Paper: Making Deep Learning Go Brrrr From First Principles

By Horace He (2022)

Paper Overview

- Improving the performance of deep learning models on GPU **is not a function of simply scaling GPU compute.**
- Efficiency is driven by
  - Data flows (intra/inter GPU)
  - Type of operations (element wise ops vs mat muls)
- Key Question: *What should we optimize? Where to start?*
- Consider different abstraction levels (hardware -> kernels -> framework).

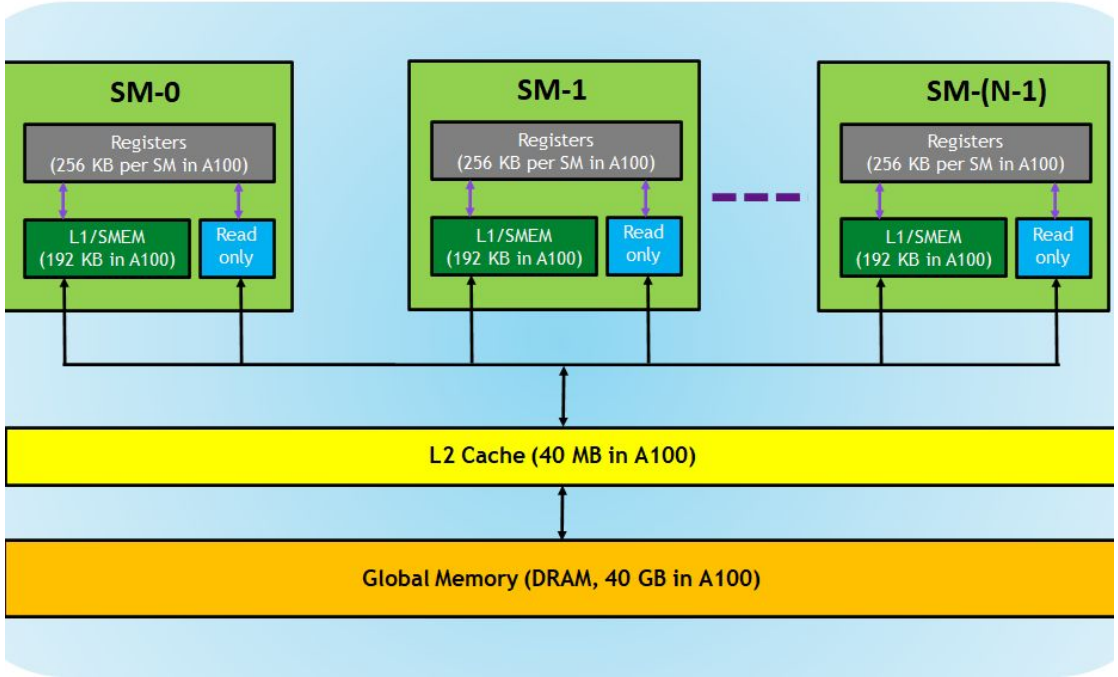
The 3 Performance Regimes

- **Compute-bound** - when the GPU is busy doing actual floating point operations (FLOPs)
- **Memory (bandwidth)-bound** - when data transfer (global memory, DRAM, moving tensors) is the limiter.
- **Overhead-bound** - all extra costs (Python dispatch, kernel launch, framework overhead, small ops) that arent compute or memory.

The Hardware

Graphical Processing Units (GPUs) are composed highly parallel processor cores (SMs) architectures designed to accelerate matrix-multiplication.

- Each SM
  - has registers (fastest)
  - Shared L1 cache
- Shared across all SMs
  - L2 cache
  - Global memory



GPU Speed (FLOPs)

- Tensor cores = *extremely* fast (hundreds of TFLOPs)
- Standard cores (FP32/FP64) = much slower (tens of TFLOPs)
- **GPUs are optimized for matmul on tensor cores.**
- **Other ops won't reach peak FLOPs.**

NVIDIA A100 TENSOR CORE GPU SPECIFICATIONS (SXM4 AND PCIE FORM FACTORS)

	A100 40GB PCIe	A100 80GB PCIe	A100 40GB SXM	A100 80GB SXM
FP64		**9.7 TFLOPS**		
FP64 Tensor Core		**19.5 TFLOPS**		
FP32		**19.5 TFLOPS**		
Tensor Float 32 (TF32)		**156 TFLOPS | 312 TFLOPS***		
BFLOAT16 Tensor Core		**312 TFLOPS | 624 TFLOPS***		
FP16 Tensor Core		**312 TFLOPS | 624 TFLOPS***		

Common DNN Operations

- Reductions - pooling layers, normalization.
- Dot Products - matmul. Fully connected layers
  - Convolution can also be represented as matrix-vector and matrix-matrix multiplies.
- Elementwise - ReLU, add, bias, scale

“Normalization and pointwise ops actually achieve 250x less FLOPS and 700x less FLOPS than our matmuls”

FLOPs on BERT for different operator types

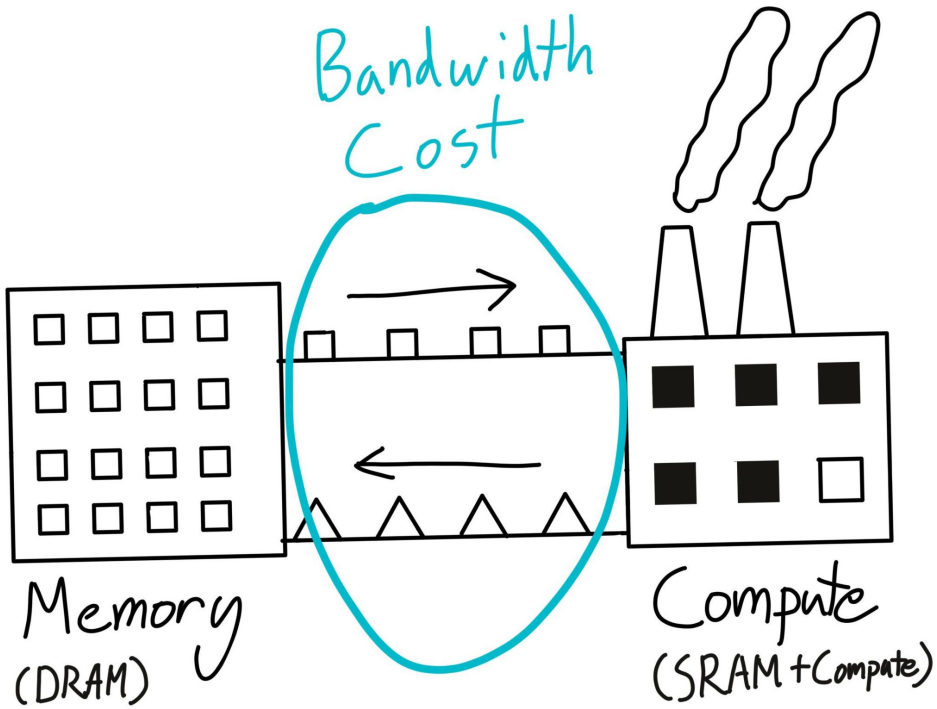
Table 1. Proportions for operator classes in PyTorch.

Operator class	% flop	% Runtime
△ Tensor contraction	99.80	61.0
□ Stat. normalization	0.17	25.5
○ Element-wise	0.03	13.5

Tensor contractions = ~100% of FLOPs, **but only ~60% of runtime.**

Normalization + elementwise = <1% FLOPs, **but >35% runtime.**

Compute ≠ Everything: The Factory Abstraction

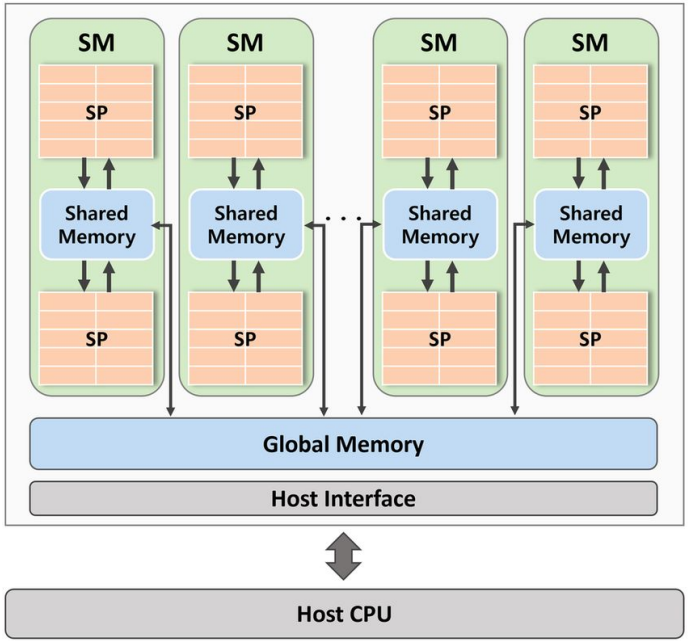


Data movement (bandwidth) often limits performance more than compute.

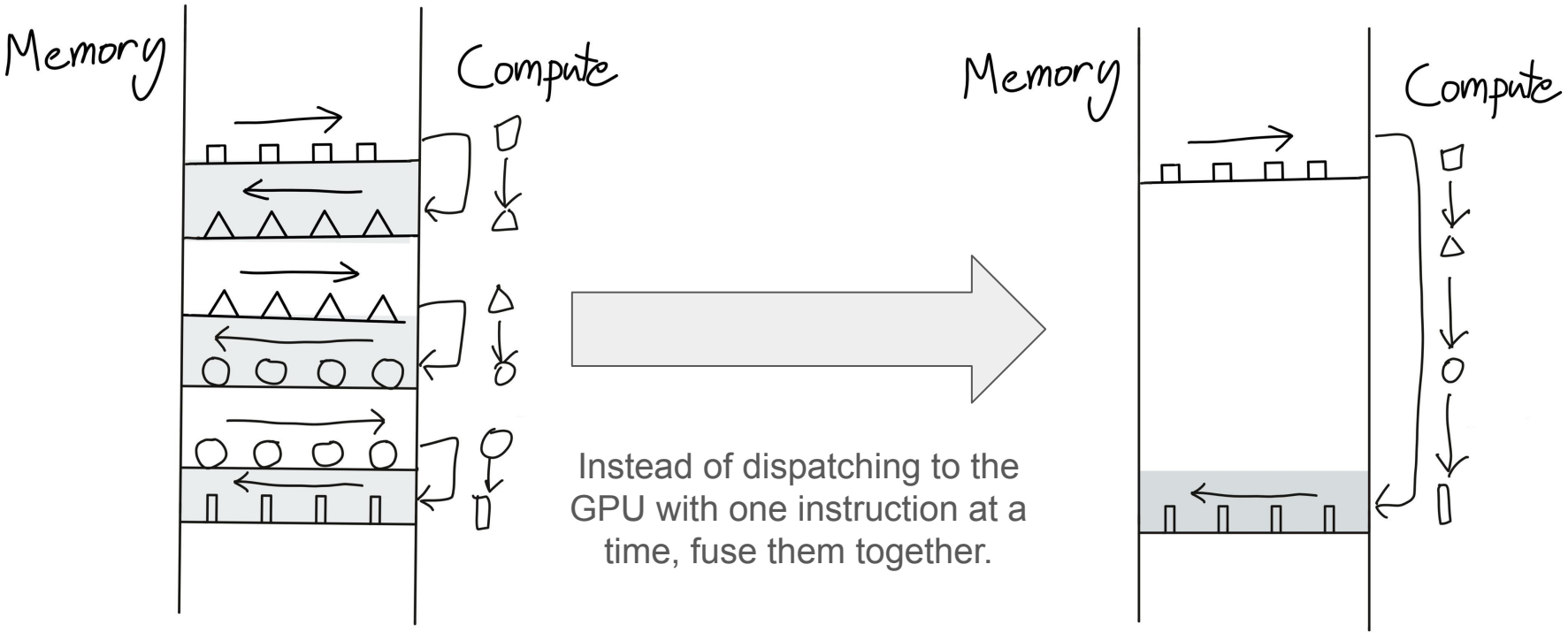
Memory Bandwidth

Even with fast matmuls, performance is often limited by how quickly tensors move through the GPU memory hierarchy.

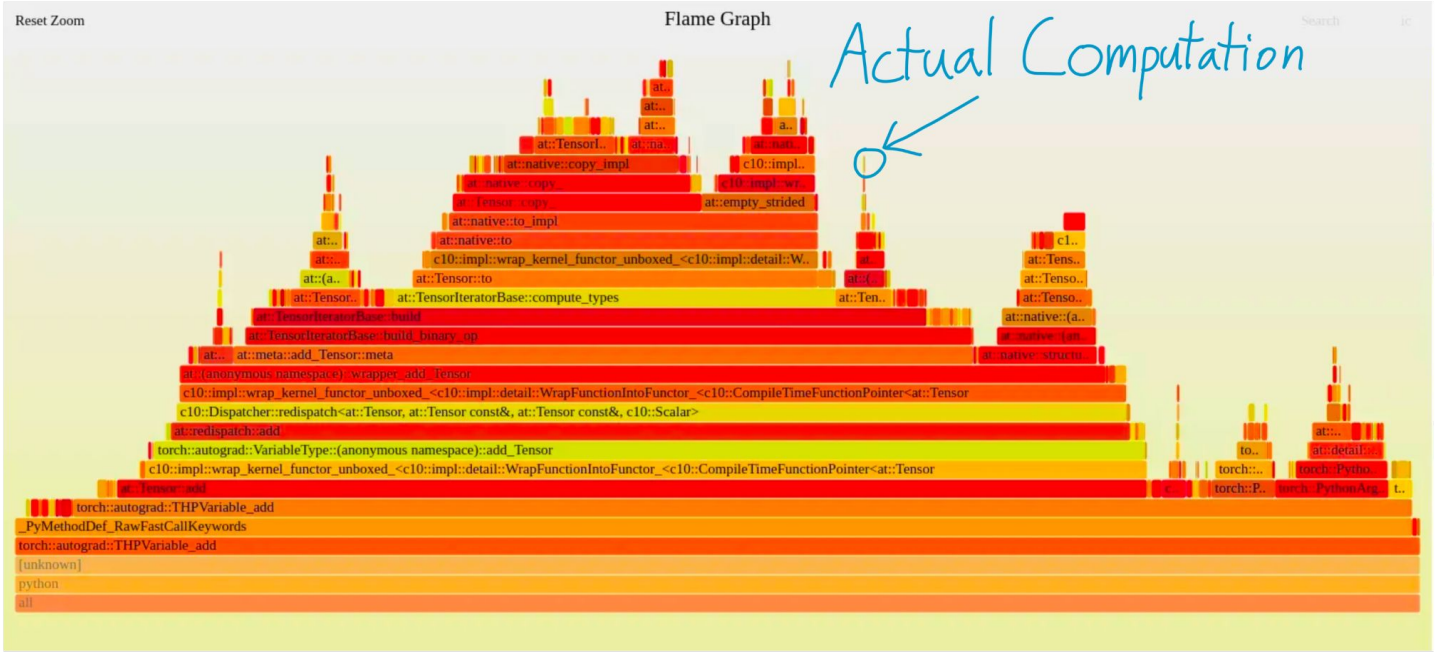
- Host Interface → GB/s
- Global mem → 1-2 TB/s
- Shared mem → multi-TB/s
- Registers (inside SMs) → tens of TB/s
-



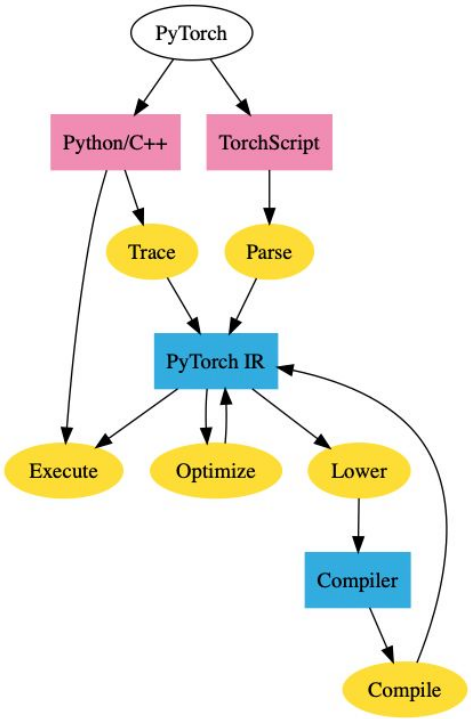
Kernel Fusion for high bandwidth costs…



Overhead


Reset Zoom
Flame Graph
Actual Computation
at::native::copy_impl
at::empty_strided
at::native::to_impl
at::native::to
c10::impl::wrap_kernel_functor_unboxed_<c10::impl::detail::W..
at::Tensor::to
at::TensorIterator..
at::TensorIteratorBase::compute_types
at::TensorIteratorBase::build
at::TensorIteratorBase::build_binary_op
at::meta::add_Tensor::meta
at::(anonymous namespace)::wrapper_add_Tensor
c10::impl::wrap_kernel_functor_unboxed_<c10::impl::detail::WrapFunctionIntoFunctor_<c10::CompileTimeFunctionPointer<at::Tensor
c10::Dispatcher::redispatch<at::Tensor, at::Tensor const&, at::Tensor const&, c10::Scalar>
at::redispatch::add
torch::autograd::VariableType::(anonymous namespace)::add_Tensor
c10::impl::wrap_kernel_functor_unboxed_<c10::impl::detail::WrapFunctionIntoFunctor_<c10::CompileTimeFunctionPointer<at::Tensor
torch::autograd::THPVariable_add
_PyMethodDef_RawFastCallKeywords
torch::autograd::THPVariable_add
[unknown]
python
all

Optimizations Galore



Performance Regime	Plausible Solutions
Overhead-Bound	Tracing, Operator Fusion, don't use Python, a *real* JIT :^)
Bandwidth-Bound	Operator Fusion
Compute-Bound	Use Tensor Cores, give Nvidia more money

Rooflines for LLM Scaling

Overview / Agenda

- Motivation
- What is a GPU?
- Networking
- Rooflines

Motivation

- There are various bottlenecks that an LLM training system may be governed by
- Primarily, they are compute and networking limitations
- The goal of this work was to quantitatively describe the conditions in which each of those factors become the bottleneck
- In pursuit of that, some background is required

What is a GPU?

- For the purpose of this talk, GPUs are little more than matrix multiplication accelerators
- They’re more general purpose than TPUs in that they have contain more modular hardware components that are capable of things *besides* matrix multiplication
- And various advancements have resulted in more memory capacity, memory bandwidth, etc.
- But in the LLM domain, their performance is dominated by their matrix multiplication capabilities

Networking

- For NVIDIA GPUs, there are two types of links that they can use to communicate
- There is NVLink, which is an NVIDIA-specific protocol/interface that’s used for communication between GPUs within a pod
- NVLink allows for NVSwitches, which allow for 1-hop communication between GPUs in a pod
- And there is a normal network link, like infiniband or ethernet, which allows for communication over an arbitrary number of hops

Pods

- NVIDIA organizes their NVLink-connected GPUs into pods, which have standard sizes
- For the H100, the standard size of a pod is 8 GPUs, with 4 NVSwitches
- In addition, each H100 GPU is paired with a 400G NIC

Networking beyond Pods

- Between pods, packets flow across standard networking interfaces like IB and ethernet
- The topology of these networks is left up to operators
- Many decide to use a fat-tree topology, which guarantees some level of bisection bandwidth
- Depending on sharding scheme, may place additional constraints on roofline

Data Parallelism

$$T_{\text{math}} = \frac{2 \cdot 2 \cdot 2 \cdot BDF}{X \cdot C}$$

$$T_{\text{comms}} = \frac{2 \cdot 2 \cdot 2 \cdot DF}{W_{\text{collective}}}$$

Therefore, for $T_{\text{math}} > T_{\text{comms}}$, we need $B/(XC) > 1/W_{\text{collective}}$ or

$$\frac{B}{X} > \frac{C}{W_{\text{collective}}}$$

Tensor Parallelism

$$T_{\text{math}} = \frac{2 \cdot 2 \cdot BDF}{Y \cdot C}$$

$$T_{\text{comms}} = \frac{2 \cdot 2 \cdot BD}{W_{\text{collective}}}$$

$$Y < \frac{F \cdot W_{\text{collective}}}{C}$$

Expert Parallelism

$$T_{\text{math}} = \frac{4 \cdot B \cdot k \cdot D \cdot F}{Z \cdot C}$$

$$T_{\text{comms}} = \frac{4 \cdot B \cdot D \cdot (Z - 8)}{W \cdot Z} \cdot \min\left(\frac{8 \cdot k}{Z}, 1\right)$$

We either need $K > Z/8$ with $F > \alpha \cdot (Z - 8)/k$ or $Z \gg K$ and $F > 8 \cdot \alpha$, where $\alpha = C/W$. This gives you two domains in which expert parallelism is possible, one with a small amount of expert parallelism (roughly 2-node) and small F, or one with large F and Z arbitrarily large (up to E-way expert parallelism).

Pipeline Parallelism

$$T_{\text{total PP comms}} = \frac{2BD}{W \cdot N_{\text{MB}}} \cdot (N_{\text{MB}} + N_{\text{stages}} - 2)$$

$$T_{\text{per-layer comms}} \approx 1.5 \cdot \frac{2BD}{W \cdot N_{\text{layers}}}$$

Summary

- **Data parallelism or FSDP (ZeRO-1/3) requires a local batch size of about 2500 tokens per GPU**, although in theory in-network reductions + pure DP can reduce this somewhat.
- **Tensor parallelism is compute-bound up to about 8-ways** but we lack the bandwidth to scale much beyond this before becoming comms-bound. This mostly limits us to a single NVLink domain (i.e. single-node or need to use GB200NVL72 with to 72 GPUs).
- **Any form of model parallelism that spans multiple nodes can further reduce the cost of FSDP**, so we often want to mix PP + EP + TP to cross many nodes and reduce the FSDP cost.
- **Pipeline parallelism works well if you can handle the code complexity of zero-bubble pipelining and keep batch sizes fairly large to avoid data-parallel bottlenecks.** Pipelining usually makes ZeRO-3 impossible (since you would need to AllGather on each pipeline stage), but you can do ZeRO-1 instead.

At a high level, this gives us a recipe for sharding large models on GPUs:

- For relatively small dense models, aggressive FSDP works great if you have the batch size, possibly with some amount of pipelining or tensor parallelism if needed.
- For larger dense models, some combination of 1-2 node TP + many node PP + pure DP works well.
- For MoEs, the above rule applies but we can also do expert parallelism, which we prefer to TP generally. If $F > 8 * C/W_{\text{node}}$, we can do a ton of multi-node expert parallelism, but otherwise we're limited to roughly 2-node EP.

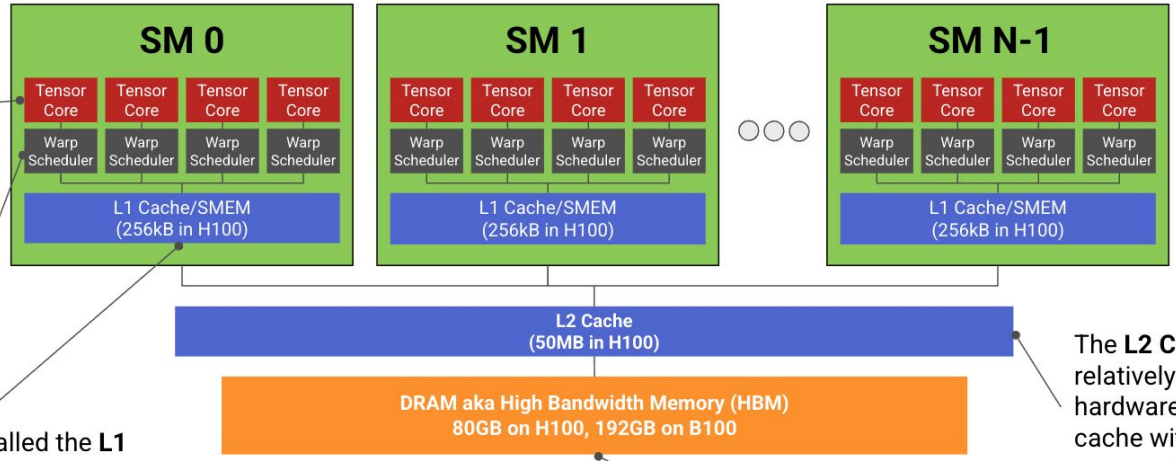
Terminology

The GPU houses both Dynamic RAM and static RAM. DRAM is for global memory, that is memory persisted and accessible across the different cores of the GPU. Shared memory (SMEM) is implemented on individual cores. When data reaches the GPU but are not being actively used in a computation, they live on the DRAM. When an operation on data is to be run, the instruction and associated data move onto the shared memory of one or more cores.

SMEM is a fast on chip cache compared to global memory.

The **Tensor Core** performs matrix multiplications and accounts for most of the chip FLOPs/s, similar to TPU MXU

The **Warp Scheduler** is a SIMD vector unit like the TPU VPU with 32 lanes, called "CUDA Cores". All lanes must perform the same operation in each cycle.



SMEM (often called the **L1 Cache**) is a small, very fast on-chip cache that can be programmer controlled. Similar to TPU VMEM but much smaller.

The **L2 Cache** is a relatively large hardware-controlled cache with faster memory bandwidth.

Abstract layout of an H100/B100 GPU

DRAM or **HBM** stores parameters, activations, optimizer state, etc.