The purpose of this is to answer some common vocab questions you might have upon arrival to GPU MODE & ML System Performance. It's not really an entry point to the space, more like a pocket book companion or cheat sheet. Several great entry points are the resource stream, the lectures, & PMPP.

Its intended audience is newcomers to this particular domain that have terminology questions that may seem obvious to those who have spent a lot of time in it.

Kernel = GPU/device function (not the OS kernel, not a convolution matrix, just a function you write to run on the GPU)

Kernel Fusion - the coolest way of saying doing multiple operations using at most smem before needing to access gmem

CPU - Central Processing Unit - Host

- Modern CPUs consist of many cores (consumer CPUs usually 2-16, server CPUs usually 8-30)
- A single core (deals with 1 thread (process) in any instant), achieves perceived parallelism by **context switching** (saving where it was in the process (Program Counter - PC) and data relevant to that thread, and loading in that of another process) between threads, which is expensive time-wise (as it must read & write to main memory, unless there's a cache hit, or the control unit of a core has room for 2 contexts, the latter are called **multithreaded CPU cores**), but occurs fast enough that we don't notice it.
- What decides when a CPU core changes processes is the OS *scheduler*, which is a process itself, a *kernel*-level process, in this context *kernel* basically meaning highest permission level
- The way CPU processes are handled, context switching, *ensures* data from one process is not accessible by other processes

GPU - Graphics Processing Unit - Device

- When one says "GPU cores", they are likely talking about
    - **compute cores = CUDA cores = streaming processors** i.e. INT32, FP32 units within SMs - the number of them dictate how many operations of those data types can be done concurrently
    - a closer analogy to a CPU core is a **processing block** on an **SM** (Streaming Multiprocessor, not Shared Memory - smem), because this is the level the control units are located (hold PC and process data). Since there is dedicated space on processing blocks for this information, context switches are free
- GPUs are programmed following the **SPMD - Single Program Multiple Data** model, meaning the same program is dispatched to multiple units (SMs). Each SM works on the segment of data it's responsible for, and within an SM, threads collaborate on that data.
- SPMD *often* employs **SIMD - Single Instruction Multiple Data -**like instructions, meaning multiple units executes one instruction simultaneously

- SMs employ **SIMT - Single Instruction Multiple Thread** architecture, which is like SIMD, but a key difference is that SIMT "[specifies] the execution and branching behavior of a single thread … [enabling] programmers to write thread-level parallel code for independent, scalar threads, as well as data-parallel code for coordinated threads".
- Source & further reading: Chapter 4 of PMPP, * = Sec. 4 of CUDA Programming Guide

Other Cores:

- **Tensor cores -** See pros explain here
- Ray-tracing cores, TMU, Media Engines / Video decoders & encoders, **do we care?**
- **Apple NPUs or Neural Engines -** Apple's tensor core equivalent. You'll have to dig for details.
  On the 12 CPU core M3 Max, has 16 NPUs, I believe intended only to be used through CoreML. 18 TOPS (I'm assuming 18 tera- (10^{12}) operations per second)
  Does anyone else care about making their iPads go brrr?

CPU memory hierarchy

- Registers
- L1 cache
- L2 cache
- L3, L4 …
- Main memory (but is off-chip)

GPU memory (general)

- Registers = **RMEM** = Register file (each processing block has one, so can be allocated to cores) - always 32bits in NVIDIA GPUs
- Shared Memory = **SMEM** = SRAM, physically the same place as L1 Data Cache
- Global Memory = **GMEM** = HBM = an abstraction of L2 Cache + VRAM, access times refer to time to access VRAM
- Local Memory = a portion of Global Memory, private to threads within a block

(sources for: _mem, SRAM/HBM, Pauleonix's note)

RAM = DRAM = Main Memory = CPU memory (but is off-chip)

Cache as a concept

- Saving data close-by so that you don't have to do either expensive recomputation or expensive fetching of the data, whether that fetching is over the internet, from RAM, or from Global Memory

PTX - One can think of the compilation & runtime process as **CUDA** -> **PTX** -> **SASS** (assembly for NVIDIA GPUs)

Don't know what assembly is? It's the lowest level human-readable language and is basically a mapping to machine code

PTX is an intermediate representation (a LLVM IR) that enables forward compatibility by JIT compiling PTX contained in fat-binaries (executables expanded with code native to multiple instruction sets) to SASS for newer architectures, at runtime.

GPU Cache Programmability - Cache Operators & Cache Eviction Priority Hints -

- Cache Operators & Cache Eviction Priority Hints are the tools we have to influence this
- Cache Operators on load/store instructions hint to the compiler how the cache should be used. Read about them here. In the examples linked, they use Load Cache Streaming, meaning the data is expected to be used once, which means load it into L1 and L2, but evict it first, to free up space in precious cache
    - CUDA C++ interface (documentation, example in llm.c)
    - Thrust & CUB documentation, example in llm.cpp
- Cache Eviction Priority Hints are used to modify persistence in cache. Read about here
- Looking at it through a programmer's lens, note "Cache operators on load or store instructions are treated as performance hints only. The use of a cache operator on an `ld` [load] or `st` [store] instruction does not change the memory consistency behavior of the program." (source)
  I think this means you can recommend it to the compiler but its probably going to do what it has to so that the compiled code is correct
- Don't worry too much about influencing cache or PTX in general until you're sure you need to. It's like, just use PyTorch until you need Triton, just use Triton until you need CUDA C++, just use higher-level CUDA until you really think you need to care about this stuff

NVIDIA GPU architecture generations (Grace Hopper, Blackwell, Hopper, Ampere, Volta, etc):
Architectures correspond to compute capabilities (Blackwell has compute capability 10.0, Ampere has 8.0, etc).

These versions enable new capabilities, like Ampere [and onward?] can make use of sparsity, Volta and onward can independently schedule threads of the same warp, etc.

- Table of architectures & GPU models (not comprehensive). Note that server class GPUs have the first letter of their architecture as a prefix (H100 -> Hopper), but not consumer class.
- Table including compute capabilities (not comprehensive)

Data center GPUs are the types you rent from a cloud vendor and send and receive work from (Or just use, if you are the cloud vendor). The turn-around-time/latency from sending and receiving work (in the simple case that there isn't intermediate results sent) will be *network latency + time the work takes on the GPU*.

Consumer GPUs are the types you either build into or come with a pre-built PC. These are marketed as GeForce RTX/GTX, the difference between RTX and GTX is that RTX have ray-tracing cores (specialized units for realistic graphics).
Ex. Ampere -> GeForce 30 series, and people often refer to for example, a GeForce RTX 3090, simply as a “3090”.

Benefits of consumer over data-center GPUs is that it’s the only real privacy guarantee you can have, latency is never network bound, you have the upfront cost of a card and whatever electricity costs, but never have to worry about renting GPUs or buying “Compute Units”, it’s reliably the same GPU (not always guaranteed by GPU/cloud providers, for example, A100s may not be available on Colab, or AWS doesn’t guarantee you which GPU you’re getting for the type of instance you select).

“Nvidia seems to roughly alternate between somewhat "experimental" architectures purely for the data center and ones that are used more generally (both for the data center and consumers).

- Pascal was both (GTX 10 series and P100),
- Volta was data center only (V100),
- Turing both but no to "T100" card to compete with V100 (RTX 20 series and T40),
- Ampere both (RTX 30 series and A100),
- Ada Lovelace both but no "L100" to compete with H100 (RTX 40 series and L40),
- Hopper data center only (H100),
- Blackwell both (RTX 50 series and B100)
- and future Rubin will probably be data center only as far as I have heard.

So Ampere is certainly not the only one although I guess Pascal is not very relevant anymore and Blackwell is not out yet.

"Professional" is probably not the best term for the "Tesla"/data center products as it is commonly used for the "Quadro"/workstation cards (although that name has vanished from the actual product names making things more confusing. See Turing's "Quadro RTX 6000", Ampere's "RTX A6000" and Ada Lovelace's "RTX 6000 Ada").” - Paul on Aug 31, 2024

Compute Intensity - CI = Arithmetic Intensity - AI, the Roofline model, and memory-bound vs compute-bound
Of a kernel,
CI = (Compute operations) / (Bytes accessed by gmem)
See Chapter 5.1 of PMPP

Speed of Light - SOL analysis
The analogy being that the Speed of Light is constant and the fastest thing possible, and everything else’s speed is relative to it.
Rather than using metrics like “it’s 100x faster than the previous implementation!”, if it’s only a small percent of how fast it could possibly go, it’s still not good.

SOL is how Nvidia internally reasons and communicates about speed.
It's found by first reasoning about whether an algorithm is compute or memory bound / it's placement on the Roofline model, and it is the percentage of *measured speed / theoretical SOL for this algorithm*.
Watch the pros explain here.

CUDA - Compute Unified Device Architecture
The CUDA (Compute Unified Device Architecture) platform is a software framework developed by NVIDIA to expand the capabilities of GPU acceleration. It allows developers to access the raw computing power of CUDA GPUs to process data faster than with traditional CPUs. (source)

CTA - Cooperative Thread Arrays - Thread blocks. (source)

cuBLAS - CUDA implementation of BLAS

CUTLASS - CUDA Templates for Linear Algebra Subroutines
"a collection of CUDA C++ template abstractions for implementing high-performance matrix-matrix multiplication (GEMM) and related computations at all levels and scales within CUDA." (source)

CuTe - ;)

- A core integral component of CUTLASS 3. CUTLASS 3 is a superset of CuTe.
- "By defining an appropriate **swizzling** function, *CuTe programmers* can access data the same way they would in the non-swizzling case, ***without worrying about bank conflicts***. CuTe abstracts away the swizzling details by baking in swizzle as a property of a tensor's layout using the composition operation."
  (from tensor core talk and Tutorial: Matrix Transpose in CUTLASS)
  (bank conflicts are detailed in the tutorial mentioned above, and in Ch 6.2 of PMPP)

CCCL - CUDA C++ Core Libraries = CUDA Core Compute Libraries
While raw (without CCCL or other libraries) implementations are excellent learning exercises, it is *recommended* by professionals that production code should always use abstractions from CCCL. Other professionals see it as a style choice.

CCCL is the joining of forces between
Thrust - Contains high-level and productive features
CUB - Contains low-level and more control features
libcudacxx - Contains features that span across this spectrum

NCCL (pronounced Nickel) - Nvidia Core Communications Library
Provides a way for GPUs to communicate quickly (watch the NCCL lecture!)

nvjet - "NVIDIA's most(?) optimised matmul kernels these days [H100], used by cuBLAS internally. You can look at the SASS in NSight Compute" - Aroun D

Swizzling - Tensor index and layout juggling in order to have the most efficient data access performance from different type of memory

Stream Disambiguation & Notes

- **STREAM benchmark: https://www.cs.virginia.edu/stream/ref.html**
  Community standard (microbenchmarking?) (exposed to me by: source)

- **Stream algorithm = chain scan = scan (an example being Prefix Sum)**
  "In general, if a computation is naturally described as a mathematical recursion in which each item in a series is defined in terms of the previous item, it can likely be parallelized as a parallel scan operation" **
- **A work efficient scan algorithm uses a Reduction**
- **Reduction**
  "A reduction derives a single value from an array of values. The single value could be the sum, the maximum value, the minimal value, and so on among all elements" *
  (resources: GPU Gems, Puzzle 12 of GPU Puzzles, wikipedia, * = Ch. 10, ** = Ch. 11 of PMPP, Advanced Scan)

Triton - OpenAI's python-like GPU programming language accessible through a python interface, and plays well with PyTorch. Using `torch.compile` actually compiles your PyTorch to Triton. Triton compiles to PTX (but you still need CUDA to fully leverage GPUs), and because of this, you can use **ncu** on Triton.

BLAS - Basic Linear Algebra Subprograms
"a specification that prescribes a set of low-level routines for performing common linear algebra operations such as vector addition, scalar multiplication, dot products, linear combinations, and matrix multiplication. They are the de facto standard low-level routines for linear algebra libraries"

Common Ops (some names from BLAS)
GEMM - General Matrix-Matrix Multiply
SGEMM - Single Precision (float32) General Matrix-Matrix Multiply
GEMV - General Matrix-Vector Multiply
FMA - Fused Multiply Add
MMA - Matrix Multiply Accumulate
WGMMA - Warpgroup MMA
WMMA - Warp [level] MMA (source)

TMA - Tensor Memory Accelerator - unit that can transfer large blocks of data efficiently between global memory and shared memory. TMA also supports asynchronous copies between thread blocks in a cluster (source)

Common commands

nvcc - Compile CUDA C++. It's usage is like **gcc**, a common C++ compiler.

nvcc compiles only the device code in a .cu file while the host code is forwarded to some host compiler from outside like gcc, clang or MSVC on Windows. This is why nvcc is sometimes called a compiler wrapper. (There's also the nvc++ & nvc compilers available through the NVHPC Toolkit that do both host and device code.)

```
nvcc -o my_kernel my_program.cu its_utilities.cu
./my_kernel
```

ncu - Nsight Compute, The Profiler (regarding profiling and system monitoring, there is also the PyTorch Profiler. Here's a profiling lecture on usage.)

```
ncu my_kernel
```

nsys - Nsight Systems - It takes system timelines/traces and is great for looking for bubbles in the GPU timeline, i.e. optimizing the bigger picture of many kernels on multiple GPUs and even nodes vs a single kernel with ncu.

```
nsys [global_option]
```
User Guide

nvidia-smi - Check utilization & stats about your GPUs

```
nvidia-smi
```

nvtop - GPU utilization monitoring. It's usage is like ~~**top**~~ **bpytop**

```
nvtop
```

Terminology index (Source: Computer Architecture: A Quantitative Approach, Hennesy & Pattersonm, 5th edition) :

Type	More descriptive name used in this book	Official CUDA/ NVIDIA term	Book definition and AMD and OpenCL terms	Official CUDA/NVIDIA definition
Program abstractions	Vectorizable loop	Grid	A vectorizable loop, executed on the GPU, made up of one or more "Thread Blocks" (or bodies of vectorized loop) that can execute in parallel. OpenCL name is "index range." AMD name is "NDRange".	A grid is an array of thread blocks that can execute concurrently, sequentially, or a mixture.
	Body of Vectorized loop	Thread Block	A vectorized loop executed on a multithreaded SIMD Processor, made up of one or more threads of SIMD instructions. These SIMD Threads can communicate via Local Memory. AMD and OpenCL name is "work group".	A thread block is an array of CUDA Threads that execute concurrently together and can cooperate and communicate via Shared Memory and barrier synchronization. A Thread Block has a Thread Block ID within its Grid.
	Sequence of SIMD Lane operations	CUDA Thread	A vertical cut of a thread of SIMD instructions corresponding to one element executed by one SIMD Lane. Result is stored depending on mask. AMD and OpenCL call a CUDA Thread a "work item."	A CUDA Thread is a lightweight thread that executes a sequential program and can cooperate with other CUDA Threads executing in the same Thread Block. A CUDA Thread has a thread ID within its Thread Block.
Machine object	A Thread of SIMD instructions	Warp	A traditional thread, but it contains just SIMD instructions that are executed on a multithreaded SIMD Processor. Results are stored depending on a per-element mask. AMD name is "wavefront."	A warp is a set of parallel CUDA Threads (e.g., 32) that execute the same instruction together in a multithreaded SIMT/SIMD Processor.
	SIMD instruction	PTX instruction	A single SIMD instruction executed across the SIMD Lanes. AMD name is "AMDIL" or "FSAIL" instruction.	A PTX instruction specifies an instruction executed by a CUDA Thread.

Figure 4.24 Conversion from terms used in this chapter to official NVIDIA/CUDA and AMD jargon. OpenCL names are given in the book definition.

Type	More descriptive name used in this book	Official CUDA/ NVIDIA term	Book definition and AMD and OpenCL terms	Official CUDA/NVIDIA definition
Processing hardware	Multithreaded SIMD processor	Streaming multi-processor	Multithreaded SIMD Processor that executes thread of SIMD instructions, independent of other SIMD Processors. Both AMD and OpenCL call it a "compute unit." However, the CUDA Programmer writes program for one lane rather than for a "vector" of multiple SIMD Lanes.	A streaming multiprocessor (SM) is a multithreaded SIMT/ SIMD Processor that executes warps of CUDA Threads. A SIMT program specifies the execution of one CUDA Thread, rather than a vector of multiple SIMD Lanes.
	Thread block scheduler	Giga thread engine	Assigns multiple bodies of vectorized loop to multithreaded SIMD Processors. AMD name is "Ultra-Threaded Dispatch Engine".	Distributes and schedules thread blocks of a grid to streaming multiprocessors as resources become available.
	SIMD Thread scheduler	Warp scheduler	Hardware unit that schedules and issues threads of SIMD instructions when they are ready to execute; includes a scoreboard to track SIMD Thread execution. AMD name is "Work Group Scheduler".	A warp scheduler in a streaming multiprocessor schedules warps for execution when their next instruction is ready to execute.
	SIMD Lane	Thread processor	Hardware SIMD Lane that executes the operations in a thread of SIMD instructions on a single element. Results are stored depending on mask. OpenCL calls it a "processing element." AMD name is also "SIMD Lane".	A thread processor is a datapath and register file portion of a streaming multiprocessor that executes operations for one or more lanes of a warp.
Memory hardware	GPU Memory	Global Memory	DRAM memory accessible by all multithreaded SIMD Processors in a GPU. OpenCL calls it "Global Memory."	Global memory is accessible by all CUDA Threads in any thread block in any grid; implemented as a region of DRAM, and may be cached.
	Private Memory	Local Memory	Portion of DRAM memory private to each SIMD Lane. Both AMD and OpenCL call it "Private Memory."	Private "thread-local" memory for a CUDA Thread; implemented as a cached region of DRAM.
	Local Memory	Shared Memory	Fast local SRAM for one multithreaded SIMD Processor, unavailable to other SIMD Processors. OpenCL calls it "Local Memory." AMD calls it "Group Memory".	Fast SRAM memory shared by the CUDA Threads composing a thread block, and private to that thread block. Used for communication among CUDA Threads in a thread block at barrier synchronization points.
	SIMD Lane registers	Registers	Registers in a single SIMD Lane allocated across body of vectorized loop. AMD also calls them "Registers".	Private registers for a CUDA Thread; implemented as multithreaded register file for certain lanes of several warps for each thread processor.

Figure 4.25 Conversion from terms used in this chapter to official NVIDIA/CUDA and AMD jargon. Note that our descriptive terms "Local Memory" and "Private Memory" use the OpenCL terminology. NVIDIA uses SIMT, single-instruction multiple-thread, rather than SIMD, to describe a streaming multiprocessor. SIMT is preferred over SIMD because the per-thread branching and control flow are unlike any SIMD machine.

Special thanks to Paul (Pauleonix) for his corrections and knowledge contributed to this doc 🙂