette.vfinal01.pdf

Unfortunately NVIDIA’s Hot Chips presentations have become more focused on the system level over time, and we need to dig deeper into the SM…

HOPPER H100 TENSOR CORE GPU

80B Transistors, TSMC 4N



2nd Gen Multi-Instance GPU
Confidential Computing
PCIe Gen5

New Memory System
World’s First HBM3 DRAM
Larger 50MB L2

132 SMs 2x Performance per Clock
4th Gen Tensor Core
Thread Block Clusters

4th Gen NVLink 900GB/s total BW
New SHARP support
NVLink Network

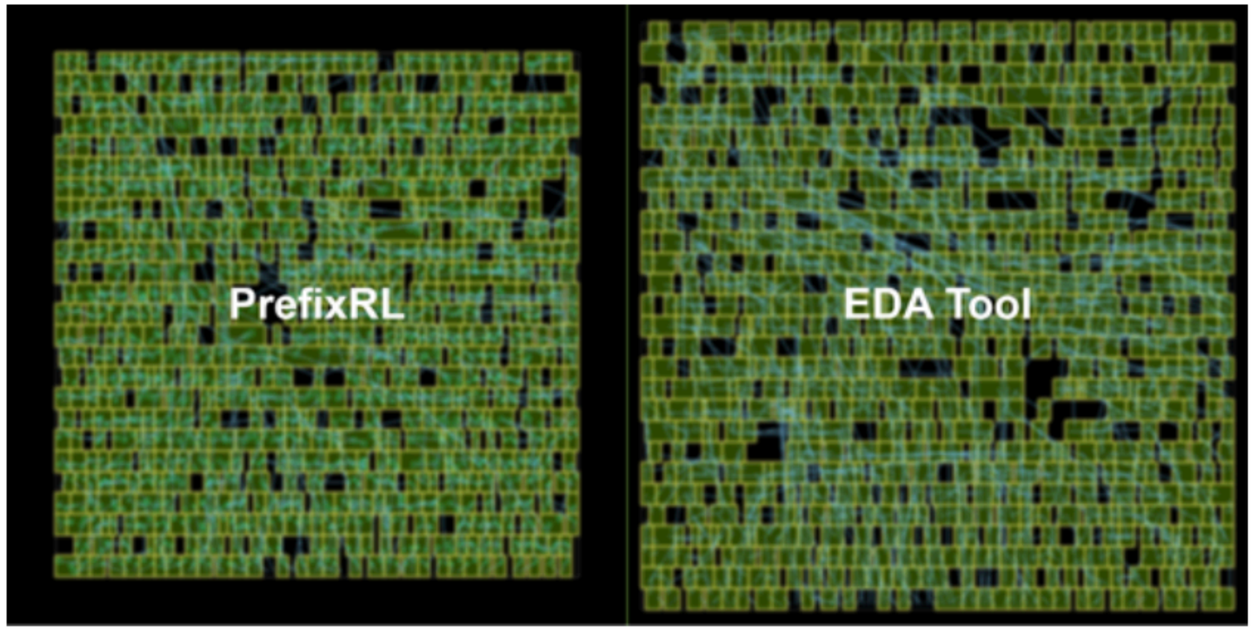
Deeper…


SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
FP64
TENSOR CORE
4th GENERATION
LD/ ST
SFU
Tensor Memory Accelerator
256 KB L1 Data Cache / Shared Memory
Tex

Okay, maybe not that deep!

https://developer.nvidia.com/blog/designing-arithmetic-circuits-with-deep-reinforcement-learning/

What if AI could learn to design these circuits? In PrefixRL: Optimization of Parallel Prefix Circuits using Deep Reinforcement Learning, we demonstrate that not only can AI learn to design these circuits from scratch, but AI-designed circuits are also smaller and faster than those designed by state-of-the-art electronic design automation (EDA) tools. The latest NVIDIA Hopper GPU architecture has nearly 13,000 instances of AI-designed circuits.



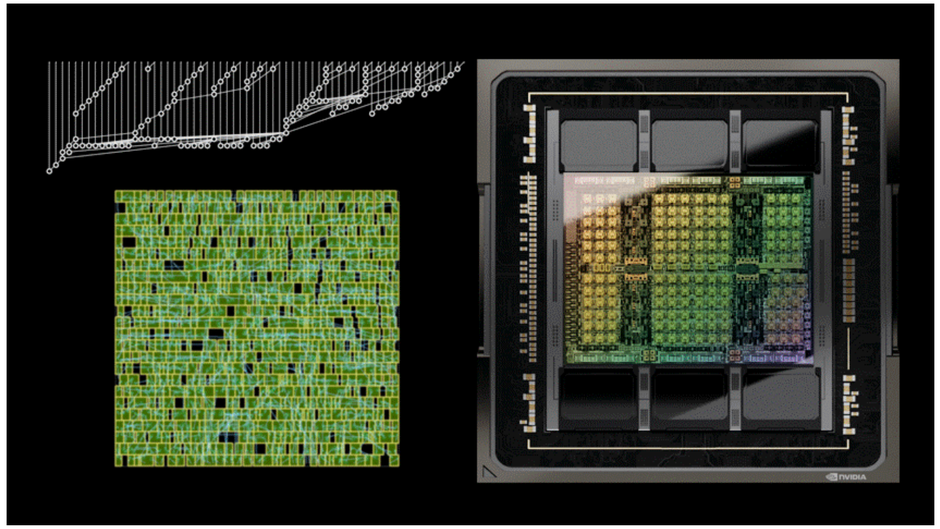
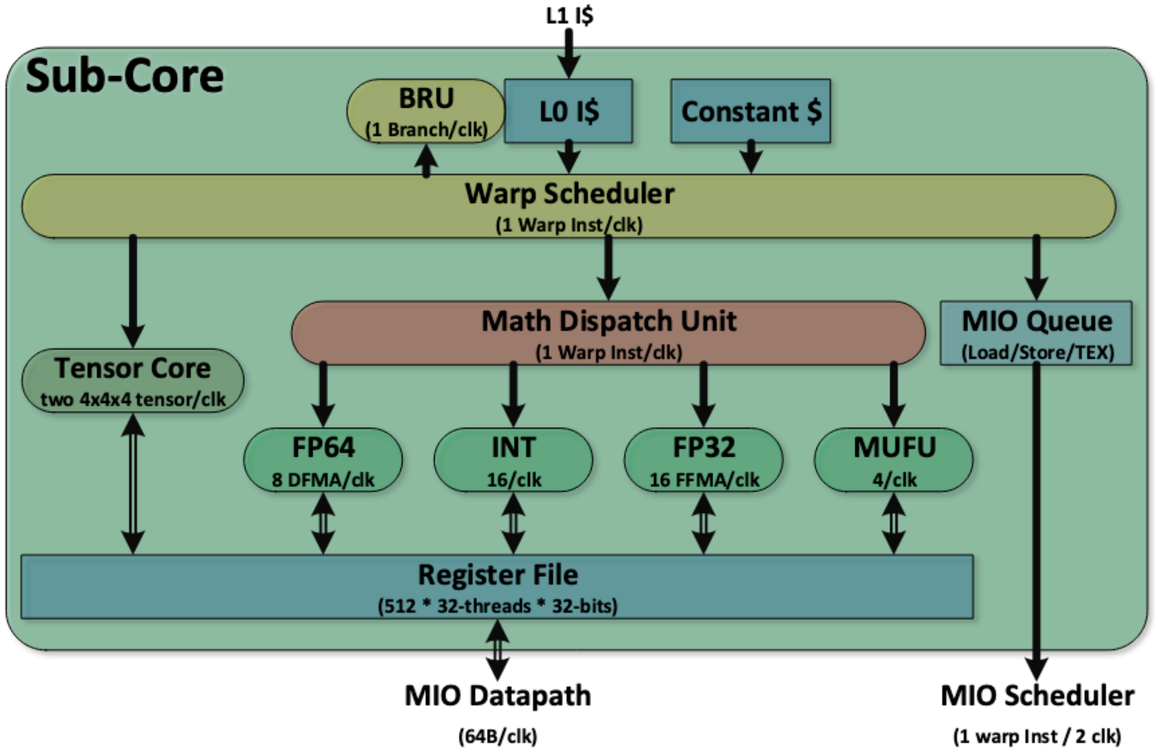


Figure 1. 64b adder circuits designed by PrefixRL AI (left) are up to 25% smaller than that designed by a state-of-the-art EDA tool (right) while being as fast and functionally equivalent

Volta Hot Chips Presentation

https://old.hotchips.org/wp-content/uploads/hc_archives/hc29/HC29.21-Monday-Pub/HC29.21.10-GPU-Gaming-Pub/HC29.21.132-Volta-Choquette-NVIDIA-Final3.pdf
(I believe MUFU is accessed through a separate MIO instruction queue, and I think L1 I$ is per TPC rather than per SM at least in later designs, but close enough!)

SUB-CORE



Warp Scheduler

- 1 Warp instr/clk
- L0 I$, branch unit

Math Dispatch Unit

- Keeps 2+ Datapaths Busy

MIO Instruction Queue

- Hold for Later Scheduling

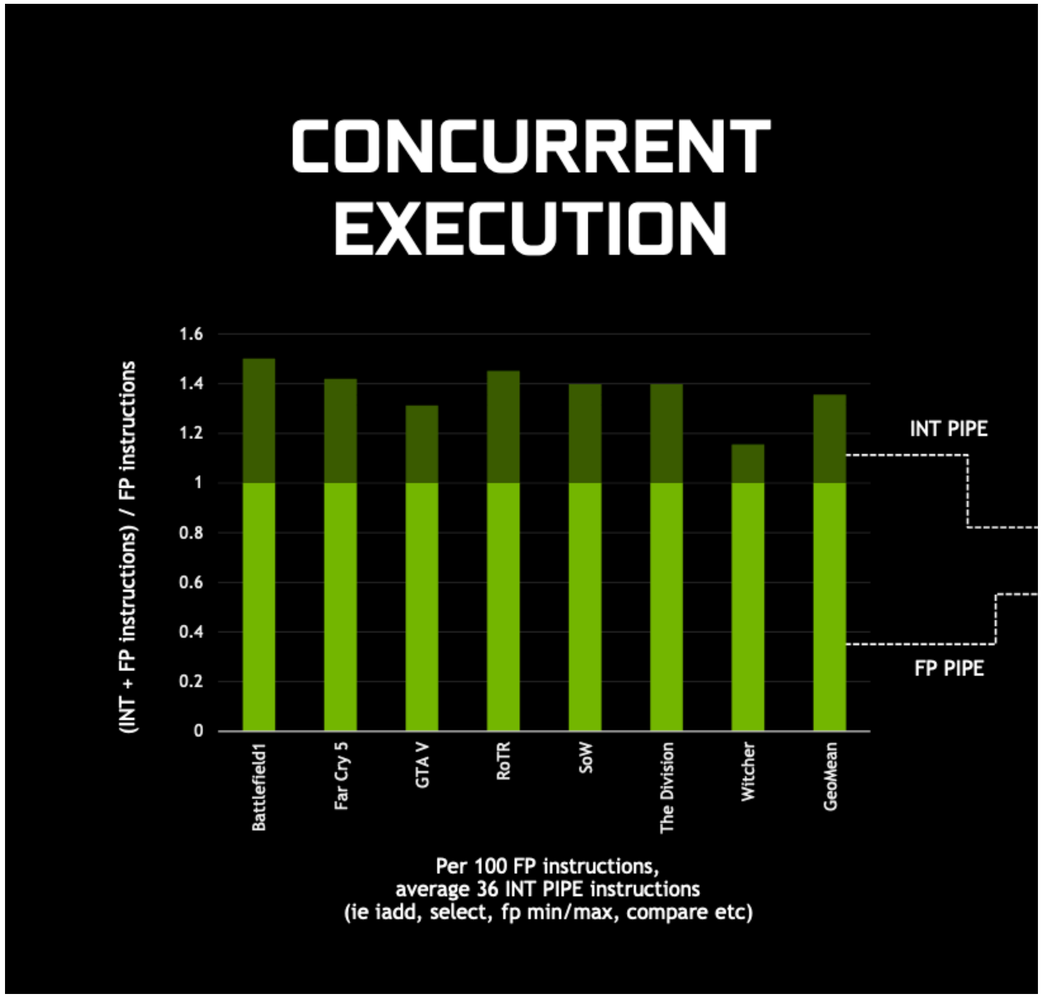
Two 4x4x4 Tensor Cores

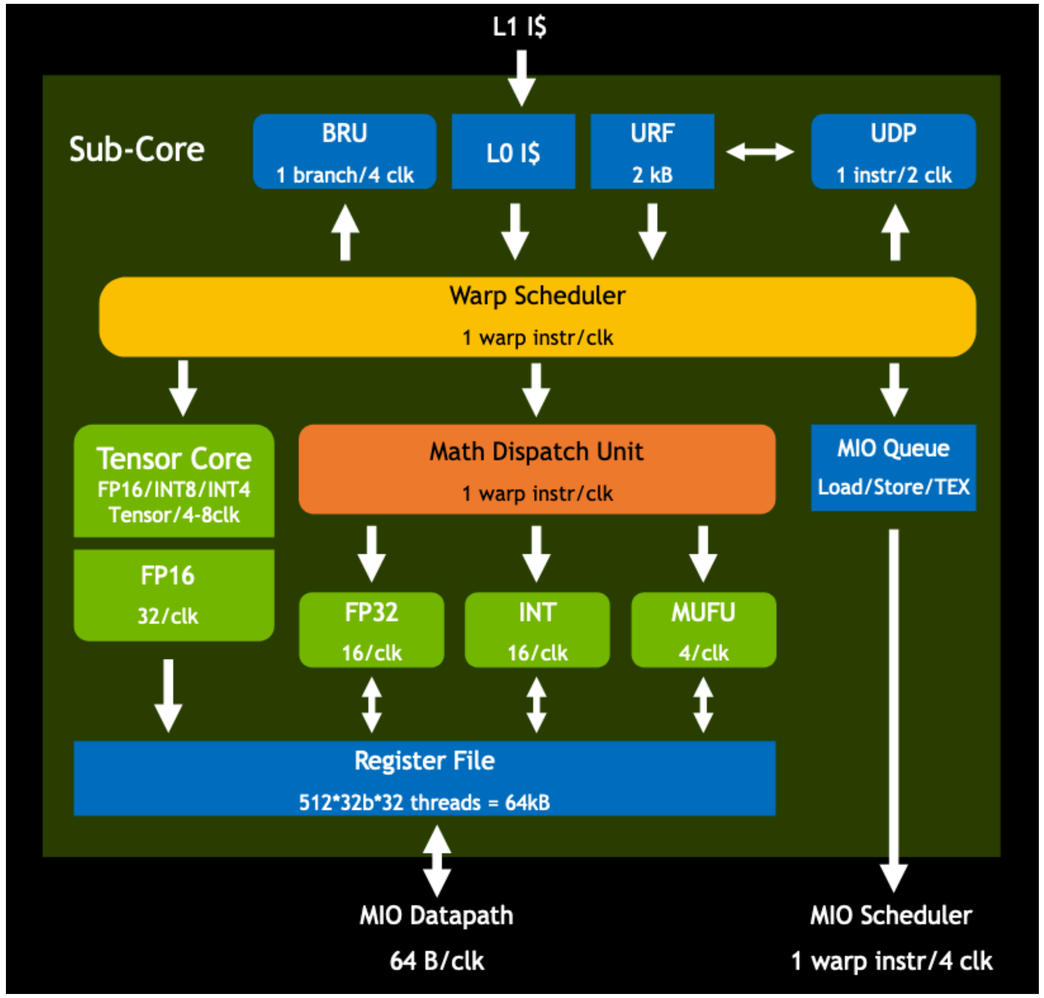
Turing Hot Chips Presentation

https://old.hotchips.org/hc31/HC31_2.12_NVIDIA_final.pdf

Correct but misleading: IMAD (32-bit integer *multiply* and add) is in “FP” pipe, and FP32 min/max is in “INT” pipe.

==> It can co-issue iadd/cmp/etc. with 32-bit imad in memory address calculation heavy kernels. In H100/GA102, there is a new “FMALite” pipe, more on that later…





Turing Uniform Registers

Note the ‘cx[UR20][0x64] for FSETP: Before Hopper, NVIDIA could access Constant Memory (e.g. kernel parameters) directly in arithmetic instructions.
If that missed in the cache, and an “arithmetic” instruction could take 1000+ cycles to come back from DRAM! Very CISC-like.
Hopper uses “LDC” instructions to load from constant memory to registers.

UNIFORM DATAPATH & REGISTER FILE

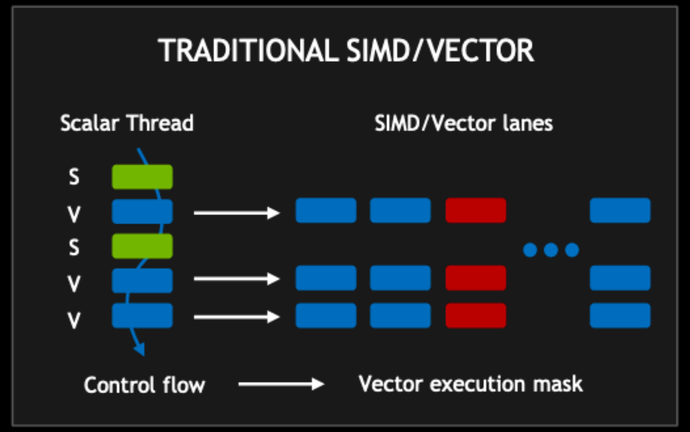
Goal: Exploit redundant computation & data across multiple threads while preserving our Independent Thread Scheduling model

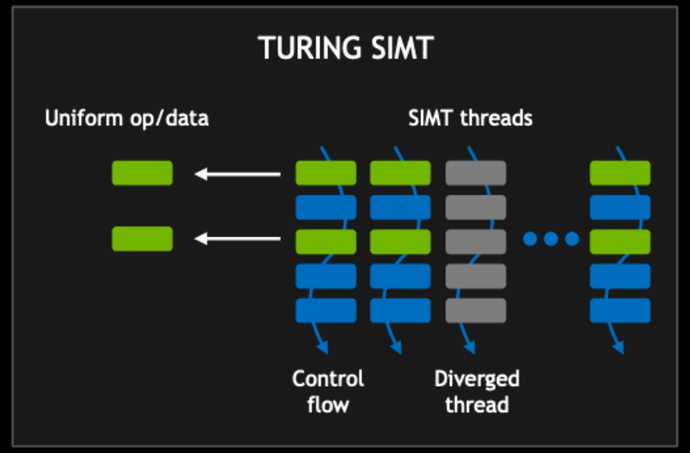
Automatically promote ops/data when warp-uniform data is detected

- Compiler + hardware assist
- Executed by an independent datapath
- ‘Reverse vectorization’

Example: Enabling DX12 bindless constants with URF/UDP on Forza MS7 yielded +12.7% performance

```
…
UIADD3          UR13, UR9, 0x300001, URZ
ULDC.64         UR20, [UR6 + 0x18], !UP7
UIADD3          UR6, UR8, UR10, URZ
UIADD3          UR8, UR9, 0x300002, URZ
FSETP.NEU.FTZ.AND P1, PT, R15, cx[UR20][0x64], PT
ULOP3.LUT       UR12, UR13, 0xfffff, URZ, 0xc0, !UP7
…
```





Barrier: the most overloaded word in CUDA?

Maybe we should just build a wall instead

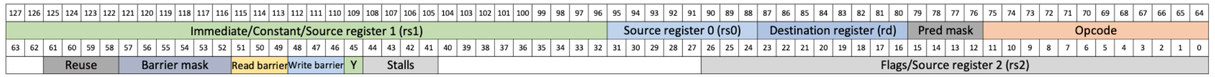


Figure 6. Instruction encoding on Nvidia Volta and Turing. White parts are left unused and are filled with 0.

- Dependencies for long latency instructions are managed through read and write "barriers" (not documented or shown in SASS & unrelated to the others!)
- __syncthreads() uses the "BAR (Barrier Synchronisation)" instruction (considered a 'type of barrier' in some presentations & other APIs)
- TMA, Asynchronous Copy, and Hopper Tensor Cores all have their own unique "barrier" & synchronisation systems :(
- etc…

Table 6: Asynchronous copies with possible source and destinations memory spaces and completion mechanisms. An empty cell indicates that a source-destination pair is not supported.

Direction		Completion mechanism	
Destination	**Source**	**Asychronous copy**	**Bulk-asynchronous copy (TMA)**
Global	Global		
Global	Shared::cta		Bulk async-group
Shared::cta	Global	Async-group, mbarrier	Mbarrier
Shared::cluster	Global		Mbarrier (multicast)
Shared::cta	Shared::cluster		Mbarrier
Shared::cta	Shared::cta		

Instruction Throughput

NVIDIA reveals more than usual here, but… how do these rates add up?
Key bottleneck: 1 instruction per clock per SMSP (4 per SM)!

5.4.1. Arithmetic Instructions

The following table gives the throughputs of the arithmetic instructions that are natively supported in hardware for devices of various compute capabilities.

Table 4: Throughput of Native Arithmetic Instructions. (Number of Results per Clock Cycle per Multiprocessor)

Compute Capability	5.0, 5.2	5.3	6.0	6.1	6.2	7.x	8.0	8.6	8.9	9.0
16-bit floating-point add, multiply, multiply-add	N/A	256	128	2	256	128	256[3]		128	256
32-bit floating-point add, multiply, multiply-add	128		64	128		64		128		
64-bit floating-point add, multiply, multiply-add	4		32	4		32[5]	32	2		64
32-bit floating-point reciprocal, reciprocal square root, base-2 logarithm (`__log2f`), base 2 exponential (`exp2f`), sine (`__sinf`), cosine (`__cosf`)	32		16	32		16				

CUDA C++ Programming Guide “5.4.1 Arithmetic Instructions”

(an unexpected ray of microarchitecture in a sea of programming models!)

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#arithmetic-instructions

Warp Scheduling & Dependencies

Understanding Warp Stalls & States in Nsight Compute

- Live Demo
  - “What does it all mean?!” (Nsight Compute)
  - NVIDIA’s documentation for its performance tools
  - 2xFP16 or Die Trying: Compiler engineers hate this one simple trick *(if we have time)*

Why do we stall? Little's Law

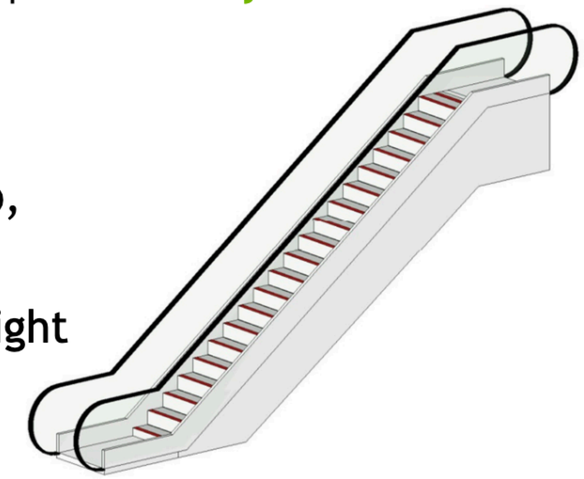
Hiding ~200 (L2) to ~1000 (far away HBM) cycles of latency requires a lot of "in flight" work! (ILP+TLP)

Little's Law

For Escalators

A step arrives every 2 seconds
Bandwidth: 0.5 person/s
20 steps tall : **Latency** = 40 seconds

- One person in flight ?
  Achieved bandwidth = 0.025 person/s
- To saturate bandwidth:
  Need one person arriving with every step,
  we need 20 persons in flight
- Need **Bandwidth x Latency** persons in flight



```
__global__
void kernel(const float * __restrict__ a,
            const float * __restrict__ b,
            float * __restrict__ c)
{
  int tid = blockIdx.x * blockDim.x + threadIdx.x;
  int stride = blockDim.x * gridDim.x;

  #pragma unroll 2
  for (int i = 0; i < 2; i++) {
    const int idx = tid + i * stride;
    c[idx] += a[idx] * b[idx];
  }
}
```

```
load a[i1]
load b[i1]
load c[i1]
load a[i2]
load b[i2]
fma c[i1], a[i1], b[i1]
store c[i1]
load c[i2]
fma c[i2], a[i2], b[i2]
store c[i2]
```

20 bytes in-flight

8 bytes in-flight

4 bytes in-flight

Arun's Corollary: As an indirect consequence of the Central Limit Theorem, all threads will sometimes be executing the 'wrong part' of the kernel at the same time… :(

ADVANCED OPTIMISATION: GELU & Perf/W

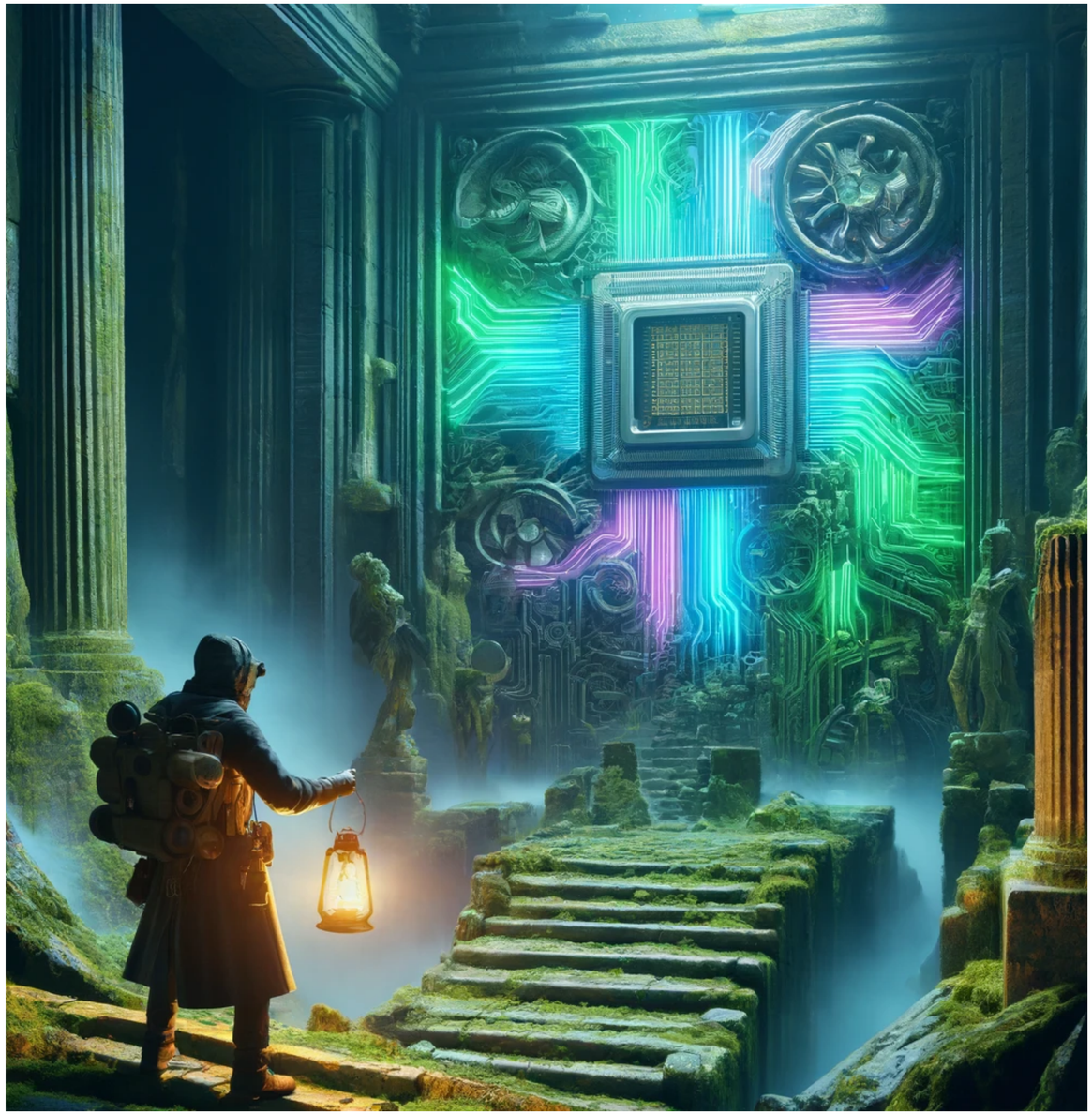
We're (probably) behind schedule so I'm going to assume a lot more background knowledge now…

- Element-wise operations should just be DRAM limited… right?
  1. gelu_forward microbenchmarks and x128/Vec4
  2. llm.c in Nsight Compute including SASS
  3. MUFU.TANH & the need for inline PTX
  4. FMA reordering & what the compiler can't do
  5. Why are we still below 100% DRAM?! A tale of HBM, reads, and writes…
- Going all in: "L2 Side Aware Memcpy" (& GELU / Elementwise)

Conclusion

HW/SW co-optimisation never ends…
until the HW does!

Looking forward to exploring Blackwell :)



Appendix

GPU uArch Papers & Presentations

Planning to update this page in the future with more links :)

https://github.com/ademeure/ademeure.github.io/blob/master/NVIDIA-Microarchitecture-Sources.md