# Low Level Ecosystem: CUDA, HIP, OpenCL

**Abhishek Bagusetty**
*Performance Engineering Team*
*Argonne Leadership Computing Facility*

# Agenda

- GP-GPU Execution & Memory Model
- CUDA
- HIP
- Tensor/Matrix Cores
- OpenCL

# GP-GPU Execution Model

**Host-Device Separation:** Execution alternates between the Host (CPU) with sequential code and the Device (GPU) running massively parallel code

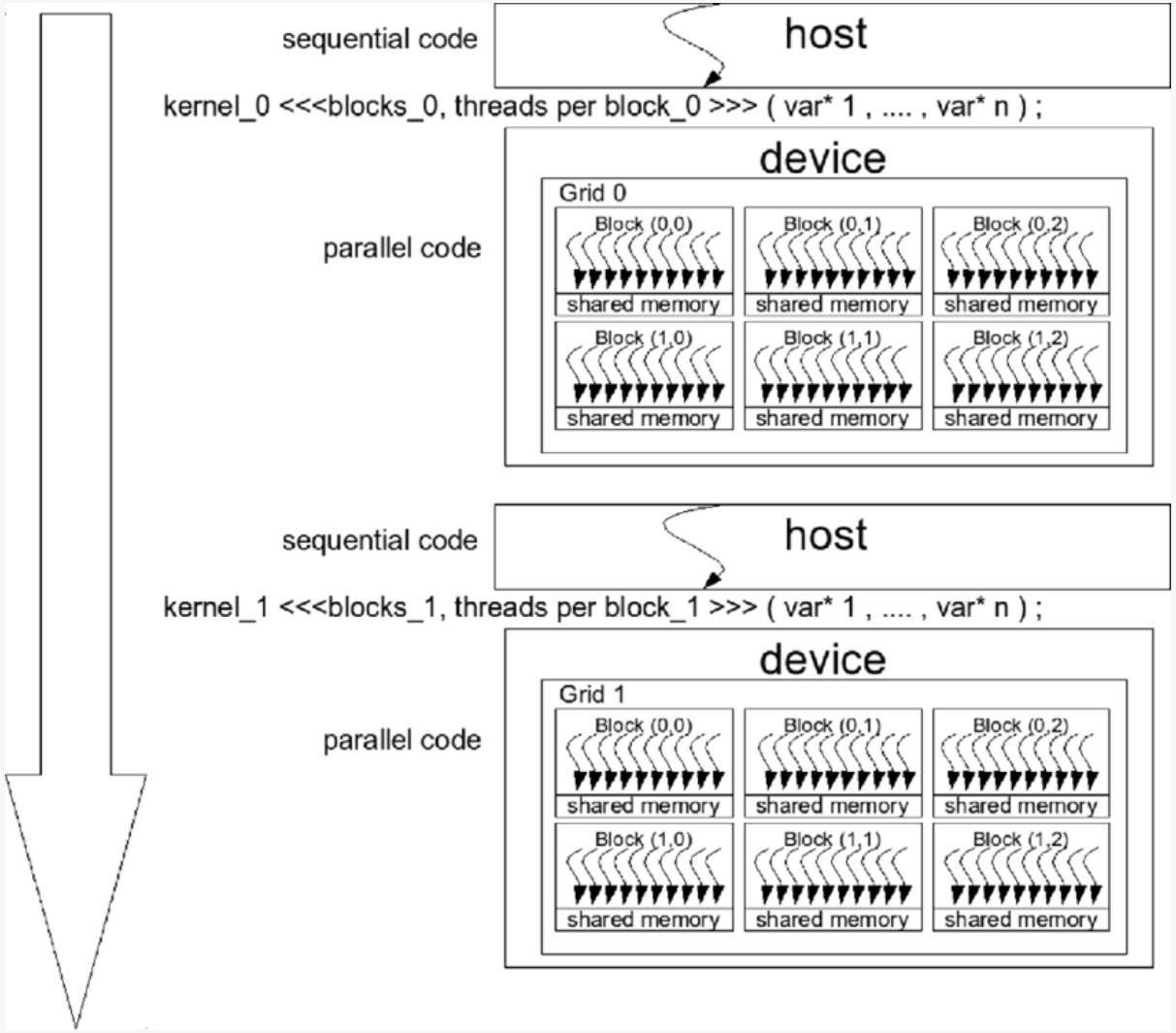
**The Kernel Launch:** The Host offloads heavy parallel tasks to the Device by launching a "kernel" (e.g., kernel_0 <<<...>>>)

- The Host specifies the dimensions of the parallel workers and passes pointers/variables.

**Grid and Block Hierarchy:**

- **Grid:** A single kernel launch creates a "Grid" encompassing all the parallel work
- **Blocks:** The Grid is divided into independent "Blocks." Blocks can be scheduled and executed in any order by the hardware

**Thread Execution:** Each Block contains a collection of individual "Threads" (the wavy arrows). All threads in a block execute the exact same kernel instructions concurrently but operate on different data indices




https://doi.org/10.1080/10407790.2010.541359

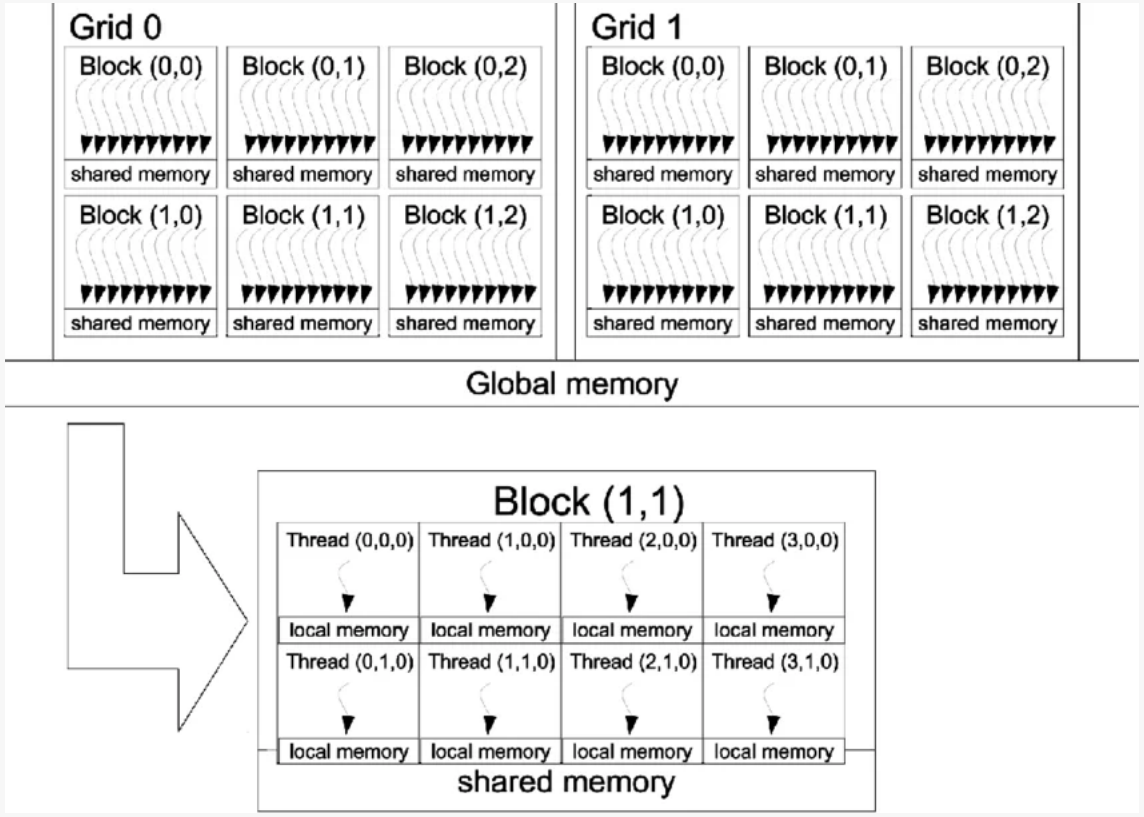
# GP-GPU Memory Model

**Global Memory (Device-Wide Scope):**
- Accessible by *all* threads across all blocks and grids
- Large capacity, but high latency
- Used for initial data input, final output and sharing data between different blocks

**Shared Memory (Block-Wide Scope):**
- Accessible only by threads *within the same specific block*
- Fast, on-chip memory

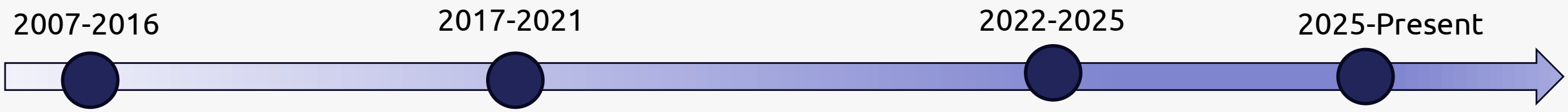
**Local Memory (Thread-Private Scope):**
- Accessible *only* by the individual thread that owns it
- Used by the compiler to store thread-private variables (like arrays or large structures) that do not fit into the thread's hardware registers

**Thread Hierarchy Mapping:** The physical execution maps to this memory structure.
- Grids contain blocks (which share Global Memory)
  - Blocks contain threads (which share Shared Memory)
    - Individual Threads with own private Local Memory




https://doi.org/10.1080/10407790.2010.541359

# CUDA – Evolution of Ecosystem




*Hardware:* Tesla through Pascal architectures (CUDA 1 – 8)

*Paradigm:* Single Instruction, Multiple Threads (SIMT). Developers wrote code from the perspective of a single, scalar thread

*Memory:* Moving data meant calculating 1D/2D offsets per thread, issuing individual load instructions, and manually synchronizing via __syncthreads()

*Hardware:* Volta and Ampere architectures (CUDA 9 – 11)

*Paradigm:* Hardware Tensor Cores demanded that threads work together to multiply small matrices

*Features:* NVIDIA introduced **Cooperative Groups** and **WMMA** (Warp Matrix Multiply-Accumulate), allowing developers to explicitly program at the "Warp" level rather than the thread level.

*Hardware:* Hopper architecture (CUDA 12)

*Paradigm:* Computation became so fast that the CPU and global memory couldn't keep up. Memory movement had to be decoupled from compute

*Features:* Introduction of the **Tensor Memory Accelerator (TMA)** and **Asynchronous Barriers**. A single thread can now fire off bulk memory transfers in the background while the rest of the block computes.

*Hardware:* Blackwell and Rubin architectures (CUDA 13+)

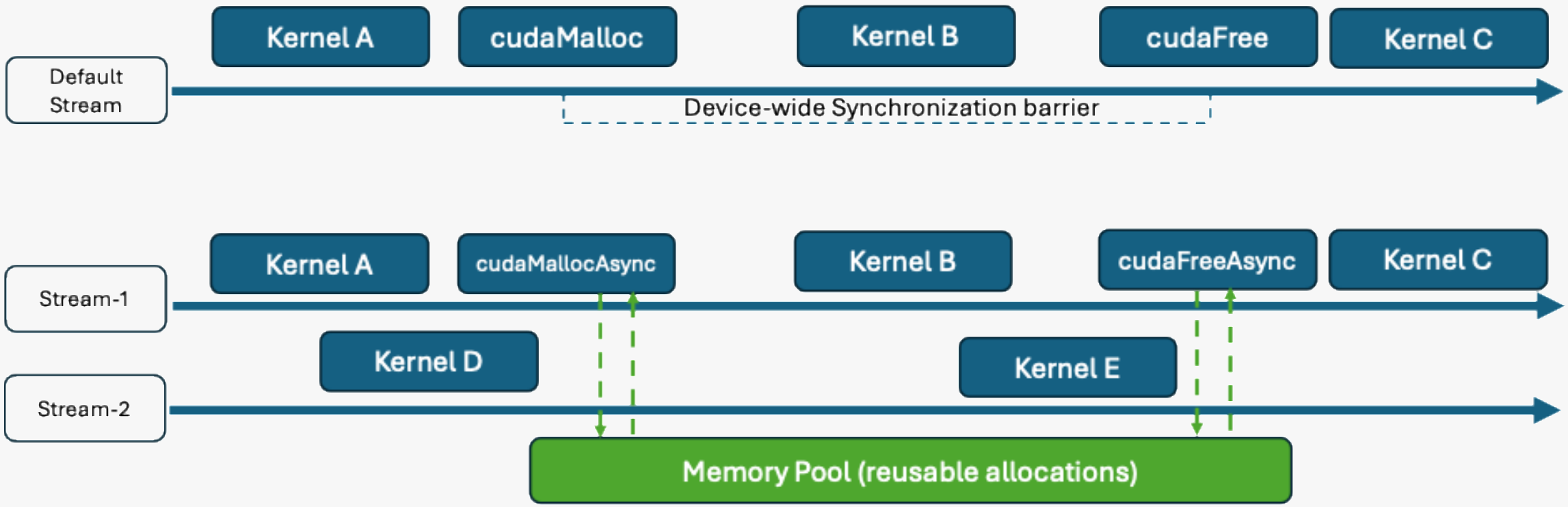
*Paradigm:* Abstracting away thread management entirely to focus on data geometry at the block level

*Features:* **cuTile** and **CuTe** allow developers to write Pythonic or C++ array operations. The compiler automatically handles TMA orchestration, thread-mapping, and mapping data directly to Tensor Cores.

# CUDA – Stream ordered Allocations

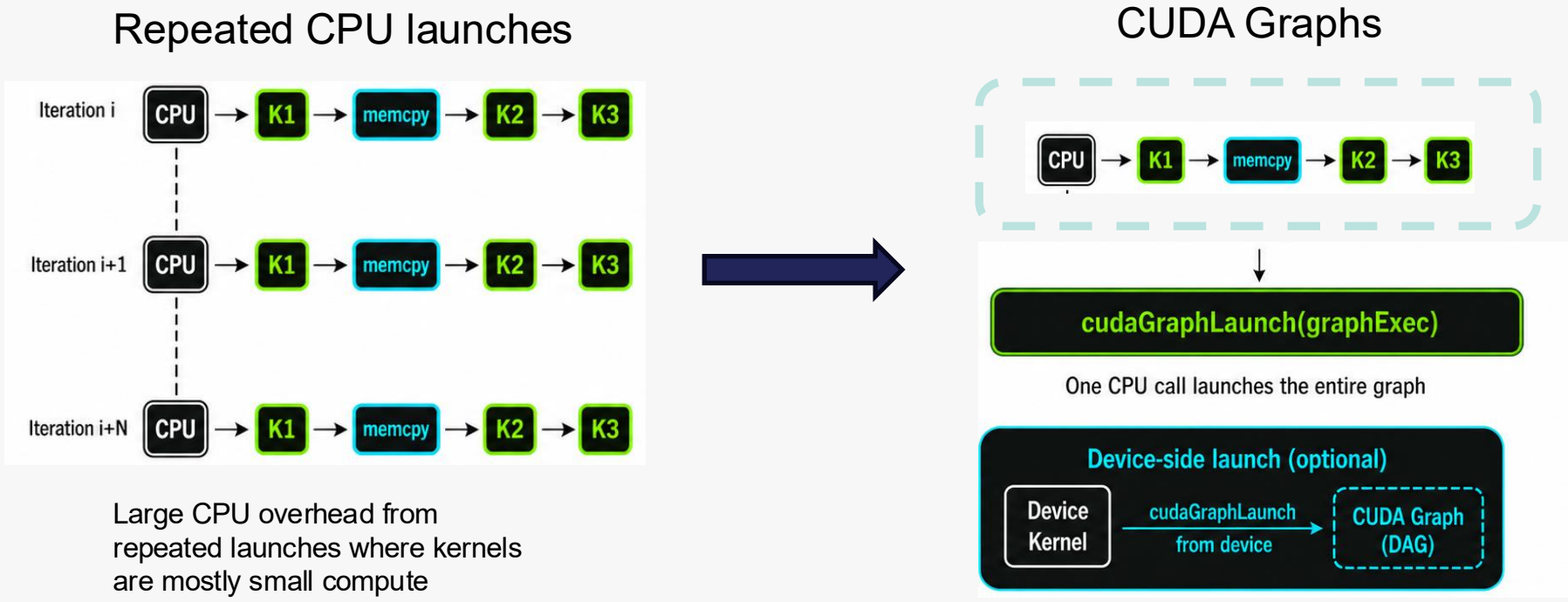
- `cudaMalloc/cudaFree` are implicit device-wide synchronization points (runs on default stream)
- Using Stream-ordered management, we can use a memory-pool for (de)allocations
- Memory Pool: A pre-allocated chunk of memory from which smaller allocations are served

- Usage: Dynamic work queues or adaptive mesh refinement where per-step buffer sizes change but you want allocation overhead amortized across the run



# CUDA – Graphs

- CUDA Graphs : Using CUDA Graphs can reduce overheads associated with launching multiple GPU operations by allowing them to be launched through a single CPU operation
- CUDA Graphs support multiple interacting streams, including kernel executions, memory copies, and functions executing on the host CPUs, and can span multiple GPUs, providing more opportunities for optimization




Large CPU overhead from repeated launches where kernels are mostly small compute

Source: https://pytorch.org/blog/accelerating-pytorch-with-cuda-graphs/

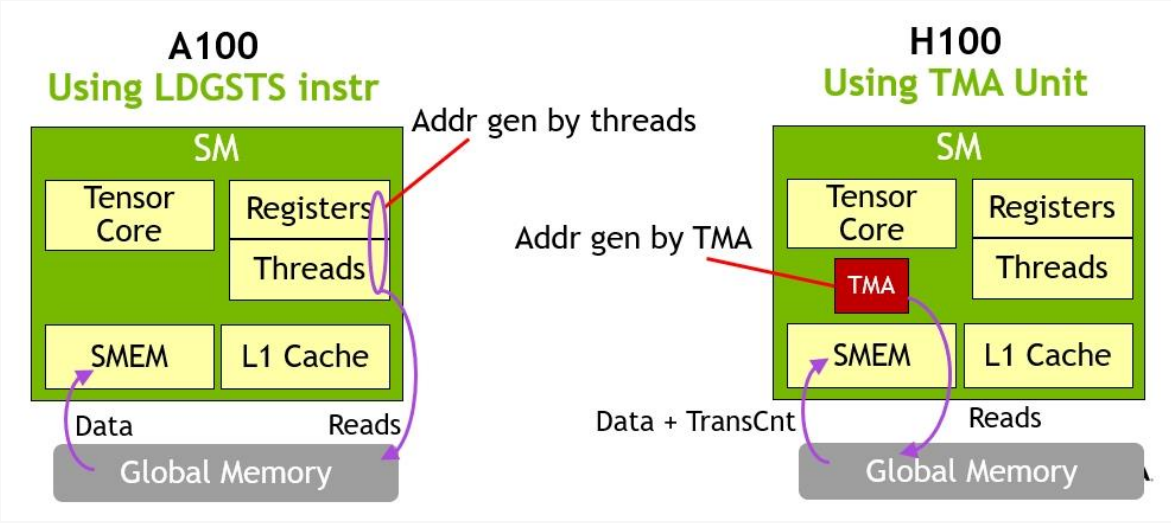
# CUDA – Tensor Memory Architectures (TMA)

- Introduced in Hopper (SM90), TMA is a dedicated hardware engine that offloads multi-dimensional data movement. It replaces manual, per-thread memory fetching with a single asynchronous hardware instruction

- Motivation: Pre-Hopper generation, every thread in a block wasted registers and compute cycles calculating multidimensional pointer offsets and manually loading matrix tiles from global memory

**The TMA Solution**

- A **single thread** can now issue a bulk transfer of 1D–5D tensors directly between Global and Shared Memory
- The hardware completely takes over, automatically handling all stride calculations, bounds checking, and shared memory swizzling




**The 3-Step Execution Flow**

1. **Host Setup:** The CPU creates a `TmaDescriptor` defining the tensor's shape/strides and passes it to the
2. **Kernel Launch:** One thread uses the `cp.async.bulk.tensor` instruction to trigger the hardware engine
3. **Compute Overlap:** All threads immediately move on to math operations while TMA moves data in the background

# CUDA – Tensor Memory Architectures (TMA)

*every* thread calculates its own unique 2D offset in global memory using blockIdx and threadIdx

Allocate shared-memory of 2D dim. [64][64] (per-block: All threads in a block will share this memory)

`(g_row * pitch + g_col)` is gone `CUtensorMap` object configured the shape, strides, and memory layout of the entire tensor on the CPU now

```
__global__ void preHopperKernel(float* global_in, int pitch) {
    __shared__ float shared_tile[64][64];

    // 1. EVERY thread computes its own 2D global memory offset
    int g_row = blockIdx.y * 64 + threadIdx.y;
    int g_col = blockIdx.x * 64 + threadIdx.x;

    // 2. EVERY thread issues a load instruction
    shared_tile[threadIdx.y][threadIdx.x] = global_in[g_row * pitch + g_col];

    // 3. Barrier: Wait for all threads to finish their individual loads
    __syncthreads();

    // ... begin math operations ...
}
```

```
#include <cuda/barrier>

__global__ void tmaKernel(const CUtensorMap* tensor_descriptor) {
    __shared__ float shared_tile[64][64];

    // TMA relies on CUDA's asynchronous transaction barriers
    __shared__ cuda::barrier<cuda::thread_scope_block> tma_barrier;

    // 1. A SINGLE thread takes charge of the memory movement
    if (threadIdx.x == 0 && threadIdx.y == 0) {
        init(&tma_barrier, blockDim.x * blockDim.y);

        // 2. Issue the TMA bulk copy.
        // The hardware engine natively handles strides and shared-memory
swizzling.
        cuda::memcpy_async(shared_tile, tensor_descriptor, tma_barrier,
blockIdx.x, blockIdx.y);
    }

    // 3. ALL threads wait for the TMA hardware engine to signal the barrier
    tma_barrier.arrive_and_wait();

    // ... begin math operations ...
}
```

_syncthreads(): All the threads in the block wait until the copy is finished

`arrive_and_wait()` Threads now are not idle waiting, they are free to compute. But they wait until TMA engine signals

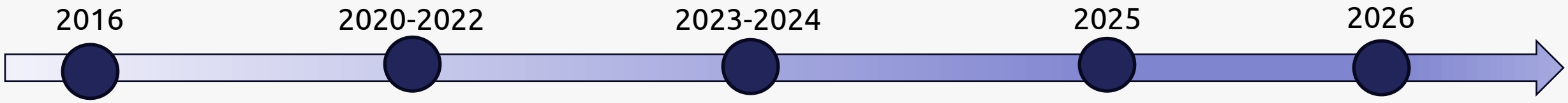
**single thread** `(threadIdx.x == 0 && threadIdx.y == 0)` takes charge. It uses `cuda::memcpy_async` to hand the descriptor to the TMA hardware engine.

# HIP – Heterogeneous-Compute Interface for Portability

HIP is a C++ dialect designed to be nearly syntactically identical to CUDA, allowing developers to maintain a single codebase that compiles to either AMD or NVIDIA GPUs

- **The Goal:** Write code once, compile it for AMD (via `hipcc` / LLVM) or NVIDIA (via `nvcc`)
- **The Mechanism:** HIP is largely a set of headers and macros. `cudaMalloc` becomes `hipMalloc`, `cudaMemcpy` becomes `hipMemcpy`
- **Kernel Launch Syntax:** HIP supports standard CUDA syntax `<<<grid, block>>>`, but also provides `hipLaunchKernelGGL()` to maintain compatibility with standard C++ parsers
- **Reality Check:** Tools like `hipify-clang` or `HIPIFY` script will automatically translate 90% of a CUDA codebase via regex/AST parsing. The remaining 10% requires manual engineering because *syntax* portability does not guarantee *performance* portability

# HIP – Evolution of Ecosystem




ROCm 1.0

**The Origin (ROCm 1.0)**
**The Problem:** AMD needed a bridge for developers locked into CUDA

**The Solution:** The Heterogeneous-compute Interface for Portability (HIP) was born, alongside the hipify tool, to mechanically translate CUDA syntax to a common C++ API that could compile for both AMD and NVIDIA hardware

Exascale Era (ROCm 4.x & 5.x)

ROCm moves from graphics-focused architectures to the CDNA compute architecture (MI100/MI250)

**The Milestones:** Fortran support is solidified, and libraries like rocBLAS and rocFFT achieve parity with cuBLAS/cuFFT to support the Frontier supercomputer (the first exascale system)

AI Pivot (ROCm 6.x)

Focus shifts heavily toward generative AI, PyTorch, and large language models (LLMs) on the MI300X

**The Milestones:** Drop-in support for vLLM, FlashAttention, and native integration into mainline PyTorch. The ecosystem prioritizes Python/Triton workflows over raw C++ kernels.

Modular & Multi-GPU Era (ROCm 6.4 & 7.x)

**ROCm 6.4:** Introduces containerized, pre-optimized environments for LLM routing and Kubernetes integrations. Introduces modular GPU drivers

**ROCm 7.x (HIP 7.0):** HIP 7.0 deliberately aligns its C++ semantics even closer to modern CUDA, reducing the friction of maintaining portable codebases. Full support for FP6/FP4 on the MI350/CDNA 4 architecture

AI-Assisted Era (ROCm 7.14+ & ROCm.AI)

Pushing beyond manual kernel tuning

**The Milestones:** Introduction of the CDNA 5 "Wave32" shift and **ROCm.AI (Hyperloom)**, integrating LLM-assisted toolchains that automatically analyze and optimize memory schedules and kernel configurations.

# HIP – Execution & Memory Model

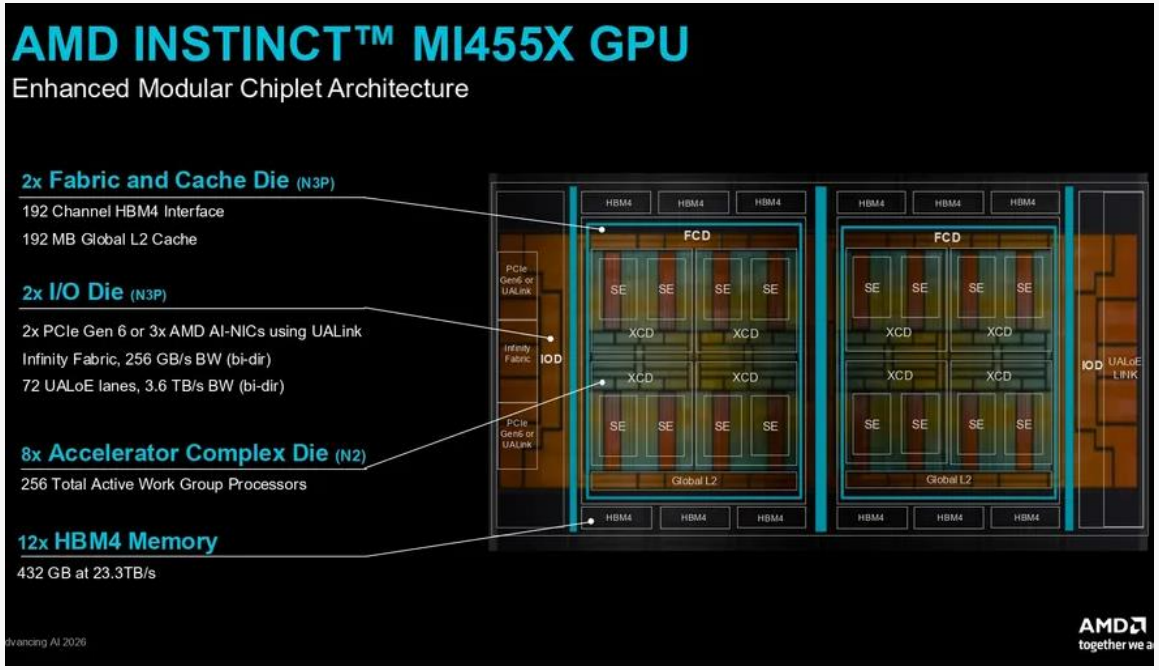
HIP is a C++ dialect designed to be nearly syntactically identical to CUDA

| Concept | NVIDIA / CUDA | AMD / HIP | Key Difference / Implication |
|---|---|---|---|
| **Lockstep Group** | Warp (32 threads) | Wavefront (32 or 64) | Hardcoding "32" loop bounds will break logic on AMD. Always use `__AMDGCN_WAVEFRONT_SIZE` |
| **Compute Core** | Streaming Multiprocessor (SM) | Compute Unit (CU) | - |
| **Execution Group** | Thread Block | Workgroup | - |
| **Shared Scratchpad** | Shared Memory | LDS (Local Data Share) | Functions the same, but bank conflict avoidance strategies can differ |
| **Registers** | Registers | VGPRs and SGPRs | AMD splits registers into Vector (unique) and Scalar (uniform) |
| **Inline Assembly** | PTX | AMDGCN | Inline PTX fails to compile on HIP. Needs rewrite to AMDGCN or cross-platform intrinsics |

# HIP – Hardware Trajectory MI455X

AMD's late-2026 CDNA 5 architecture drastically changes the programming model, similarly related to NVIDIA's Hopper/Blackwell paradigms

- **The Shift to Wave32:** CDNA 5 drops the traditional 64-wide wavefront in favor of a 32-wide wavefront (Wave32). This matches CUDA’s warp size, massively easing the porting of AI/HPC kernels and reducing VGPR register pressure
- **Tensor Data Mover (TDM):** Introduces hardware for asynchronous, direct-to-LDS (shared memory) data movement. This is AMD's direct answer to NVIDIA's TMA, allowing developers to overlap memory fetches with matrix math
- **Helios & HBM4:** AMD moves the bottleneck from the chip to the rack, featuring 432GB of HBM4 per GPU and the Helios rack-scale architecture (a 72-GPU coherent memory domain) to combat NVL72



# HIP – Modern ROCm Ecosystem

| CUDA | AMD / ROCm | 2026 Status & HPC/ML Relevance |
|---|---|---|
| **cuBLAS / cuBLASLt** | **hipBLAS / hipBLASLt** | Standard dense BLAS. hipBLASLt is critical for accessing exact Matrix Core (MFMA) heuristics and epilogue fusions |
| **CUTLASS / CuTe** | **Composable Kernel (CK)** | Templated C++ primitives for fused matrix/tensor ops. Highly recommended for custom HPC kernels that need to saturate MFMA engines |
| **WMMA API** | **rocWMMA** | C++ API for Matrix Cores. Easier than raw intrinsics, but abstracting Wave64 vs Wave32 fragmentation requires care |
| **NCCL** | **RCCL** | Multi-GPU collectives. Recently upgraded to handle Helios rack-scale topologies and custom Infiniband/RoCE routing. |
| **Triton (CUDA Backend)** | **Triton (AMD Backend)** | *Highly active development.* Compiles Python DSL directly to optimized CDNA binaries without touching HIP C++. Fast becoming the default for custom ML ops |
| **cuFFT / cuSPARSE** | **rocFFT / rocSPARSE** | Mature, highly optimized drop-in replacements for standard scientific computing domains |

# HIP – AMD Instinct Performance

AMD's architecture has rapidly shifted from prioritizing dense FP64 HPC calculations to unlocking massive throughput in lower-precision AI data types, supported by an explosion in memory bandwidth.

| Accelerator | Architecture (Release) | HPC (FP64 Matrix) | AI Training (FP16 / BF16) | AI Inference (FP8 / FP6) | Extreme Low-Precision | Memory & Bandwidth |
|---|---|---|---|---|---|---|
| **MI250X** | CDNA 2 (2021) | 95.7 TFLOPS | 383 TFLOPS | N/A | N/A | 128 GB HBM2e (3.2 TB/s) |
| **MI300X** | CDNA 3 (2023) | 163.4 TFLOPS | 1.3 PFLOPS | 2.6 PFLOPS (FP8) | N/A | 192 GB HBM3 (5.3 TB/s) |
| **MI350X** | CDNA 4 (2025) | 72.1 TFLOPS* | 2.3 PFLOPS | 4.6 PFLOPS (FP8) 4.6 PFLOPS (FP6) | 9.2 PFLOPS (FP4) | 288 GB HBM3E (8.0 TB/s) |
| **MI455X** | CDNA 5 (2026) | N/A** | 5.0 PFLOPS | 10.1 PFLOPS (FP8)** | 20.0 PFLOPS (FP4)** | 432 GB HBM4 (23.3 TB/s) |

**Note: MI350X/MI455X reduce peak dense FP64 to prioritize die space for massive FP8/FP4 Matrix Cores. For dedicated HPC FP64 workloads, AMD offers the specialized MI430X variant (up to 288 TFLOPS FP64).*
***MI455X numbers shown without sparsity. With sparsity, AI flops double (e.g., up to 40 PFLOPS FP4).*

# HIP on Nvidia

HIP is not an "AMD-only" language; it is a cross-platform portability layer. By using HIP, you can maintain one codebase that compiles to native, zero-overhead machine code for both AMD and NVIDIA GPUs

- Maintaining separate codebases for heterogeneous fleets (e.g., AMD-based Frontier vs. NVIDIA-based Perlmutter) destroys developer productivity
- The *hipcc* compiler driver is just a smart wrapper. Set **HIP_PLATFORM=nvidia**, and it routes your code directly to NVIDIA's *nvcc* compiler
- HIP on NVIDIA is not an emulator or a runtime layer. Headers simply map HIP APIs back to CUDA APIs (e.g., *hipMalloc* to *cudaMalloc*), generating identical PTX and SASS assembly
- CMake Integration: Modern CMake natively handles HIP compilation targets, allowing you to swap backends via build flags without altering your application logic
- Catch: While the core language and memory management translate 1:1, vendor-tuned math libraries (like *hipBLAS* vs. *cuBLAS*) still require C++ wrappers or abstraction frameworks like Kokkos or RAJA

# Modern Unified Platforms

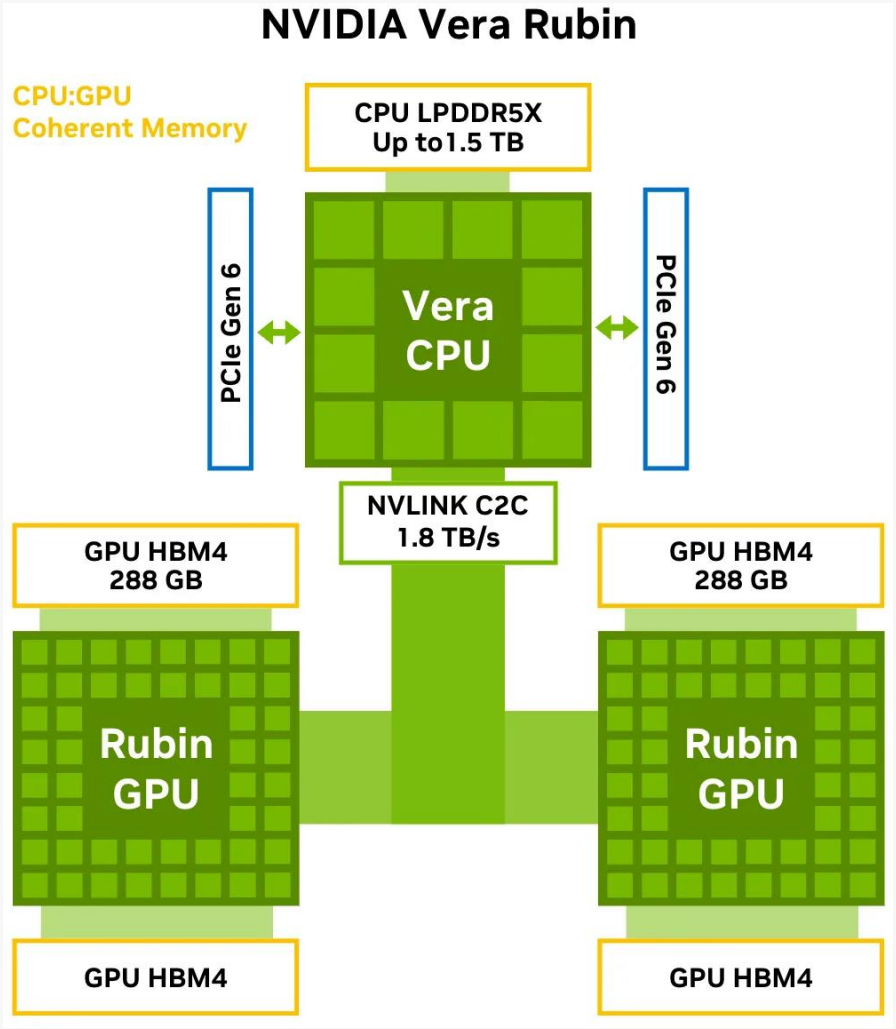
- NVIDIA calls a 'Superchip' and AMD calls an 'APU'

- "Vera" CPU and the "Rubin" GPUs are physically mounted on the exact same board
- Same is the case with AMD MI300A of LLNL El Capitan

- **Connectivity**: They are *not* using PCIe. They are connected via NVLink-C2C, which transfers data at 1.8 Terabytes per second. That is roughly 7 to 10 times faster than a standard PCIe Gen 5 or Gen 6 connection

- **Memory Coherency**: The CPU's massive 1.5 TB of LPDDR5X memory and the GPU's ultra-fast HBM4 memory act as one giant, unified pool

- No need to write cudaMemcpy/hipMemcpy
  - The GPU can reach directly into the CPU's memory space and compute on the data instantly
  - This fundamentally changes how we write applications for machines like the upcoming Vera Rubin systems or AMD's MI300A

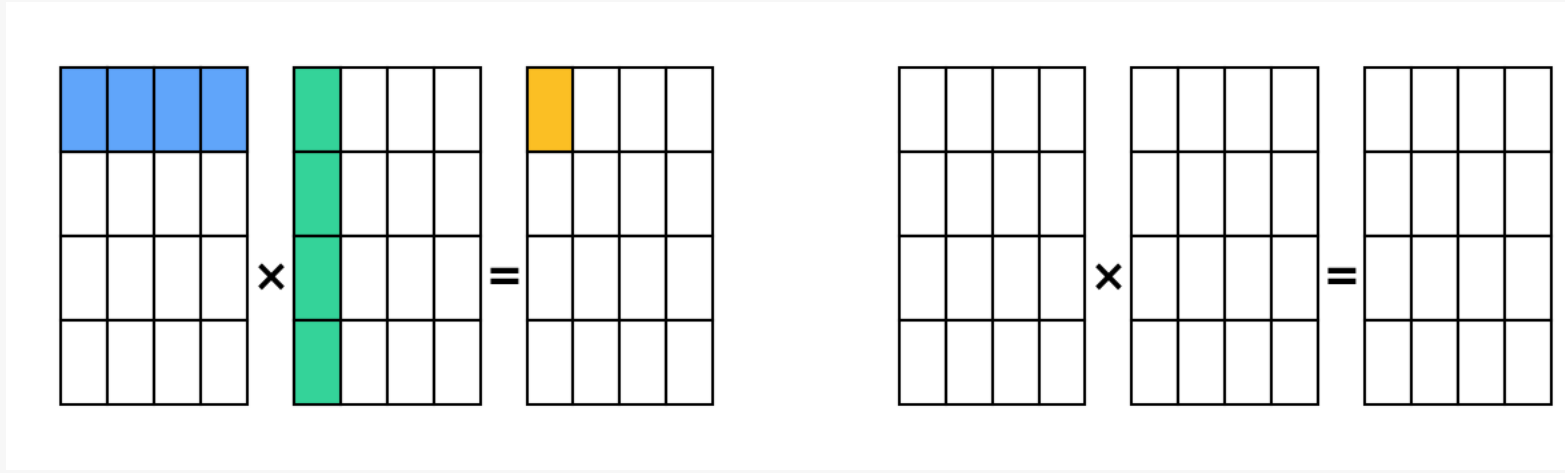


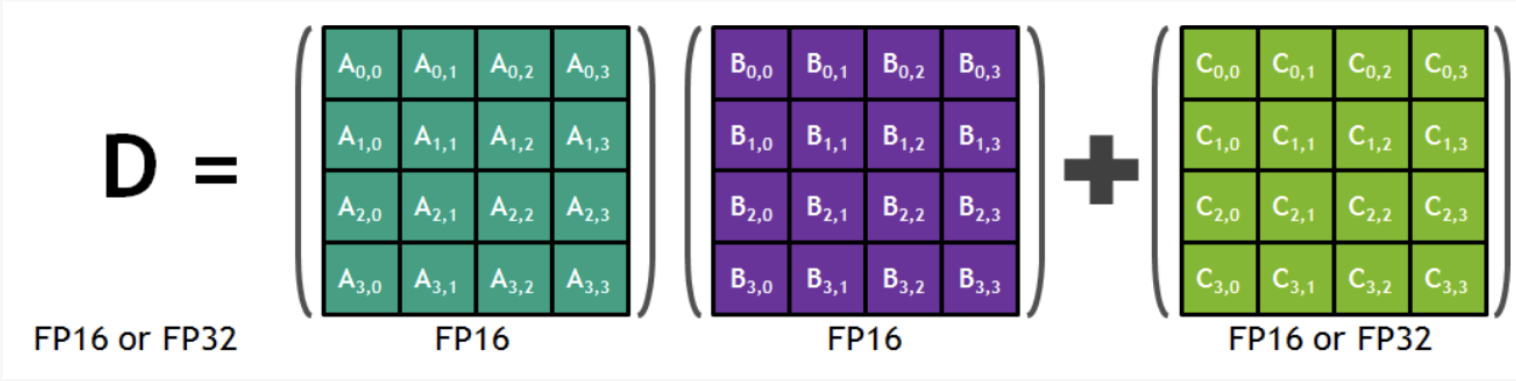
# Tensor/Matrix Cores – Introduction

- "Tensor Cores" (NVIDIA) and "Matrix Cores" (AMD) are just vendor terms for the exact same concept
- Dedicated silicon built entirely for dense matrix math
- Because they perform hundreds of operations per clock, these engines now hold **>90% of the peak FLOPs** on modern GPUs
- Usage: You can't reach this speed with simple nested loops. You must rethink your code around asynchronous data pipelines and precise memory layouts to keep these massive engines fed.

**CUDA Cores (Scalar/SIMT)**
**(computes elements sequentially)**

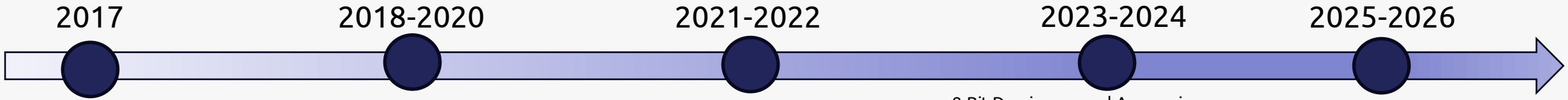
**Tensor/Matrix Cores (Matrix)**
**(computes a 4x4 block in cycle)**






First Tensor Core Support in Volta Architecture (V100): Tensor Core 4x4 Matrix Multiply and Accumulate

# Tensor/Matrix Cores – Evolution of Specialized Hardware




### 2017 — First Matrix Engine Hardware

**NVIDIA Volta (V100)**: Introduced the 1st generation of **Tensor Cores**

- **Capability**: Hardware matrix multiply-accumulate (HMMA) performing 4×4×4 matrix operations in a single clock cycle
- **Precision**: FP16 inputs with FP32 accumulation
- **Impact**: Primarily drove deep learning training

### 2018-2020 — Broadening the Precision Spectrum

**NVIDIA Turing (T4/RTX 20-series)** : 2nd generation Tensor Cores added integer support (INT8, INT4) to accelerate AI inference workloads

**NVIDIA Ampere (A100)** : 3rd generation Tensor Cores revolutionized the space by making matrix engines highly relevant to HPC, not just AI

**New Formats**: Introduced **TF32** (Tensor Float FP32 replacement for AI), **BF16**, and native **FP64 Tensor Cores** (doubling double-precision matrix math for traditional HPC)

**AMD CDNA 1 (MI100)** : **Matrix Cores** via the MFMA (Matrix Fused Multiply-Add) instruction set

### 2021-2022 — The Shift to 8-Bit and Transformer Optimization

**AMD CDNA 2 (MI250X):** Introduced packed FP64 matrix operations, traditional HPC workloads (Frontier, OLCF)

**NVIDIA Hopper (H100)**: 4th generation Tensor Cores

- **FP8 Support**: Native 8-bit floating-point math
- **Transformer Engine**: Hardware-level logic to dynamically balance between 8-bit and 16-bit
- **The Async Shift**: Replaced synchronous warp-level instructions with **WGMMA** (Warpgroup MMA), allowing 128 threads to asynchronously issue matrix math with the TMA (Tensor Memory Accelerator).

### 2023-2024 — 8-Bit Dominance and Aggressive Low Precision

**AMD CDNA 3 (MI300X):** Added native FP8 support and sparsity features to AMD's Matrix Cores

**NVIDIA Blackwell (B100/B200)**: 5th generation Tensor Cores pushed precision limits further down

- **FP4 & FP6**: Added support for 4-bit and 6-bit floating-point math to address the memory bandwidth and compute bottlenecks of massive LLMs
- **Microscaling (MX) Formats**: Implemented block-scaled formats (MXFP8/6/4) to maintain accuracy at ultra-low precisions
- **tcgen05**: Introduced Tensor Memory and coordinated CTA-pair cooperation

### 2025-2026 — The Extreme Low-Precision Era

**AMD CDNA 4 (MI355X):** Expanded the MFMA instruction set to natively support FP6 and FP4 formats, massively increasing the low-precision matrix rate to compete in next-gen LLM inference

**NVIDIA Rubin (R100):** Features an enhanced 5th generation Tensor Core specifically optimizing **NVFP4** (a proprietary 4-bit narrow-precision arithmetic). Deeply integrates special function units to accelerate attention mechanisms and sparse compute paths

# Tensor/Matrix Cores – Rules of Engagement

- **The Golden Rule:** Compute in the lowest precision possible, but accumulate in higher precision. (e.g., multiply in FP8, accumulate the sum in FP32)
- **For LLMs:** The bleeding edge is now 4-bit and 6-bit (FP4/FP6) and Microscaling (MX) formats on Blackwell and CDNA 4/5. Users must manage quantization noise and block-level scaling factors dynamically
- **For HPC:** TF32 (Tensor Float 32) and BF16 are the easiest drop-ins for mixed-precision solvers. If you strictly need FP64, AMD's CDNA architectures offer native packed FP64 matrix ops, whereas NVIDIA users might need Ozaki-style emulation (combining multiple low-precision passes)

| | Supported CUDA Core Precisions | | | | | | | | | Supported Tensor Core Precisions | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | FP8 | FP16 | FP32 | FP64 | INT1 | INT4 | INT8 | TF32 | BF16 | FP8 | FP16 | FP32 | FP64 | INT1 | INT4 | INT8 | TF32 | BF16 |
| NVIDIA Tesla P4 | No | No | Yes | Yes | No | No | Yes | No | No | No | No | No | No | No | No | No | No | No |
| NVIDIA P100 | No | Yes | Yes | Yes | No | No | No | No | No | No | No | No | No | No | No | No | No | No |
| NVIDIA Volta | No | Yes | Yes | Yes | No | No | Yes | No | No | No | Yes | No | No | No | No | No | No | No |
| NVIDIA Turing | No | Yes | Yes | Yes | No | No | Yes | No | No | No | Yes | No | No | Yes | Yes | Yes | No | No |
| NVIDIA A100 | No | Yes | Yes | Yes | No | No | Yes | No | Yes | No | Yes | No | Yes | Yes | Yes | Yes | Yes | Yes |
| NVIDIA H100 | No | Yes | Yes | Yes | No | No | Yes | No | Yes | Yes | Yes | No | Yes | No | No | Yes | Yes | Yes |

The **Microscaling (MX) Format** is an open standard (driven by the Open Compute Project, involving NVIDIA, AMD, ARM, and others) designed to push matrix math into extreme low-precision realms—like 6-bit or 4-bit—without destroying the mathematical accuracy of Neural Networks.

# Tensor/Matrix Cores – Shapes

- **Hardware Alignment:** Matrix cores do not execute arbitrary M x N x K dimensions. The hardware expects inputs to be multiples of specific tile sizes (typically 16, 32, 64, or 128 depending on the generation and precision)
- **Padding is your friend:** If your matrix is 31 x 31, it is almost always faster to pad it with zeros to 32 x 32 and run it through a Tensor Core than to run the exact 31 x 31 math on standard scalar CUDA cores
- **Memory alignment:** The leading dimension of your matrices in global memory must be aligned to 128-byte boundaries to allow vectorized memory fetches (like Hopper's TMA). Unaligned memory access will starve the matrix engines

How do I know the matrix shapes supported by a given architecture: Look for **m**, **n**, **k** shapes mentioned in NVIDIA PTX manual or AMD ISA manual

# Tensor/Matrix Cores – LLM vs HPC

**LLM Developers**

- Your biggest enemy is memory bandwidth (the "Memory Wall")
- Tensor cores compute so fast that LLM inference is almost entirely bottlenecked by moving weights and KV-cache data from HBM
- Success means aggressive quantization (Weight-Only quantization) and maximizing data reuse (fused-kernels)

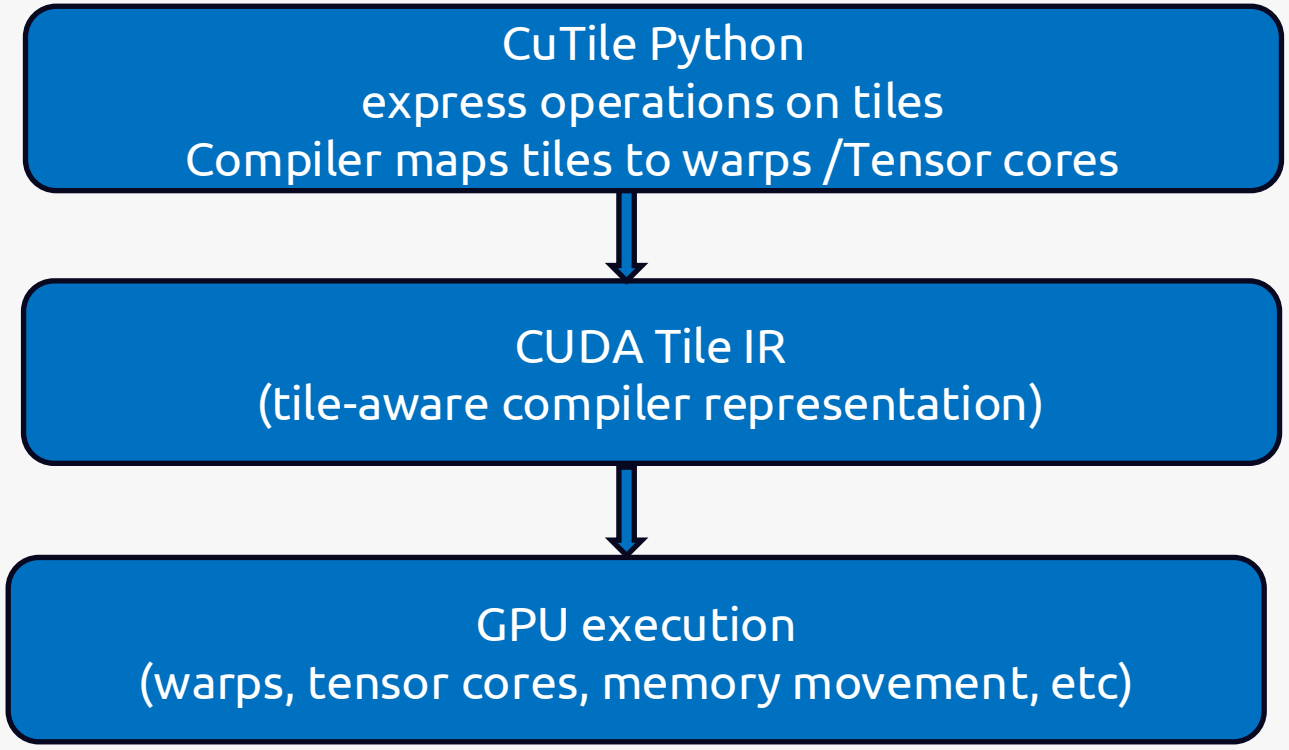
**HPC Developers**

- Your biggest enemy is the condition number and error propagation
- Success means reformulating sparse, irregular problems into dense "blocks" (e.g., Block-Jacobi preconditioning) to feed the matrix engines and using iterative refinement (computing the residual in high precision) to fix the errors introduced by the fast, low-precision matrix engines

# CUDA – cuTile

- NVIDIA's Python-based tile programming DSL, stabilized in CUDA 13.2 after debuting experimentally in 13.1
- cuTile shifts CUDA kernel from ***"what does each thread do?"*** to ***"what operation should this massive data tile perform?"***
- The compiler then maps tile operations to warps, Tensor Cores, scheduling, and resource allocation

(High level abstraction)




(Low level abstraction)

```
# Classic CUDA: express per-thread
indexing
i = blockIdx.x * blockDim.x + threadIdx.x
if i < n:
    C[i] = A[i] + B[i]
```

```
# cuTile: express a tile operation
a = tile.load(A, [row, col], shape=(BM, BN))
b = tile.load(B, [row, col], shape=(BM, BN))
tile.store(C, [row, col], a + b)
```

**Tile programming is becoming the common abstraction:** Triton is tile-oriented, CUDA Tile provides Tile IR and NVIDIA is developing a Triton-to-TileIR path that preserves tile-level semantics instead of immediately lowering them to thread-level PTX.

# CUDA – Matrix Multiplication in cuTile Python

Unlike traditional CUDA where you manually iterate through global memory and load into shared memory, cuTile lets you define the data flow at the block level

```
import cutile

# Define block tile dimensions
BLOCK_M, BLOCK_N, BLOCK_K = 128, 128, 32

@cutile.kernel
def matmul_tile_kernel(A, B, C):
    # Get block index coordinates natively
    bx, by = cutile.block_idx.x, cutile.block_idx.y

    # Initialize an accumulator tile in registers/shared memory for this block
    acc = cutile.zeros((BLOCK_M, BLOCK_N), dtype=cutile.float32)

    # Iterate through the K dimension in chunks of BLOCK_K
    for k in range(0, A.shape[1], BLOCK_K):
        # The system automatically handles loading these tiles from global memory
        # utilizing TMA if available on the hardware (e.g., Hopper/Blackwell).
        tile_A = A[by * BLOCK_M : (by + 1) * BLOCK_M, k : k + BLOCK_K]
        tile_B = B[k : k + BLOCK_K, bx * BLOCK_N : (bx + 1) * BLOCK_N]

        # Hardware-accelerated block-level matrix multiply (automatically maps to Tensor Cores)
        acc += cutile.dot(tile_A, tile_B)

    # Write the completed tile back to global memory
    C[by * BLOCK_M : (by + 1) * BLOCK_M, bx * BLOCK_N : (bx + 1) * BLOCK_N] = acc
```

# Tensor/Matrix Cores – Power Usage

**The "Power Virus" Effect**

- Tensor cores pack an immense amount of switching logic into a tiny physical area

- Running sustained dense GEMMs will cause the GPU to hit its thermal or power limit almost instantly, causing clock frequencies to drop (throttling)

**Energy Efficiency per FLOP**

- Despite the massive power draw, Tensor Cores are vastly more *energy-efficient* per calculation than standard CUDA cores

**The real energy cost is data movement**

- Moving a byte of data from HBM to the SM costs orders of magnitude more picojoules than doing the math on it

- If you fire up the Tensor Cores, you *must* maximize data reuse in the register file and shared memory; otherwise, you are just burning hundreds of watts waiting on VRAM.

# OpenCL – **Open C**ompute **L**anguage

**A Specification, Not a Product:** Maintained by the Khronos Group (the same consortium behind Vulkan and OpenGL). Khronos writes the *rules* (the API and memory model); hardware vendors (Intel, AMD, NVIDIA, ARM, etc.) write the *implementations* (the drivers)

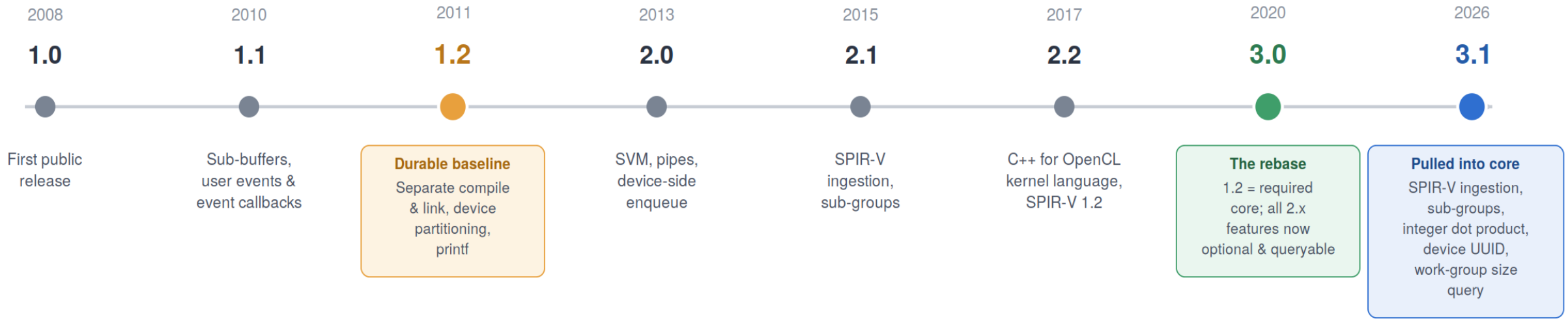

A C/C++ API used by the host CPU to discover devices, manage memory, and submit work queues.

A hierarchy of work-items and work-groups (conceptually identical to CUDA threads and thread-blocks).

A C-based language (OpenCL C) and an intermediate representation (SPIR-V) used to write the actual code that executes on the accelerator

**Hardware Agnostic:** Designed to target CPUs, GPUs, DSPs, FPGAs, and custom AI accelerators using the exact same API

# OpenCL – Evolution Timeline




**Milestones**

- 1.2 — the baseline that became the required core
- 3.0 — unified spec; portability becomes “query, don’t assume”
- 3.1 — field-proven AI/HPC features mandated across conformant implementations

*In-flight extensions: command buffers, cooperative matrix, low-precision AI data types, USM improvements*

# OpenCL – Evolution Timeline

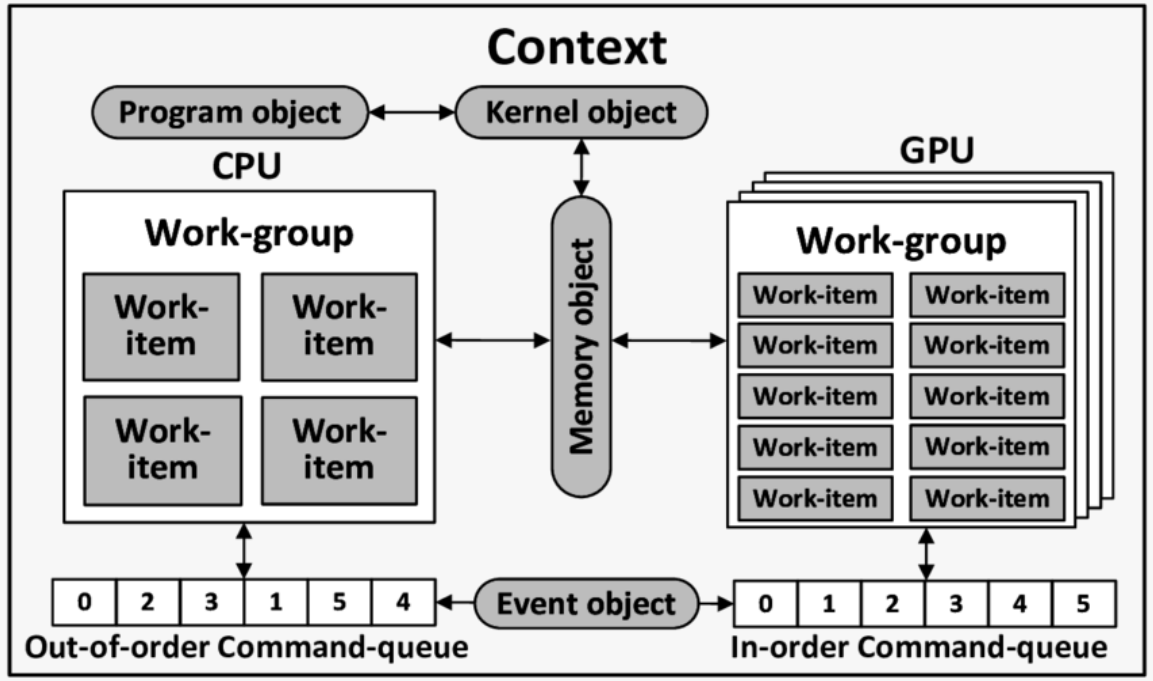
| Concept / Feature | CUDA (NVIDIA) | HIP (AMD) | OpenCL (Multi-Vendor) |
|---|---|---|---|
| **Individual Execution Unit** | Thread | | Work-item |
| **Lockstep Group (SIMT)** | Warp (Fixed at 32) | Wavefront (64 or 32) | Subgroup (size queried at runtime) |
| **Shared Memory Group** | Thread Block | | Work-group |
| **Global Dispatch Size** | Grid | | NDRange |
| **Per-Thread Memory** | Local Memory (Registers/Stack) | Local Memory (VGPRs/Stack) | Private Memory |
| **Fast On-Chip Memory** | Shared Memory | LDS (Local Data Share) | Local Memory |
| **Main Device Memory** | Global Memory | | |
| **Unified Pointer Chasing** | cudaMallocManaged | hipMallocManaged | SVM/USM (Unified Shared Memory) |
| **CPU Overhead Reduction** | CUDA Graphs | HIP Graphs | Command Buffers (cl_khr_command_buffer) |
| **Compilation Model** | Single-source C++ (nvcc/PTX) | Single-source C++ (hipcc/LLVM) | Decoupled Host/Device (SPIR-V binaries) |
| **Argument Binding** | Implicit via C++ function calls | | Explicit via clSetKernelArg |
| **Matrix/Tensor Math** | WMMA / WGMMA / tcgen05 | MFMA Intrinsics | cl_khr_subgroup_matrix_... |
| **Hardware Extensions** | Thread Block Clusters, TMA | Wave32 (CDNA 5), Tensor Data Mover | None (Abstracted away from hardware) |

# OpenCL – Execution & Memory Model

OpenCL's hierarchies are strictly defined by "scope" and "visibility." This strictness ensures a kernel can execute safely on a discrete GPU, a multi-core CPU, or an FPGA without changing the logic

**Execution Model (The NDRange)**
OpenCL dispatches work over an N-Dimensional Range (1/2/3D)

- **Work-item (Thread):** The fundamental unit of execution. It executes the kernel and has a unique global and local ID
- **Subgroup (Warp/Wavefront):**
  - *Now a core requirement in OpenCL 3.1*
  - Grouping of work-items that execute in lockstep (SIMT)
  - Query the subgroup size at runtime
- **Work-group (Thread Block):** A collection of work-items that execute concurrently on a single Compute Unit (e.g., an NVIDIA SM or AMD WGP)




**Memory Model**
**Private (Registers):** Visible only to a single work-item
**Local (Shared Memory / LDS):** Visible to all work-items in a *work-group*
**Global / Constant Memory (VRAM):** Visible to *all* work-groups across the entire NDRange
**Host Memory (System RAM):** Historically separated by the PCIe bus, requiring explicit `clEnqueueRead/Write` commands to move data

# OpenCL – Modern Features

Within **OpenCL**, **Shared Virtual Memory (SVM)** is the official feature introduced in OpenCL 2.0, while **Unified Shared Memory (USM)** is a newer OpenCL extension that provides a simpler allocation model inspired by SYCL.

- **Use USM** if your OpenCL implementation supports **cl_khr_unified_shared_memory** (or the earlier **cl_intel_unified_shared_memory** extension):
  - Simpler and more intuitive memory allocation APIs.
  - Supports **host**, **device**, and **shared** memory allocations
  - Recommended for new OpenCL applications when available
- **Use SVM** if:
  - Your target platform supports **OpenCL 2.x SVM** but not the USM extension
  - You need compatibility with existing OpenCL 2.x applications that already use SVM
  - Your application relies on SVM-specific features (e.g., fine-grained or system SVM, where supported)

| Feature | SVM | USM |
|---|---|---|
| **OpenCL version** | OpenCL 2.0 | OpenCL extension (`cl_intel_unified_shared_memory`), later standardized as `cl_khr_unified_shared_memory` |
| **Allocation API** | clSVMAlloc() | `clHostMemAllocINTEL()`, `clDeviceMemAllocINTEL()`, `clSharedMemAllocINTEL()` (or KHR equivalents) |
| **Pointer usage** | SVM pointers | USM pointers |
| **Device-only allocations** | No | Yes |
| **Fine-grained support** | Depends on hardware capabilities | Not based on fine/coarse-grained SVM capabilities |

# OpenCL 3.1 – Raising the floor for HPC & AI

OpenCL 3.1 re-establishes a strong baseline by taking field-proven extensions and making them **mandatory core requirements**

**1. Mandatory SPIR-V Ingestion**
**What changed:** Drivers must now officially support consuming SPIR-V intermediate representation binaries
**Why it matters:** It not required to ship raw OpenCL C source and rely on the runtime driver compiler, shrinks application startup

**2. Subgroups Are Now Core -** The subgroup programming model (CUDA Warps or HIP Wavefronts) is core

**3. Native Hardware AI Primitives**
**Integer Dot Products:** Built-in hardware support for integer dot-product ops for quantized (INT8/INT4) AI inference
**On the Horizon (Drafts):** OpenCL 3.1 sets the stage for standardizing `cl_khr_cooperative_matrix` (matrix multiply-add ops) and low-precision AI data types, providing a vendor-neutral path to Matrix/Tensor Cores

**4. Device Identification & Alignment**
**Device UUIDs:** Standardized UUID queries (matching Vulkan's behavior) are now mandatory
**Why it matters:** In complex, multi-GPU, or mixed-API workflows (e.g., Vulkan rendering mixed with OpenCL compute), developers can deterministically map a specific physical GPU across different APIs

**5. Quality of Life Updates**
**Work-Group Sizing:** A new standardized query allows the runtime to suggest the optimal local work-group size for a given kernel and device architecture

**Language Improvements:** Relaxed memory model scopes and massively improved `printf()` behavior directly within OpenCL C

# OpenCL – Cooperative Matrix Extensions for Machine Learning

- ***Problem:*** Exposing hardware Tensor Cores, Matrix Cores, Intel XMX, and mobile NPUs typically requires vendor-locked built-ins
- ***Standard:*** `cl_khr_cooperative_matrix` (working draft, published late 2025, developed with Arm, Intel, Qualcomm)
- **Sub-group Scoped Math:** Introduces matrix load/store/multiply-accumulate operations directly at the sub-group level acting as a portable abstraction layer over vastly different physical matrix hardware
- **Cross-API Convergence:** It deliberately mirrors Vulkan's `VK_KHR_cooperative_matrix` (which has been shipping since 2023
- **The SPIR-V Backbone:** Because both Vulkan and OpenCL now share the exact same SPV_KHR_cooperative_matrix SPIR-V backbone, AI and graphics compiler ecosystems can finally target a single, unified matrix primitive instead of reinventing it per API

# OpenCL – Developer Ergonomics

**CUDA / HIP:** Mostly single-source C++. `__host__` and `__device__` code in the same `.cu/.cpp`

- *Pros:* Template metaprogramming (like CUTLASS) is seamless

**OpenCL API:** Host and Device code are explicitly decoupled. The host runs standard C/C++, while the device loads a SPIR-V module

- *Pros:* Unmatched binary portability across vendors. OpenCL 3.1's mandatory SPIR-V ingestion guarantees massive reductions in application startup
- *Cons:* Binding arguments via `clSetKernelArg` is verbose and manual. Writing heavily templated C++ device code requires configuring offline LLVM toolchains rather than just calling nvcc

**The Modern Solution (SYCL):** If you want the single-source C++ ergonomics of CUDA/HIP (no clSetKernelArg) but the cross-platform reach of OpenCL, the industry answer is **SYCL**. SYCL acts as the modern C++ front-end that compiles down to the SPIR-V artifacts that OpenCL 3.1 consumes

# Power & Energy

Why a PhD student cares now: energy-to-solution is a reportable metric, node power caps are real, and reviewers increasingly ask

| CUDA | HIP | Portable/Cross Vendor |
|---|---|---|
| **NVML** — instantaneous power, accumulated energy, power limits, clock locking<br>`nvmlDeviceGetPowerUsage()`<br>`nvmlDeviceGetTotalEnergyConsumption()`<br><br>NVIDIA Data Center GPU Manager (**DCGM**) — multi-node monitoring and fleet telemetry<br><br>**CUPTI** — performance-counter sampling (not really a direct power or energy measurement)<br><br>**nvidia-smi** — quick interactive checks | **AMD SMI** — power, energy accumulators, and power-cap control<br>`amdsmi_get_power_info()`<br><br>**rocprof / rocprofiler** — profiling-counter access<br><br>**Migration:** ROCm-SMI → AMD SMI | **PAPI** — hardware-counter abstraction<br><br>**LIKWID / Variorum / EAR** — node and platform energy tooling<br><br>**Slurm energy accounting** — scheduler-visible job metrics<br><br>**ml-energy / Zeus** — ML-oriented measurement and experiment logging |

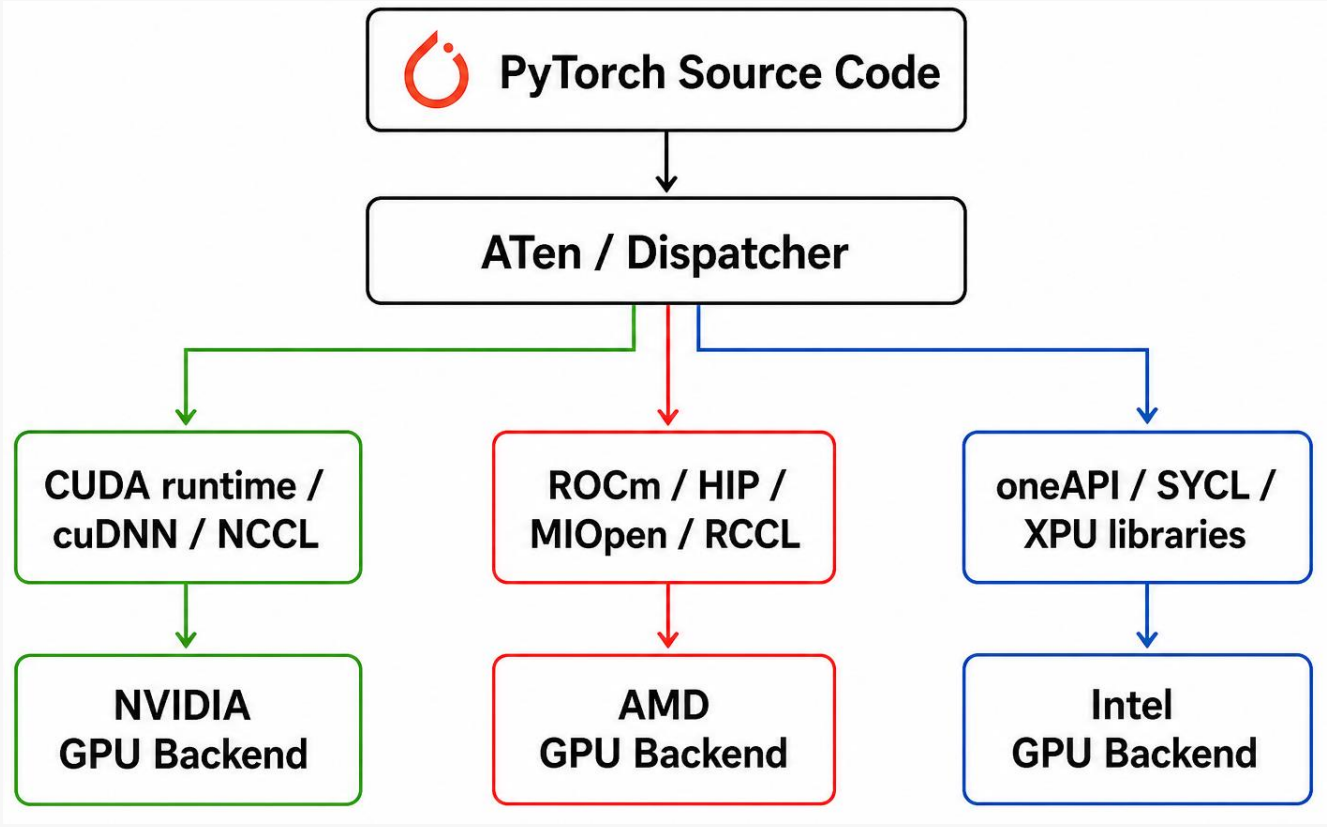
**Superchips/APUs:** The coherent CPU–GPU design can reduce costly data movement relative to PCIe-based (discrete) architectures, but it makes **whole-workflow energy** more meaningful than a GPU-only number. (eg., GH200, GB300, Vera-Rubin, MI300A)

# How AI/ML frameworks reach the GPU

**The PyTorch Dispatch Architecture**
**The ATen Dispatcher:** When you call a PyTorch function, the ATen (A Tensor) library and the Dispatcher act as traffic controllers and instantly route the math to the correct hardware backend.dev-discuss

**Write Once, Run Anywhere:**
*NVIDIA:* Routes to native CUDA kernels and cuDNN
*AMD:* Routes to ROCm/HIP (PyTorch automatically generates HIP code from its CUDA backend to ensure feature parity
*Intel:* Routes through SYCL (oneAPI)

**The Advantage:** Developers write one single Python codebase (MACE, TorchSim, Torch-CFD, etc) and the framework handles the complex translation to heterogeneous hardware at runtime



# ARGONNE TRAINING PROGRAM ON EXTREME-SCALE COMPUTING

Produced by Argonne National Laboratory, a U.S. Department of Energy Laboratory managed by UChicagoArgonne, LLC under contract DE-AC02-06CH11357.

Special thanks to the National Energy Research Scientific Computing Center (NERSC) and Oak Ridge Leadership Computing Facility (OLCF) for the use of their resources during the training event.