CSCI 5451: Introduction to Parallel Computing

Lecture 27: Practical Extensions & Real-World CUDA

computer science
& engineering

University of Minnesota
Driven to Discover®

Overview

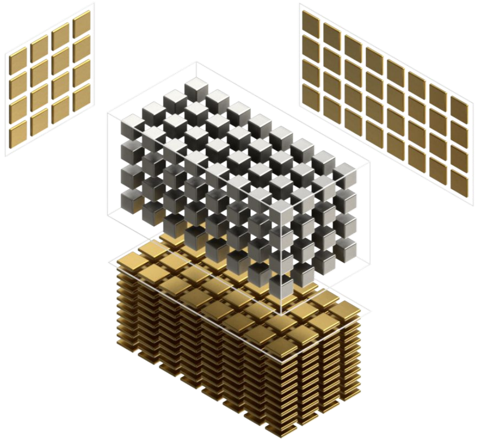
- ❑ Advanced CUDA Constructs
- ❑ Libraries
- ❑ Examples of Parallelism in Practice
- ❑ Tooling
- ❑ Alternative Hardware Architectures

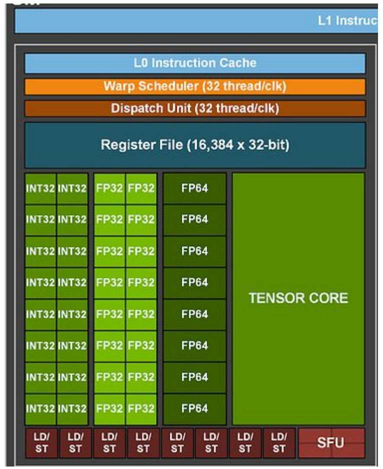
Overview

- ❑ **Advanced CUDA Constructs**
- ❑ Libraries
- ❑ Examples of Parallelism in Practice
- ❑ Tooling
- ❑ Alternative Hardware Architectures

Tensor Cores

- ❑ Specialized matrix-multiply-accumulate (MMA) units inside NVIDIA GPUs designed to accelerate dense linear algebra.
- ❑ Operate on small matrix tiles (e.g., 16×16×16) at extremely high throughput.
- ❑ Perform **D = A × B + C** in one hardware instruction (mma.sync).
- ❑ Accept low-precision inputs (FP16, BF16, TF32, INT8, FP8 on newer GPUs) and accumulate into higher precision (FP16/FP32).
- ❑ Integrated into each SM; a warp cooperatively issues a tensor operation.

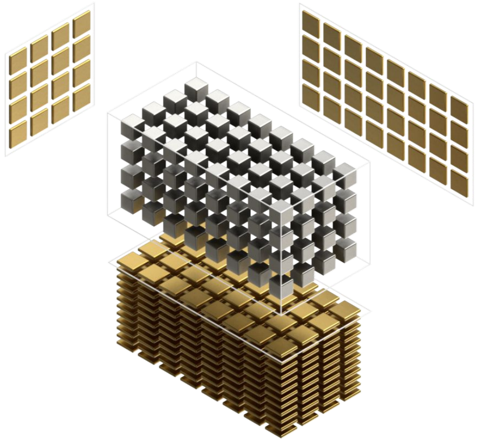


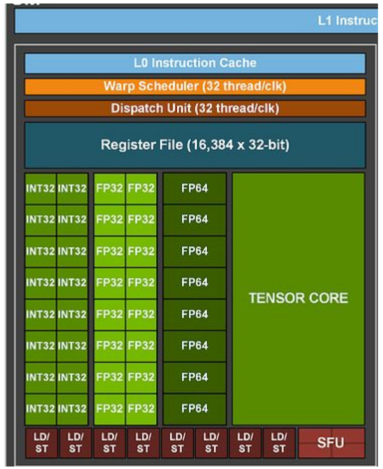

L1 Instruc
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
INT32 INT32
FP32 FP32
FP64
TENSOR CORE
LD/ ST
SFU

Tensor Cores

- ❑ A single warp (32 threads) collectively loads tile fragments of A, B, and C.
- ❑ The warp calls an MMA instruction → hardware performs the full matrix multiply & accumulate on the tile.

  Each instruction computes many FMA ops in parallel (e.g., 256 FMAs for a 16×16×16 tile).
- ❑ Outputs are stored into per-warp registers and then written to memory.
- ❑ The compiler maps WMMA/CUTLASS calls to optimized Tensor Core instructions.




L1 Instruc
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
INT32 INT32 FP32 FP32 FP64
TENSOR CORE
LD/ ST
SFU

Tensor Cores

```
__global__ void wmma_example(const half *A, const half *B, float *C) {
    // Each warp computes one 16x16 tile
    wmma::fragment<wmma::matrix_a, 16,16,16, half, wmma::row_major> a_frag;
    wmma::fragment<wmma::matrix_b, 16,16,16, half, wmma::row_major> b_frag;
    wmma::fragment<wmma::accumulator,16,16,16, float> c_frag;

    wmma::fill_fragment(c_frag, 0.0f);

    // Load 16×16 tiles from global memory
    wmma::load_matrix_sync(a_frag, A, 16);
    wmma::load_matrix_sync(b_frag, B, 16);

    // Tensor Core fused-multiply-accumulate:  C += A * B
    wmma::mma_sync(c_frag, a_frag, b_frag, c_frag);

    // Store result
    wmma::store_matrix_sync(C, c_frag, 16, wmma::mem_row_major);
}
```

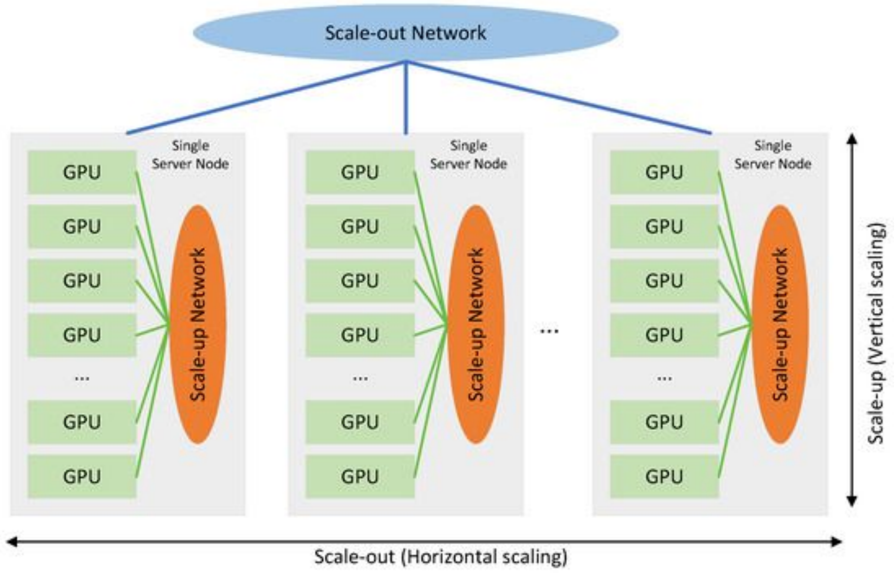
CUDA Streams

- A CUDA stream is a sequence of GPU operations executed in issue order.
  Operations in the same stream run *serially*; operations in different streams *may run concurrently*.
- Default stream (stream 0) synchronizes with all other work by default (“legacy” behavior).
- Streams enable concurrency:
  - Overlap kernel execution + memory copies
  - Run multiple independent kernels in parallel
  - Pipeline workloads for improved throughput
- Asynchronous operations
  - Most CUDA API calls have an async version (e.g., cudaMemcpyAsync) that takes a stream argument.

```
cudaStream_t s1, s2;
cudaStreamCreate(&s1);
cudaStreamCreate(&s2);

float *d_a, *d_b;
cudaMalloc(&d_a, N*sizeof(float));
cudaMalloc(&d_b, N*sizeof(float));

cudaMemcpyAsync(d_a, h_a, N*sizeof(float),
cudaMemcpyHostToDevice, s1);
cudaMemcpyAsync(d_b, h_b, N*sizeof(float),
cudaMemcpyHostToDevice, s2);

myKernel<<<grid, block, 0, s1>>>(d_a);
myKernel<<<grid, block, 0, s2>>>(d_b);

cudaStreamSynchronize(s1);
cudaStreamSynchronize(s2);
```

Multiple GPUs

- Oftentimes, you will not have a single GPU, but many
- These gpus will be organized into a single node
  - E.g. 4xA100 implies a node with 4 A100s
- A typical HPC setup will have many nodes connected to each other
- To use all GPUs in this configuration, we use primitives similar to MPI (NCCL)



Multiple GPUs

- ❑ Oftentimes, you will not have a single GPU, but many
- ❑ These gpus will be organized into a single node
  - o E.g. 4xA100 implies a node with 4 A100s
- ❑ A typical HPC setup will have many nodes connected to each other
- ❑ To use all GPUs in this configuration, we use primitives similar to MPI (NCCL)

Parallel Execution

```
size_t size = 1024 * sizeof(float);
cudaSetDevice(0);
float* p0;
cudaMalloc(&p0, size);
MyKernel<<<1000, 128>>>(p0);

cudaSetDevice(1);
float* p1;
cudaMalloc(&p1, size);
MyKernel<<<1000, 128>>>(p1);
```

Multiple GPUs

- ❑ Oftentimes, you will not have a single GPU, but many
- ❑ These gpus will be organized into a single node
  - o E.g. 4xA100 implies a node with 4 A100s
- ❑ A typical HPC setup will have many nodes connected to each other
- ❑ To use all GPUs in this configuration, we use primitives similar to MPI (NCCL)

Between GPU Communication

```
cudaSetDevice(0);
float* p0;
size_t size = 1024 * sizeof(float);
cudaMalloc(&p0, size);

cudaSetDevice(1);
float* p1;
cudaMalloc(&p1, size);

cudaSetDevice(0);
MyKernel<<<1000, 128>>>(p0);

cudaSetDevice(1);
cudaMemcpyPeer(p1, 1, p0, 0, size);
MyKernel<<<1000, 128>>>(p1);
```

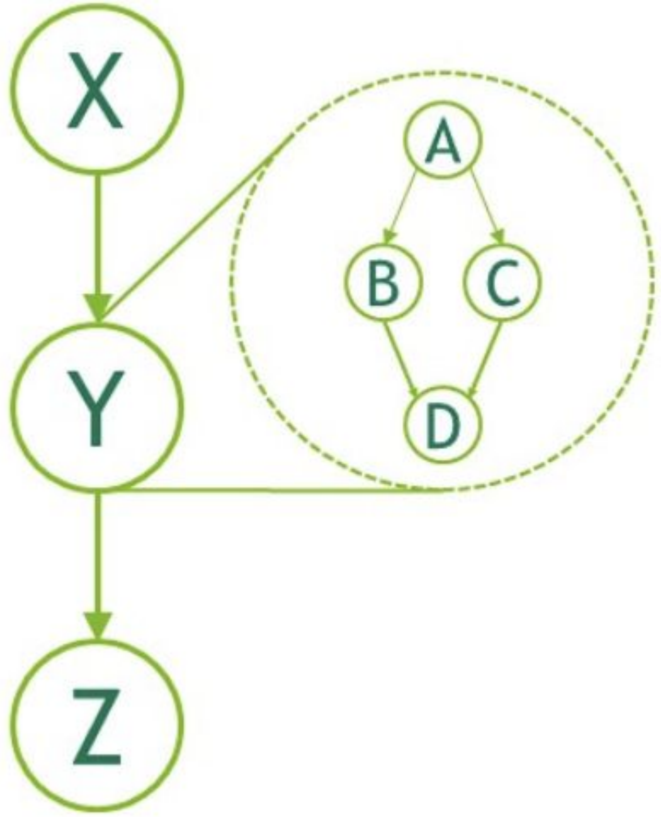
Dynamic Parallelism

- ❑ Dynamic Parallelism allows kernels to launch other kernels directly.
- ❑ Useful for irregular or adaptive workloads
- ❑ Reduces CPU–GPU synchronization by letting the GPU decide when to spawn work.
- ❑ Child kernels use the same <<<grid, block>>> syntax as host launches.
- ❑ Parent kernel must synchronize (or return) before child results are guaranteed.

```
__global__ void childKernel(int *out) {
    int i = threadIdx.x;
    out[i] = i * 2;
}

__global__ void parentKernel(int *out) {
    if (threadIdx.x == 0) {
        // Launch the child kernel from the device
        childKernel<<<1, 8>>>(out);
    }
}

int main() {
    int *d_out;
    cudaMalloc(&d_out, 8 * sizeof(int));

    parentKernel<<<1, 1>>>(d_out);
    cudaDeviceSynchronize();
}
```

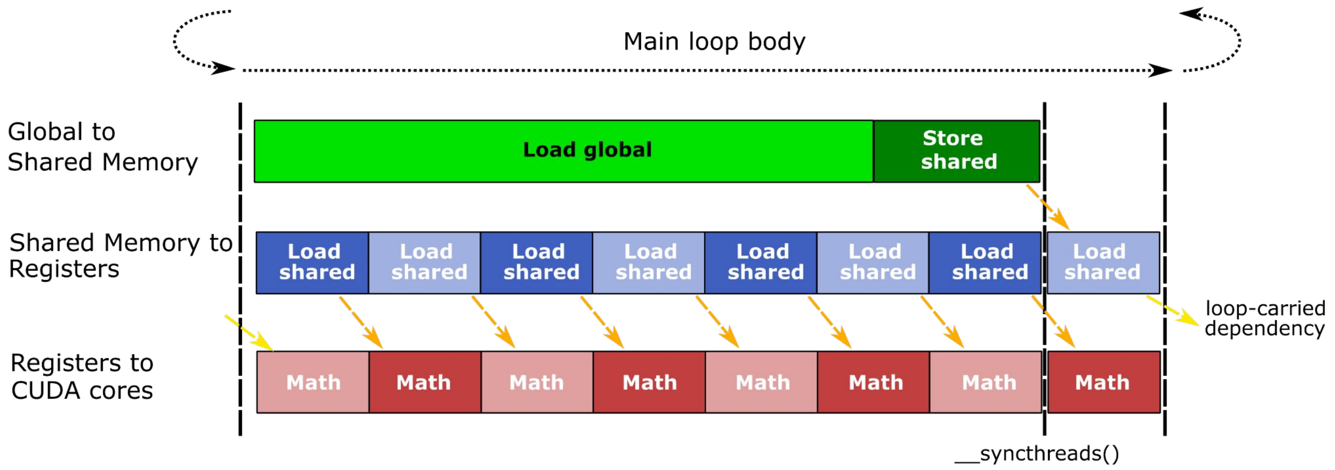
CUDA Graphs

- ❑ Allows for defining structured computation which will likely be repeated
- ❑ Useful for long running applications, or where kernels are invoked many times, but do not individually take long
- ❑ In these cases, the cost of starting up the kernel can begin to accumulate
- ❑ CUDA graphs allow for amortization of this cost only once at program start



Pipelining

- ❑ CUDA pipelining overlaps data transfer and kernel execution.
- ❑ Requires multiple non-default streams.
- ❑ Use cudaMemcpyAsync + kernel launches in different streams.
- ❑ Typical pattern: H2D → Compute → D2H done in staggered streams.
- ❑ Improves throughput when kernel time ≈ memcpy time.



Pipelining

```
Chunk 0: H2D → Kernel → D2H
Chunk 1:          H2D → Kernel → D2H
Chunk 2:                H2D → Kernel → D2H
```

```
for (int i = 0; i < N; i += CHUNK) {
    int off = i;

    cudaMemcpyAsync(d_in + off, h_in + off, CHUNK*sizeof(float),
                    cudaMemcpyHostToDevice, s1);
    kernel<<<grid, block, 0, s1>>>(d_in + off, d_out + off);
    cudaMemcpyAsync(h_out + off, d_out + off, CHUNK*sizeof(float),
                    cudaMemcpyDeviceToHost, s1);

    cudaMemcpyAsync(d_in + off + CHUNK, h_in + off + CHUNK,
                    CHUNK*sizeof(float), cudaMemcpyHostToDevice, s2);
    kernel<<<grid, block, 0, s2>>>(d_in + off + CHUNK,
                                   d_out + off + CHUNK);
    cudaMemcpyAsync(h_out + off + CHUNK, d_out + off + CHUNK,
                    CHUNK*sizeof(float), cudaMemcpyDeviceToHost, s2);
}

cudaDeviceSynchronize();
```

PTX/SASS

- PTX = NVIDIA’s virtual ISA (IR between CUDA and hardware).
- SASS = actual hardware instructions executed by the GPU.
- PTX is stable across architectures; SASS is architecture specific.
- Inline PTX allows precise control over instructions inside CUDA C++.
- Use `nvcc -ptx` to inspect compiler-generated PTX.
- Use `cuobjdump --dump-sass` (or `nvdisasm`) to inspect SASS.
- Performance debugging often requires looking at SASS, not PTX.
- Register pressure, memory ops, and instruction mix visible in SASS.

PTX/SASS

PTX

```
ld.global.f32  %f1, [x];
ld.global.f32  %f2, [y];
add.f32        %f3, %f1, %f2;
st.global.f32  [z], %f3;
```

SASS

```
LDG.E.SYS R4, [R2];
LDG.E.SYS R5, [R3];
FADD R6, R4, R5;
STG.E.SYS [R7], R6;
```

C

```
// compile with: nvcc -arch=sm_80 ptx_inline.cu -o ptx_inline
__global__ void add_ptx(float *x, float *y, float *z) {
    int i = threadIdx.x;
    float a = x[i], b = y[i], c;

    // inline PTX: c = a + b
    asm("add.f32 %0, %1, %2;" : "=f"(c) : "f"(a), "f"(b));

    z[i] = c;
}

int main() {
    add_ptx<<<1, 32>>>(x, y, z);
}
```

Overview

- ❑ Advanced CUDA Constructs
- ❑ **Libraries**
- ❑ Examples of Parallelism in Practice
- ❑ Tooling
- ❑ Alternative Hardware Architectures

cuBLAS

- ❑ cuBLAS is NVIDIA’s optimized BLAS library for GPUs.
- ❑ General library for performing vector, matrix-vector, matrix-matrix operations
- ❑ cuBLAS achieves near–peak FLOP/s using tensor cores when allowed (TF32/FP16/etc.).
- ❑ cublasCreate() allocates internal context; reuse the handle for performance.
- ❑ High level of abstraction - if you just need primitives, this library is a good place to start

cuBLAS

GPU-accelerated basic linear algebra (BLAS) library

cuBLAS

```
// assume A,B,C allocated on device; A(m×k), B(k×n), C(m×n)
cublasHandle_t handle;
cublasCreate(&handle);

const float alpha = 1.0f, beta = 0.0f;

// C = alpha * A * B + beta * C   (column-major convention)
cublasSgemm(handle,
            CUBLAS_OP_N, CUBLAS_OP_N,
            n, m, k,                   // note argument order: col-major
            &alpha,
            B, n,                      // B is on the left in column-major
            A, k,
            &beta,
            C, n);

cublasDestroy(handle);
```

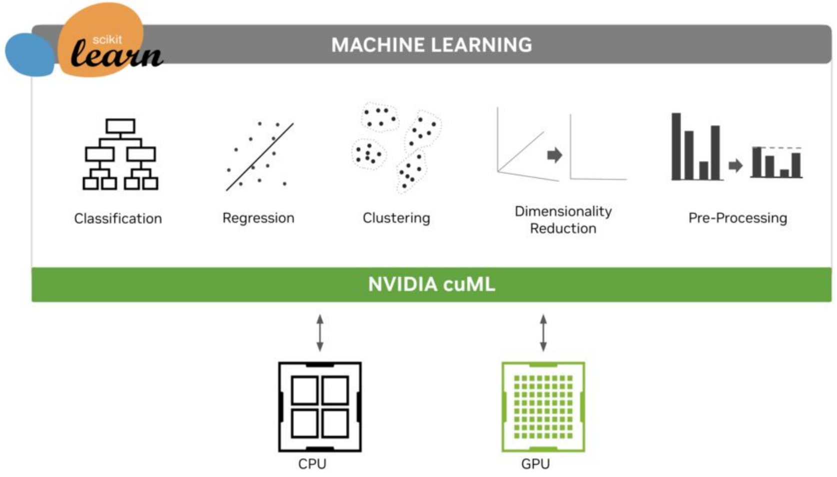
cuDNN

- ❑ cuDNN = NVIDIA's high-performance deep-learning primitives (conv, RNN, norm, activations).
- ❑ Provides highly-optimized GPU kernels (including Tensor Core usage) for DL frameworks.
- ❑ You configure operations using descriptors (tensor, filter, convolution, activation).
- ❑ Runtime selects fastest algorithm (FFT, Winograd, Implicit GEMM, Tensor Op).
- ❑ cuDNN performs: convolution, pooling, softmax, batchnorm, RNN/LSTM, attention (newer).
- ❑ Used in PyTorch, TensorFlow, JAX, MXNet, ONNX Runtime, Triton, TensorRT.
- ❑ Typical workflow: create handle → set descriptors → choose algorithm → allocate workspace → run op.

Other useful, high level cuda libraries

Each of the following gives high level kernels that can be called "out-of-the-box". No configuration is required outside of making the calls with the necessary inputs for your given problem

- ❑ cuFFT
- ❑ cuSparse
- ❑ cuSolver
- ❑ CuPy
- ❑ cuML



CUTLASS C++

- ❑ CUTLASS is a CUDA C++ template library for high-performance kernels.
- ❑ Provides building blocks for tensor cores, tiling, and epilogue fusion.
- ❑ Backbone of PyTorch, TensorRT, cuBLAS, and many NVIDIA ML kernels.
- ❑ Supports FP16, BF16, TF32, FP32, INT8, and Hopper FP8 pipelines.
- ❑ Used to write custom fused kernels beyond what cuBLAS exposes.
- ❑ If you need lower level than cuBLAS, but want higher abstractions than kernels themselves, then use CUTLASS

```
int main() {
  using Gemm = cutlass::gemm::device::Gemm<
      float, cutlass::layout::RowMajor,
      float, cutlass::layout::RowMajor,
      float, cutlass::layout::RowMajor>;

  Gemm gemm_op;

  float *A, *B, *C;   // assume allocated on device
  int M = 1024, N = 1024, K = 1024;

  Gemm::Arguments args({M, N, K}, {A, K}, {B, N},
{C, N}, {C, N});

  gemm_op(args);
}
```

CuTe DSL

- Essentially the python version of CUTLASS C++
- Python API wraps tuned kernels without requiring C++ compilation.

```
import cutlass
import torch

A = torch.randn(1024, 1024, device="cuda", dtype=torch.float16)
B = torch.randn(1024, 1024, device="cuda", dtype=torch.float16)
C = torch.zeros(1024, 1024, device="cuda", dtype=torch.float16)

gemm = cutlass.op.gemm(
    A.shape[0], A.shape[1], B.shape[1],
    element=A.dtype, accumulate=torch.float32
)

gemm(A, B, C)
torch.cuda.synchronize()
print("Done")
```

OpenAI Triton

- ❑ Triton enables writing custom GPU kernels in Python-like syntax.
- ❑ Kernels operate over program_id blocks instead of CUDA thread IDs.
- ❑ Compiler performs automatic tiling, vectorization, and memory coalescing.
- ❑ Supports NVIDIA GPUs (Ampere, Hopper, Ada) with fast kernel generation

```
@triton.jit
def add_kernel(x_ptr, y_ptr, out_ptr, n, BLOCK: tl.constexpr):
    pid = tl.program_id(0)
    offs = pid * BLOCK + tl.arange(0, BLOCK)
    mask = offs < n
    x = tl.load(x_ptr + offs, mask=mask)
    y = tl.load(y_ptr + offs, mask=mask)
    tl.store(out_ptr + offs, x + y, mask=mask)

def add(x, y):
    n = x.numel()
    out = torch.empty_like(x)
    BLOCK = 128
    grid = lambda meta: (triton.cdiv(n, meta['BLOCK']),)
    add_kernel[grid](x, y, out, n, BLOCK=BLOCK)
    return out

x = torch.randn(1024, device='cuda')
y = torch.randn(1024, device='cuda')
z = add(x, y)
print(z[:5])
```

Overview

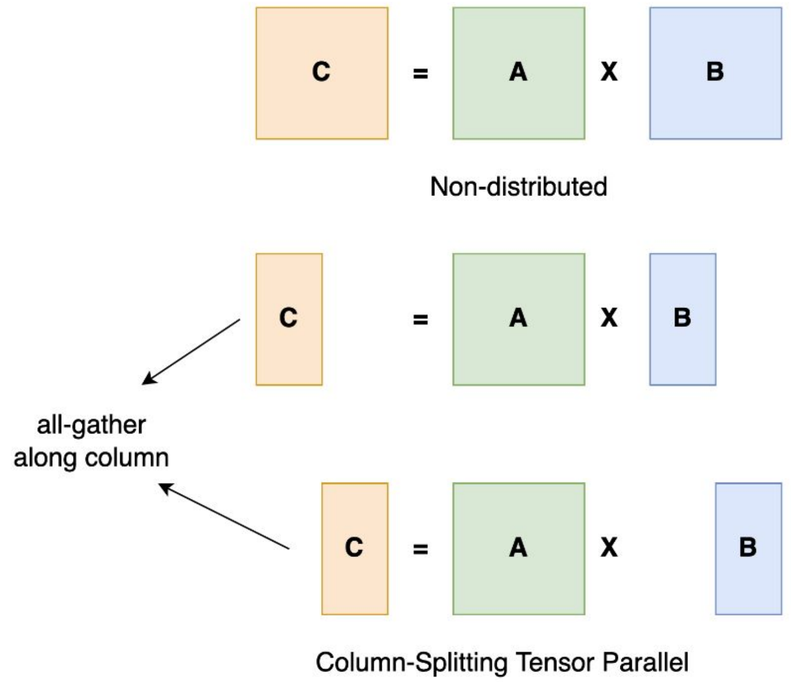
- ❑ Advanced CUDA Constructs
- ❑ Libraries
- ❑ **Examples of Parallelism in Practice**
- ❑ Tooling
- ❑ Alternative Hardware Architectures

LLM Parallelism

- ❑ Tensor Parallelism
- ❑ Data Parallelism
- ❑ Context Parallelism
- ❑ Pipeline Parallelism

LLM Parallelism

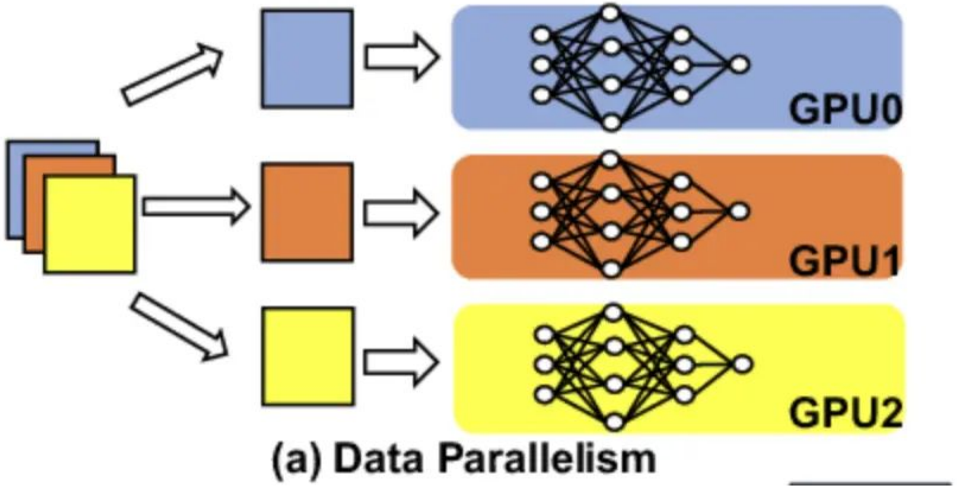
- ❑ **Tensor Parallelism**
- ❑ Data Parallelism
- ❑ Context Parallelism
- ❑ Pipeline Parallelism



Tensor parallel illustration

LLM Parallelism

- ❑ Tensor Parallelism
- ❑ **Data Parallelism**
- ❑ Context Parallelism
- ❑ Pipeline Parallelism

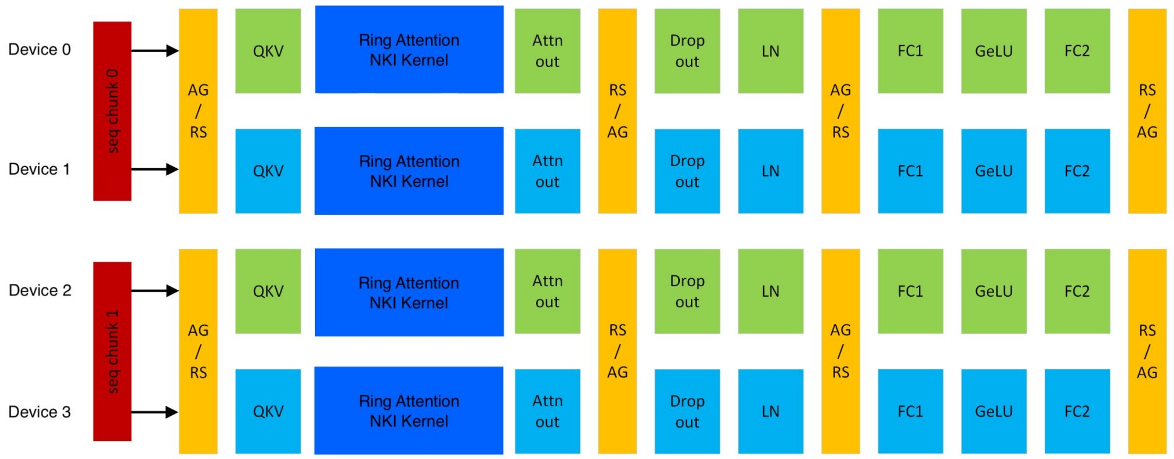


(a) Data Parallelism

LLM Parallelism

- ❑ Tensor Parallelism
- ❑ Data Parallelism
- ❑ **Context Parallelism**
- ❑ Pipeline Parallelism

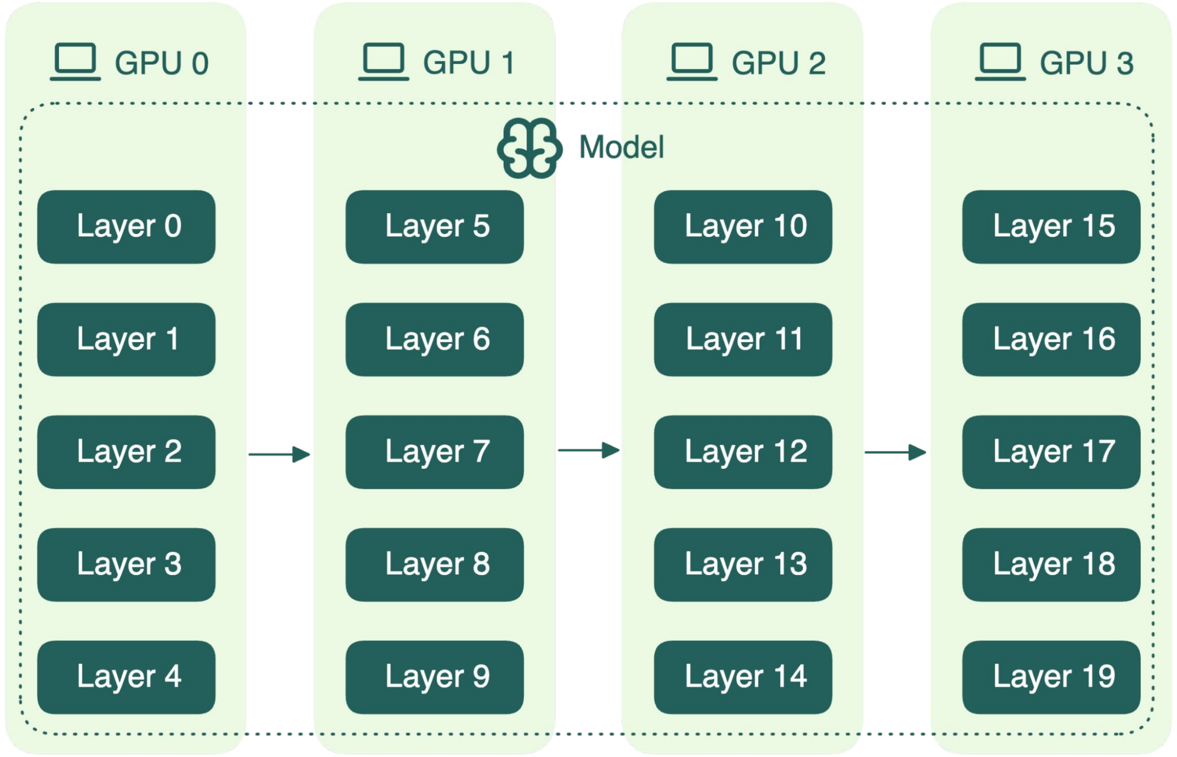
Transformer Layer with TP+CP



LLM Parallelism

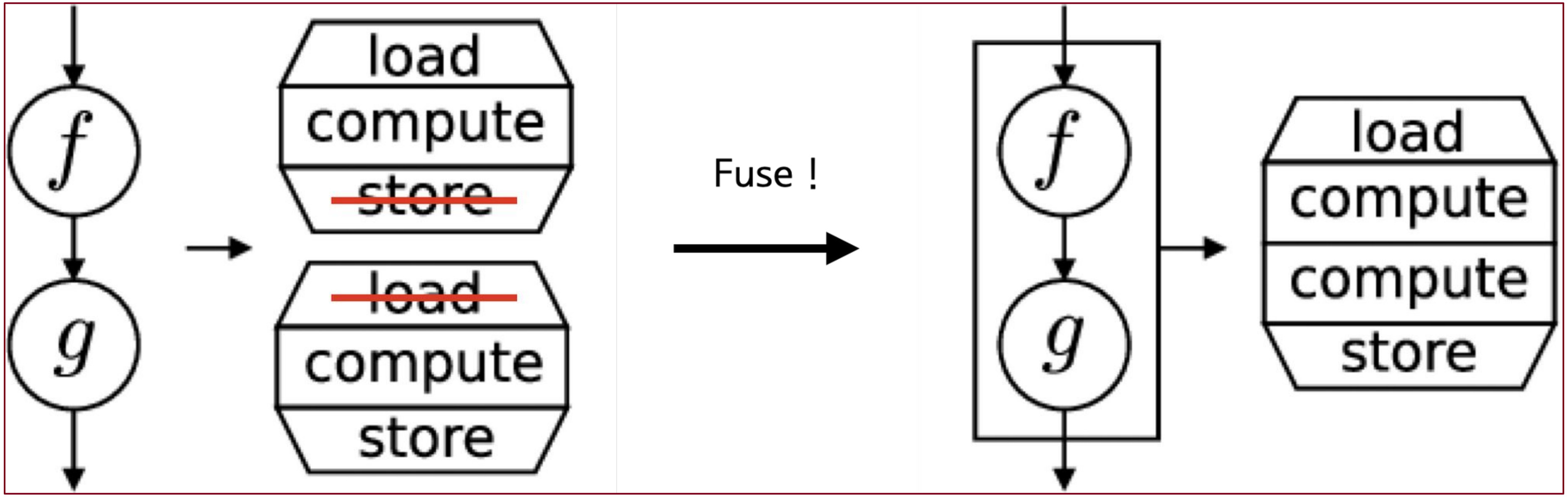
- ❑ Tensor Parallelism
- ❑ Data Parallelism
- ❑ Context Parallelism
- ❑ **Pipeline Parallelism**

Pipeline Parallelism



Kernel Fusion

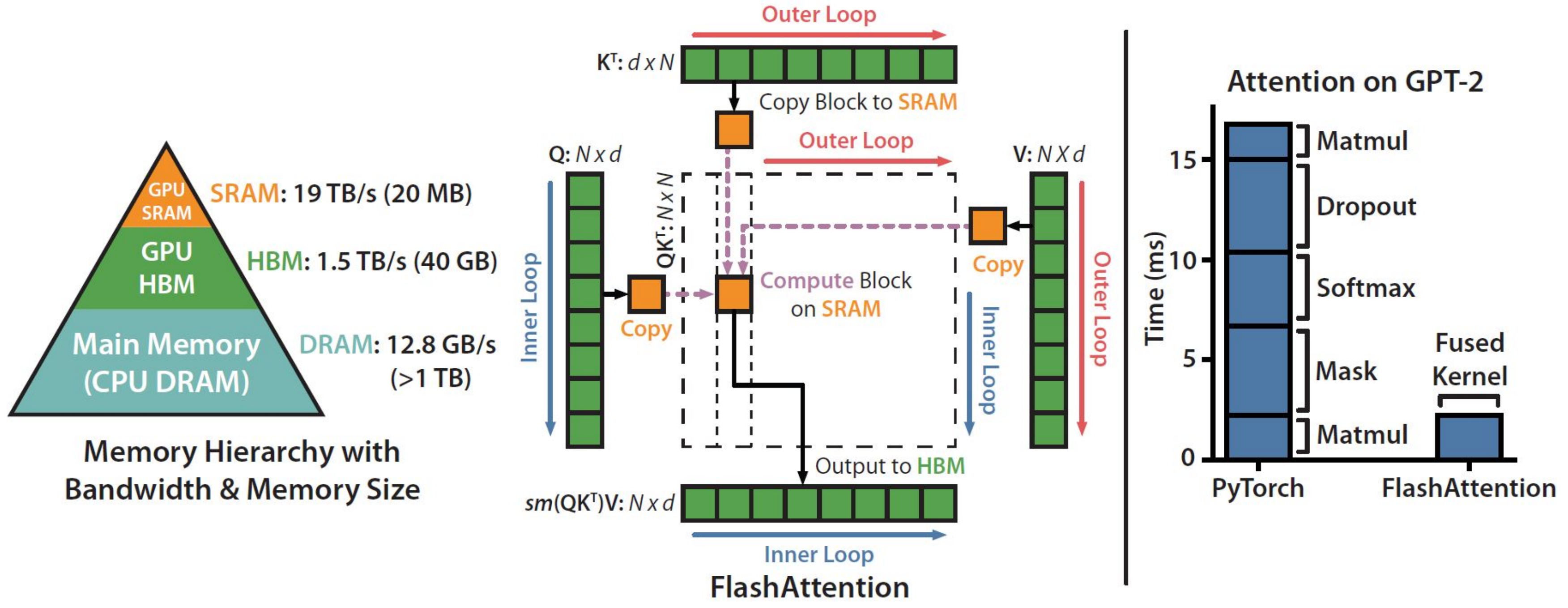
Combine multiple, separate kernels,
into a single kernel



Deep Learning Compilers

- ❑ Deep learning compilers optimize the entire computation graph, not just individual kernels.
- ❑ Kernel fusion reduces memory traffic (HBM bottleneck) and greatly improves performance.
- ❑ Compilers select and autotune the best kernels (CUTLASS, cuDNN, Triton, custom fused ops).
- ❑ Generate device-specific optimized code (PTX/SASS, tensor-core pipelines, tiling).
- ❑ Manage memory, scheduling, and parallelism across GPU(s) for training and inference.

Attention



GPU SRAM
SRAM: 19 TB/s (20 MB)
GPU HBM
HBM: 1.5 TB/s (40 GB)
Main Memory (CPU DRAM)
DRAM: 12.8 GB/s (>1 TB)
Memory Hierarchy with Bandwidth & Memory Size
Outer Loop
K^T: $d \times N$
Copy Block to SRAM
Outer Loop
Q: $N \times d$
V: $N \times d$
QK^T: $N \times N$
Copy
Inner Loop
Compute Block on SRAM
Copy
Inner Loop
Outer Loop
Output to HBM
$sm(QK^T)V$: $N \times d$
Inner Loop
FlashAttention
Attention on GPT-2
Time (ms)
0
5
10
15
Matmul
Dropout
Softmax
Mask
Matmul
Fused Kernel
PyTorch
FlashAttention

Worklogs + Reference Lectures for Advanced CUDA

- ❑ Inside NVIDIA GPUs: Anatomy of high performance matmul kernell (Blog)
- ❑ How to Optimize a CUDA Matmul Kernel for cuBLAS-like Performance: a Worklog (Blog)
- ❑ Learning CUDA by optimizing softmax: A worklog (Blog)
- ❑ Normalization layer optimization : Worklog (Blog)
- ❑ GPU-MODE (lectures on more cuda internals)

Overview

- ❑ Advanced CUDA Constructs
- ❑ Libraries
- ❑ Examples of Parallelism in Practice
- ❑ **Tooling**
- ❑ Alternative Hardware Architectures

Profiling (Nvidia Nsight Compute)

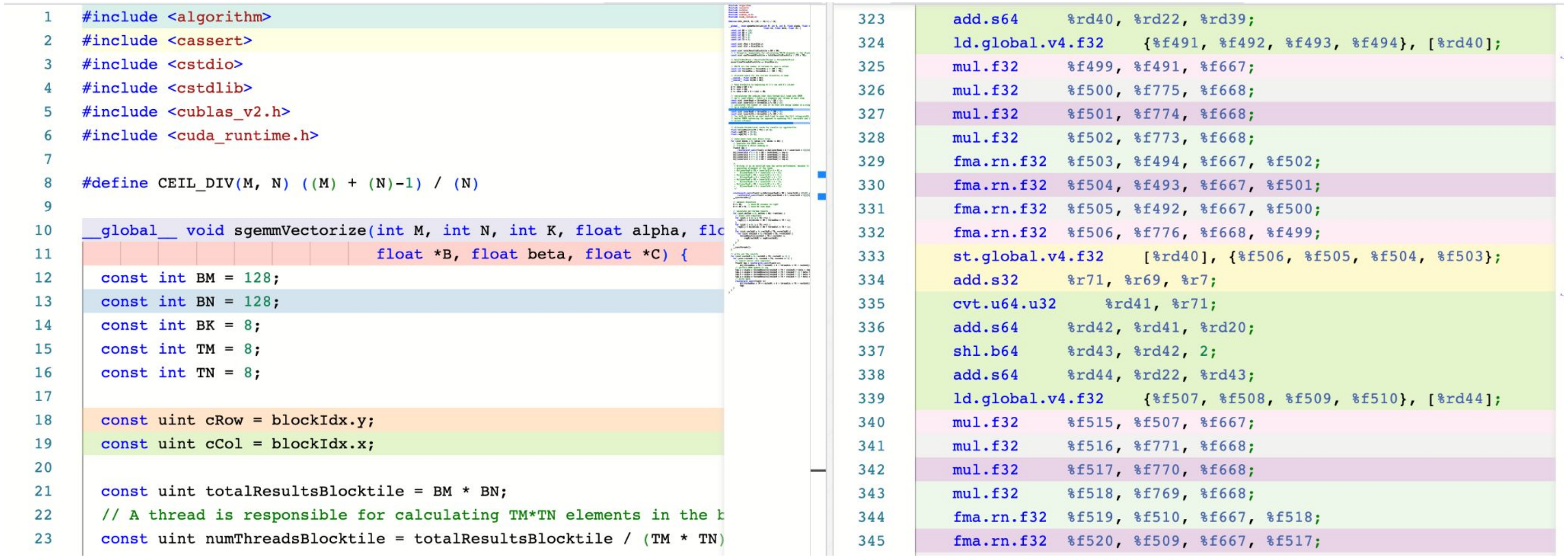
- Determines how the kernel itself performs
- Nsight Compute provides fine-grained GPU metrics such as cache hit rates, occupancy, warp stalls, and memory throughput.
- Kernel-level GPU profiler focused on per-kernel performance metrics.
- Reports warp occupancy, memory throughput, tensor core usage, and stall reasons.
- Allows section-based analysis (Memory, Scheduler, Roofline, Warp State).
- Supports detailed profiling on modern NVIDIA chips (Volta, Turing, Ampere, Hopper, Blackwell).

Profiling (Nvidia Nsight Systems)

- ❑ Determines end-end performance of kernel
- ❑ System-wide profiler for CPU–GPU interactions and timeline analysis.
- ❑ Identifies kernel launch bottlenecks, synchronization stalls, and pipeline gaps.
- ❑ Provides multi-process and multi-node tracing for HPC workloads.
- ❑ Supports profiling across NVIDIA GPU generations (Kepler → Ada/Lovelace → Hopper/Blackwell).

Godbolt

- ❑ Useful site for examining PTX/SASS in real-time
- ❑ Interactive compiler to examine how your work compiles to PTX, then into SASS
- ❑ For very high performance kernels, examining this code is crucial



```
#include <algorithm>
#include <cassert>
#include <cstdio>
#include <cstdlib>
#include <cublas_v2.h>
#include <cuda_runtime.h>

#define CEIL_DIV(M, N) ((M) + (N)-1) / (N)

__global__ void sgemmVectorize(int M, int N, int K, float alpha, flo
                               float *B, float beta, float *C) {
  const int BM = 128;
  const int BN = 128;
  const int BK = 8;
  const int TM = 8;
  const int TN = 8;

  const uint cRow = blockIdx.y;
  const uint cCol = blockIdx.x;

  const uint totalResultsBlocktile = BM * BN;
  // A thread is responsible for calculating TM*TN elements in the b
  const uint numThreadsBlocktile = totalResultsBlocktile / (TM * TN)
```

```
        add.s64         %rd40, %rd22, %rd39;
        ld.global.v4.f32        {%f491, %f492, %f493, %f494}, [%rd40];
        mul.f32         %f499, %f491, %f667;
        mul.f32         %f500, %f775, %f668;
        mul.f32         %f501, %f774, %f668;
        mul.f32         %f502, %f773, %f668;
        fma.rn.f32      %f503, %f494, %f667, %f502;
        fma.rn.f32      %f504, %f493, %f667, %f501;
        fma.rn.f32      %f505, %f492, %f667, %f500;
        fma.rn.f32      %f506, %f776, %f668, %f499;
        st.global.v4.f32        [%rd40], {%f506, %f505, %f504, %f503};
        add.s32         %r71, %r69, %r7;
        cvt.u64.u32     %rd41, %r71;
        add.s64         %rd42, %rd41, %rd20;
        shl.b64         %rd43, %rd42, 2;
        add.s64         %rd44, %rd22, %rd43;
        ld.global.v4.f32        {%f507, %f508, %f509, %f510}, [%rd44];
        mul.f32         %f515, %f507, %f667;
        mul.f32         %f516, %f771, %f668;
        mul.f32         %f517, %f770, %f668;
        mul.f32         %f518, %f769, %f668;
        fma.rn.f32      %f519, %f510, %f667, %f518;
        fma.rn.f32      %f520, %f509, %f667, %f517;
```

Overview

- ❑ Advanced CUDA Constructs
- ❑ Libraries
- ❑ Examples of Parallelism in Practice
- ❑ Tooling
- ❑ **Alternative Hardware Architectures**

MLX (Apple)

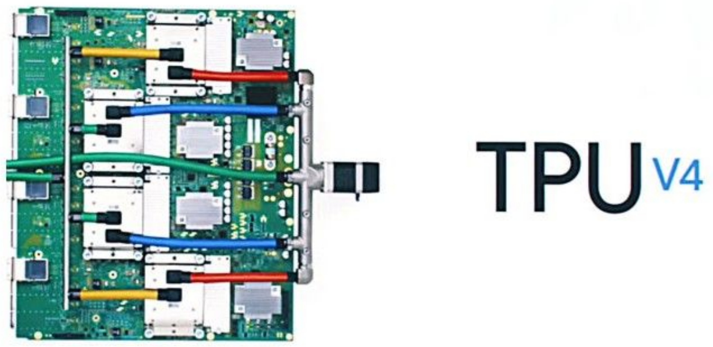
- MLX is Apple's lightweight array framework providing NumPy-like APIs with JIT compilation for CPU/GPU acceleration on Apple Silicon.
- Uses Metal Performance Shaders (MPS) and Metal compute kernels under the hood, mapping ops to tile-optimized GPU pipelines rather than CUDA-style SMs.
- Supports automatic differentiation, fused kernels, and graph-level optimizations to reduce memory traffic and launch overhead.

MI-Series (AMD)

- AMD Instinct accelerators optimized for HPC and AI workloads.
- Provide high HBM bandwidth and large memory capacity for large-scale parallel computation.
- Support ROCm software stack with HIP for CUDA-portable GPU programming.
- Current flagship models include MI210, MI250X, and MI300 series accelerators used in large HPC systems.

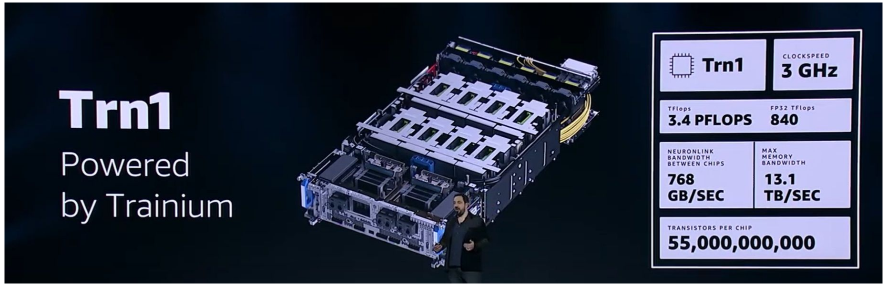
TPU (Google)

- ❑ TPUs are Google's domain-specific accelerators optimized for large-scale matrix multiplication.
- ❑ They use systolic arrays to maximize data reuse and throughput for ML workloads.
- ❑ TPU memory hierarchy emphasizes high-bandwidth on-chip buffer access over large off-chip DRAM.
- ❑ Production generations include TPU v1–v5p, with v5e/v5p powering modern large-model training clusters.



Trainium (Amazon)

- ❑ AWS Trainium is a custom accelerator optimized for large-scale deep learning training.
- ❑ Uses NeuronCore v2 architecture with high memory bandwidth and dataflow-style execution.
- ❑ Integrates tightly with AWS EC2 Trn1/Trn2 instances for scalable distributed training.
- ❑ Trainium family includes Trainium (training) and Inferentia/Inferentia2 (inference) chips.


Trn1
Powered by Trainium
Trn1
CLOCKSPEED 3 GHz
TFlops 3.4 PFLOPS
FP32 TFlops 840
NEURONLINK BANDWIDTH BETWEEN CHIPS 768 GB/SEC
MAX MEMORY BANDWIDTH 13.1 TB/SEC
TRANSISTORS PER CHIP 55,000,000,000

Gaudi (Intel)

- Intel Gaudi is a purpose-built AI accelerator optimized for deep learning training.
- Architecture emphasizes high-bandwidth on-chip SRAM and integrated RoCE NICs for scale-out.
- Supports BF16, FP8, and mixed-precision training with high throughput.
- Product line includes Gaudi, Gaudi2, and Gaudi3 accelerators used in commercial AI clusters.

Other Potential Hardware in the Future

- ❑ Qualcomm
- ❑ Broadcomm/OpenAI
- ❑ Cerebras
- ❑ …