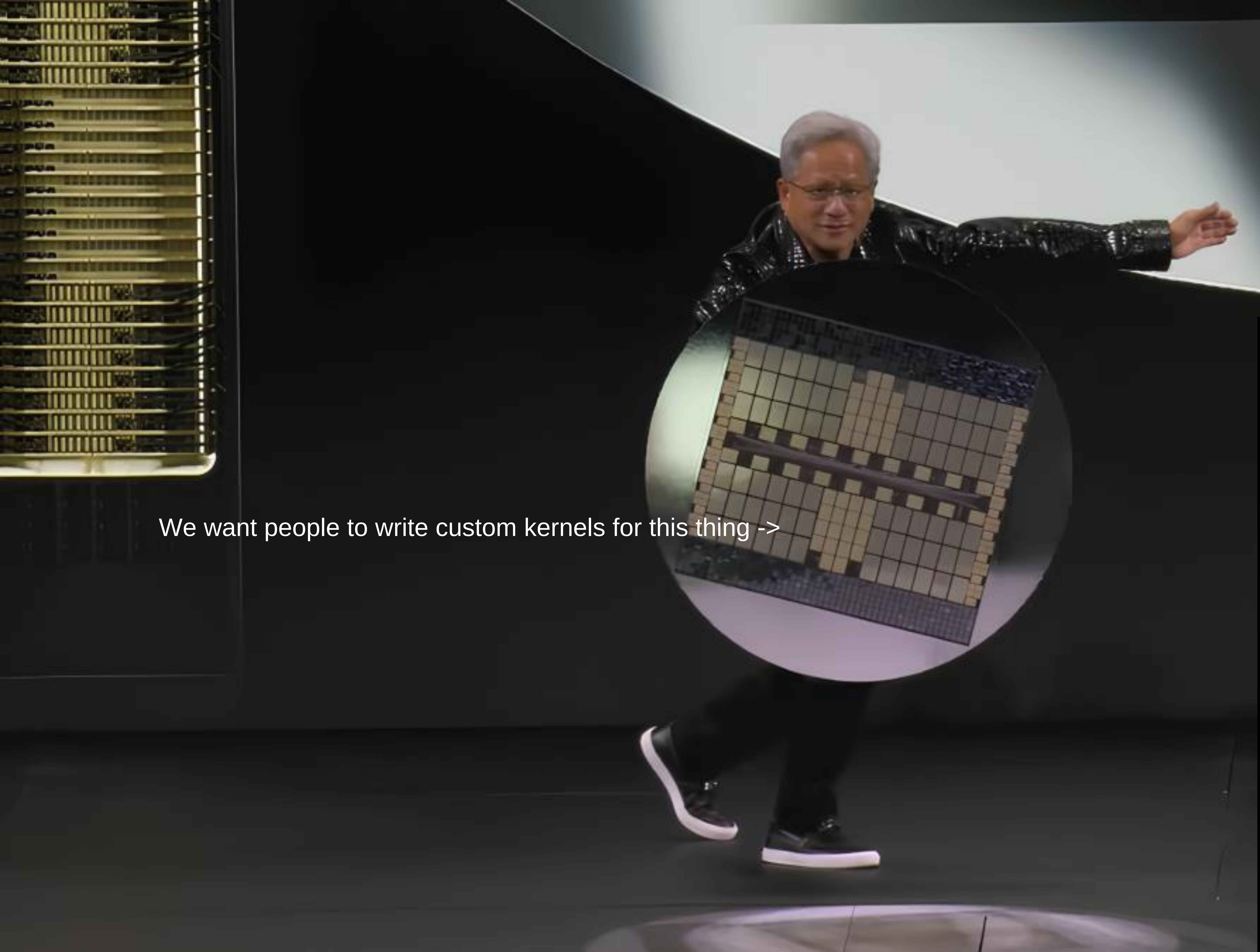
The Present and Future of CUTLASS Tensor Core Programming

Vijay Thakkar, Senior Architect

SemiAnalysis Blackwell Hackathon - 2025/03/16



We want people to write custom kernels for this thing ->

NVIDIA Blackwell Architecture

New Blackwell Hardware Features at a glance

Blackwell Tensor Cores – tcgen05

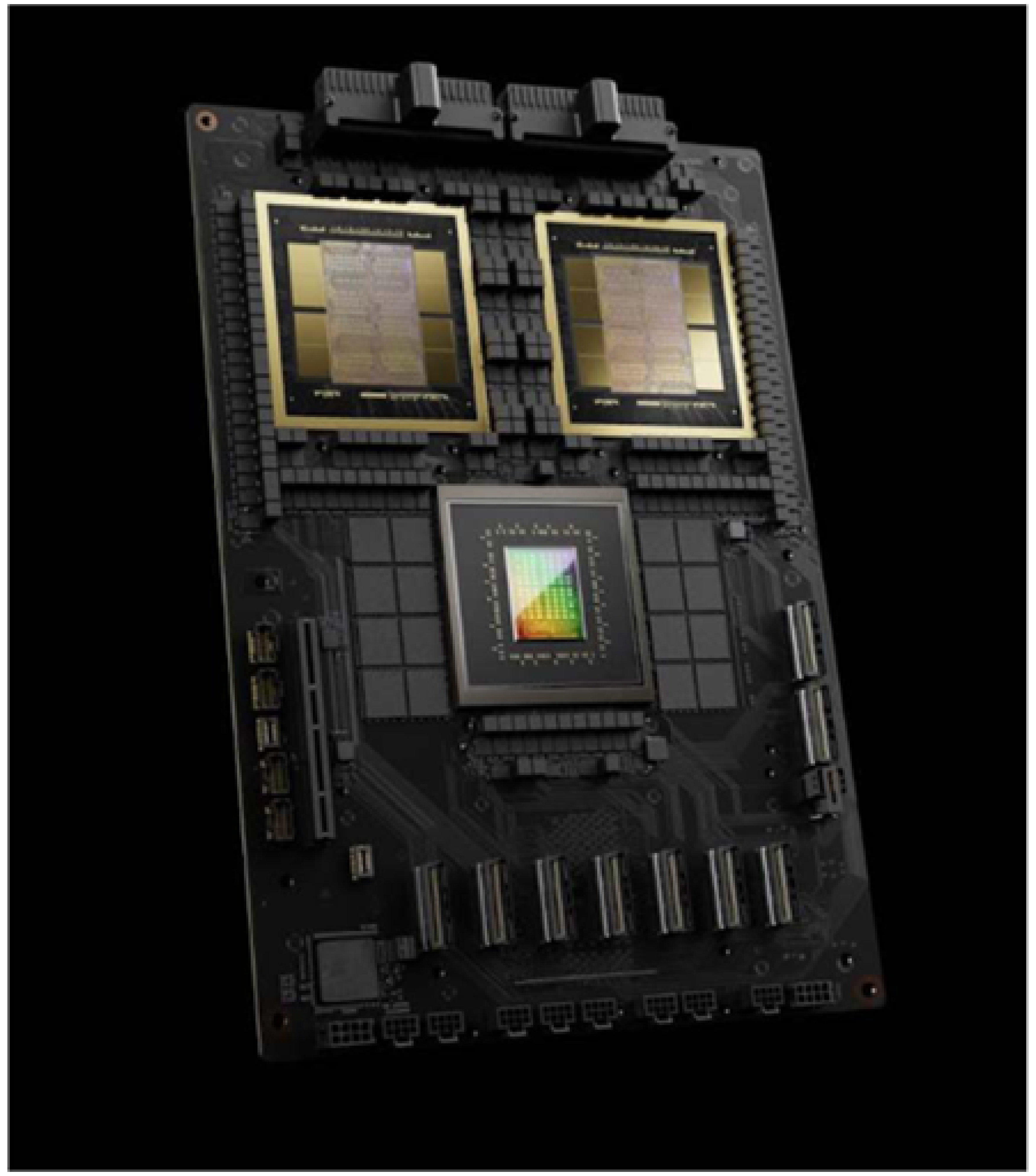
- 2x throughput vs Hopper Tensor Cores at ISO clocks.
- Expanding Tensor Core execution to two SMs
- Fully asynchronous Tensor Core programming model
- Support for new 8b (MXFP8), 6b (MXFP6) and 4b (MXFP4) micro-scaled types
  - MXFP8 / MXFP6 - 2x throughput vs Hopper FP8 at ISO clocks
  - MXFP4 - 4x throughput vs Hopper FP8 at ISO clocks

Tensor Memory (TMEM)

- New memory on each SM with same capacity as the Register File.
- TMEM based Tensor Core inputs and outputs; freeing registers for SIMT cores.

New Scheduling Capabilities

- Ability to programmatically fetch Thread Block Clusters.
- Ability to launch CUDA grids with two Thread Block Cluster configurations.



NVIDIA GB200 Superchip
Two Blackwell GPUs and One Grace CPU

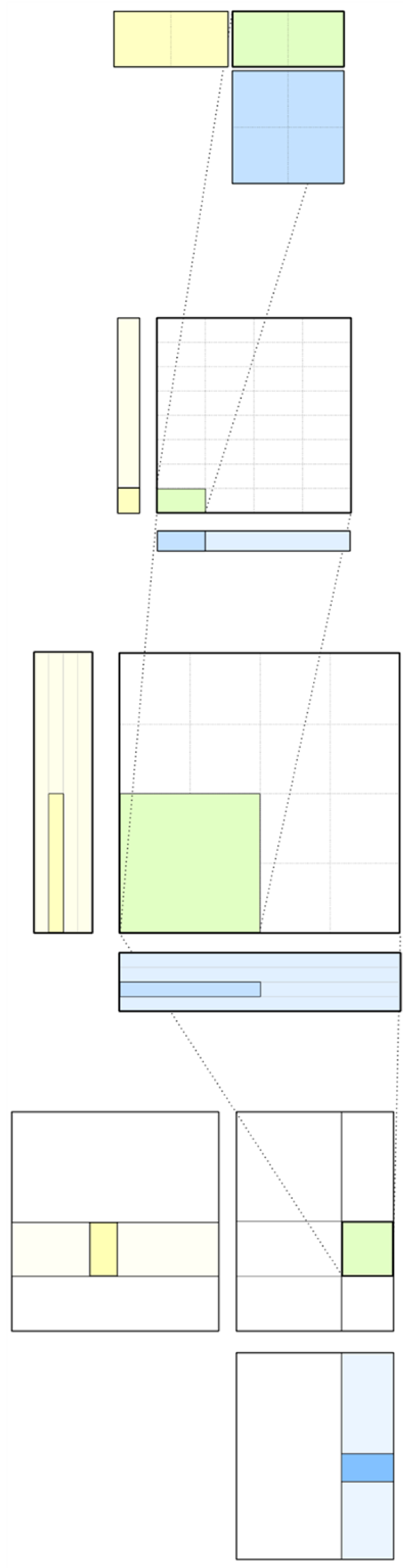
CUTLASS

CUDA C++ Template Library for High Performance Linear Algebra

Tensor computations at all scopes and scales, decomposed into their "moving parts"

Open source: https://github.com/NVIDIA/cutlass

- **7K+ stars, 4M+ clones/month, 100+ contributors**, and many active users
- Designed for off the shelf kernels AND custom kernel writers
- Peel away the layers as you need control
- Foundation of many OSS kernels such as FlashAttention 2 & 3, Machete, DeepSeek, Marlin etc.

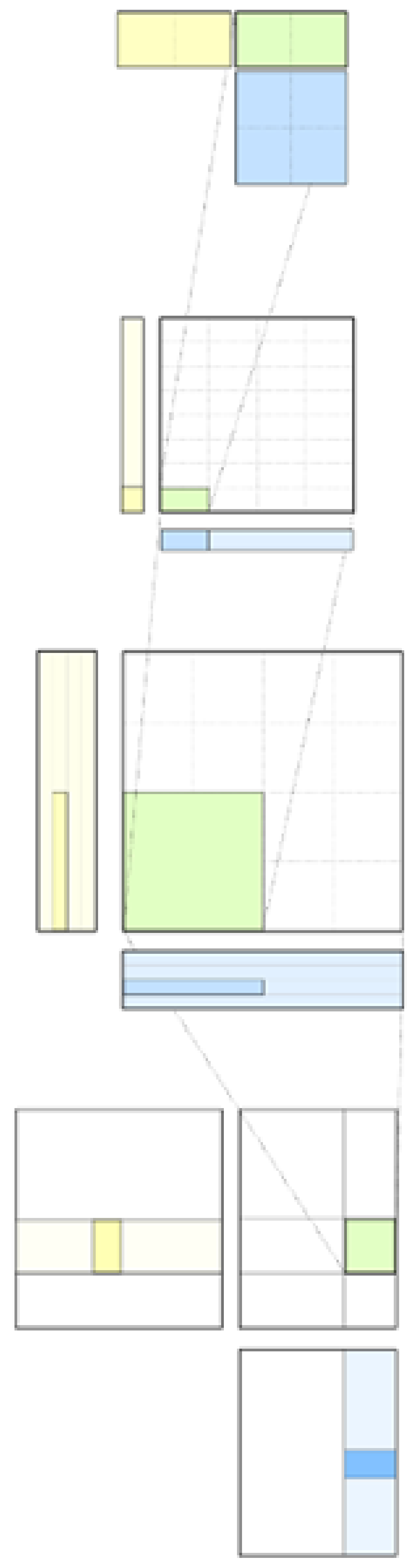


No changes at all for Blackwell!

CUTLASS 3 Conceptual Hierarchy

CUTLASS 3 computation hierarchy is not centered around the hardware hierarchy

- **Atom layer:** Architecture instructions and associated meta-information
  - Smallest set of threads and values that must participate in an architecture accelerated specified math/copy op
- **Tiled MMA/Copy**: Spatial Microkernel layer
  - **Describes the complete spatial tiling of a math/copy operation (across threads and data)**
  - Write canonical loops with arch specific instructions.
- **Collective layer:** Temporal Microkernel layer
  - **Describes the complete temporal tiling of spatial microkernels and computing one output tile**
  - Abstract complex arch-specific synchronization, warp-specialization, pipelining, and instruction interleaving
- **Kernel layer**: Outermost loops around collectives
  - Conceptually: A collection of all threadblock/clusters in the grid
  - Responsible for load balancing across tiles, thread marshalling, grid planning, and arguments construction
- **Device layer:** host side setup and interface



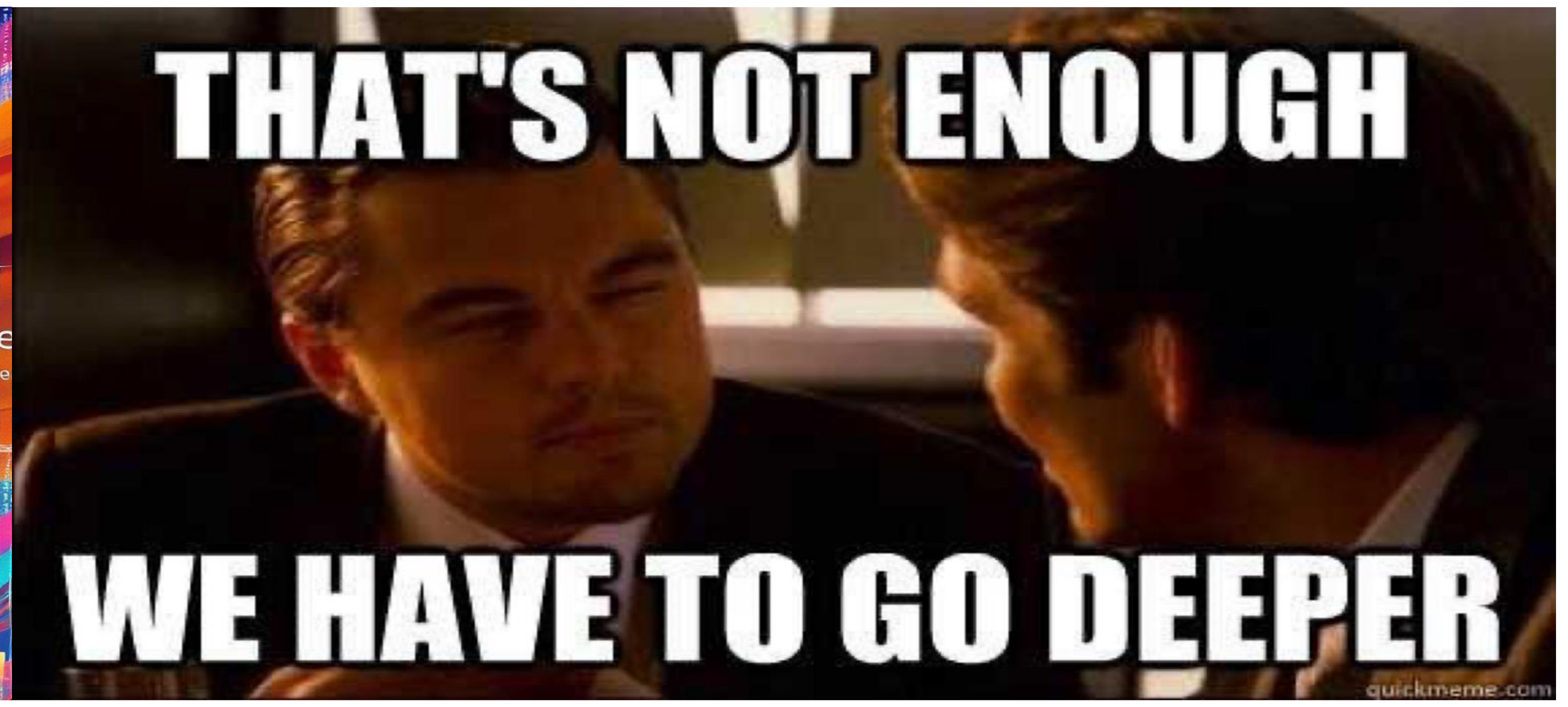
New builder features for Blackwell

Dynamic datatypes

- Some combinations of data types can be type erased
- These data types can configure the kernel at runtime
- The exact type encoding becomes a kernel argument
- Great for reducing binary size and compile times
- Does not have any performance penalty
- Static types can still be used

```
using CollectiveOp = typename collective::CollectiveBuilder<
          arch::Sm90, arch::OpClassTensorOp,
          float_e5m2_t, LayoutA, 8,
          float_e4m3_t, LayoutB, 8,
          float,
          Shape<_128,_128,_64>, Shape<_1,_2,_1>,
          gemm::collective::StageCountAuto,
          gemm::collective::KernelScheduleAuto
       >::CollectiveOp;
```

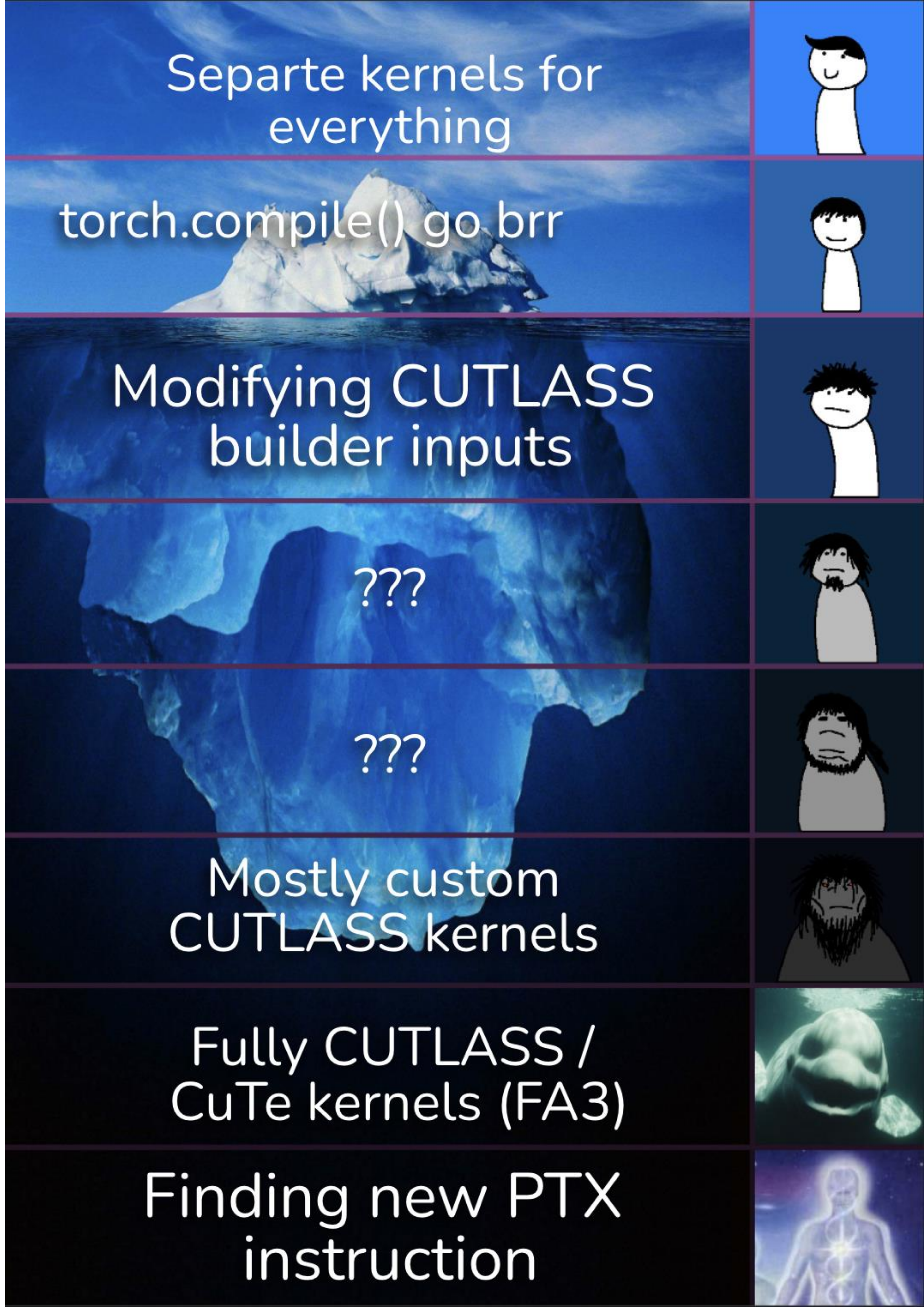
```
using CollectiveOp = typename collective::CollectiveBuilder<
          arch::Sm100, arch::OpClassTensorOp,
          type_erased_dynamic_float8_t, LayoutA, 8,
          type_erased_dynamic_float8_t, LayoutB, 8,
          float,
          Shape<_128,_128,_64>, Shape<int,int,_1>,
          gemm::collective::StageCountAuto,
          gemm::collective::KernelScheduleAuto
       >::CollectiveOp;
```



Custom kernel iceberg today

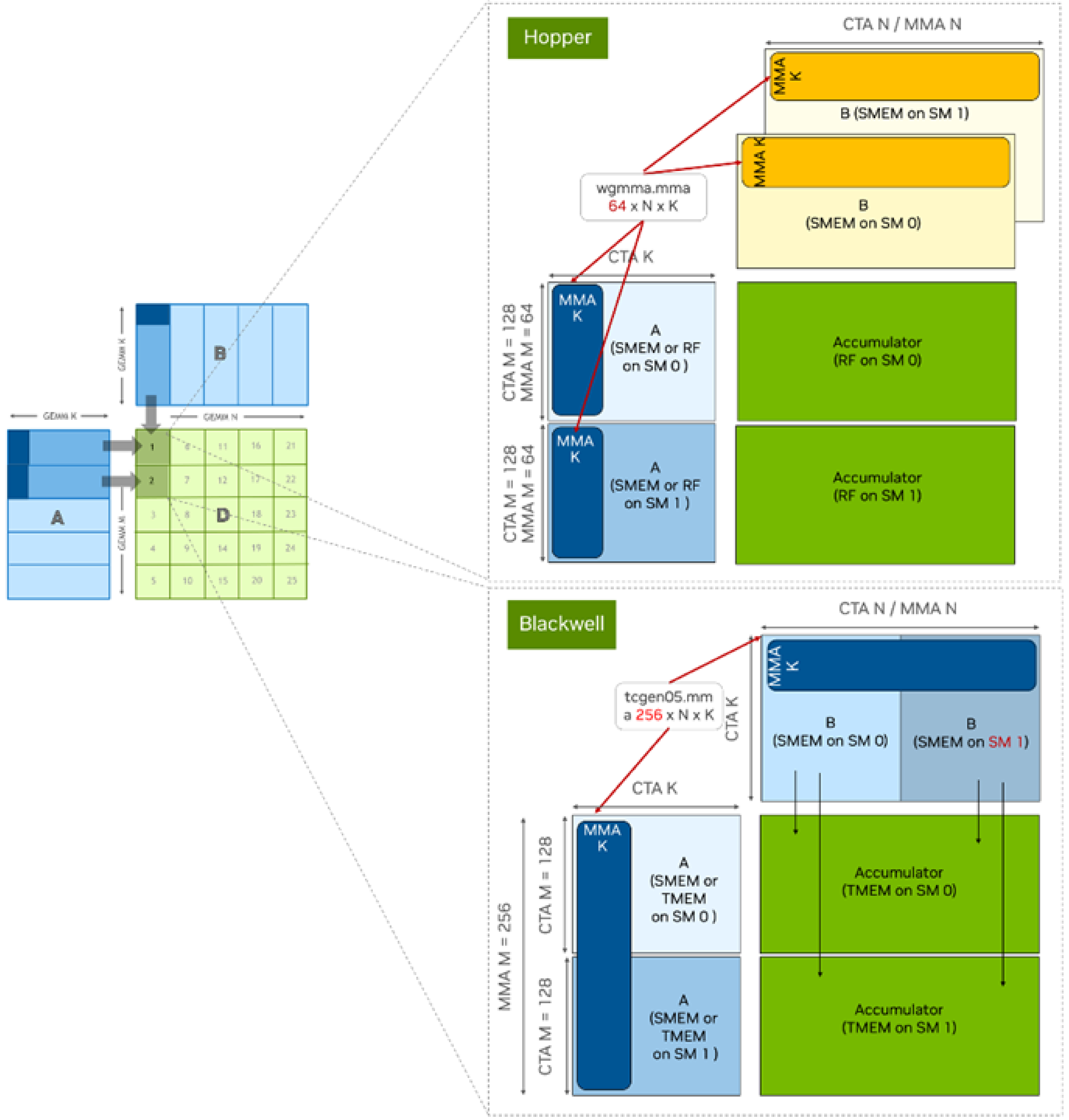
Large unaddressed chasm

- Without hardware aware custom kernels we leave a lot on the table
- Automatic compilers are still not “there” yet
- DeepSeek levels of co-design are a differentiator
- Almost certainly require custom kernels


Separte kernels for everything
torch.compile() go brr
Modifying CUTLASS builder inputs
???
???
Mostly custom CUTLASS kernels
Fully CUTLASS / CuTe kernels (FA3)
Finding new PTX instruction

Blackwell Tensor Core

Expanding Tensor Core execution to 2 SMs



- Blackwell expands Tensor Core instruction to 2 SMs.

- Pairs of 2x1 CTAs issue MMA across 2 SMs
  - 2x1 cluster → 1 CTA pair
  - 2x2 cluster → 2 CTA pairs in 1x2 layout
  - 4x4 clusters → 8 CTA pairs in 2x4 layout

- B matrix data is shared across 2 SMs; each SM provides one half.

- A matrix and accumulator is split evenly, each SM provides one half.

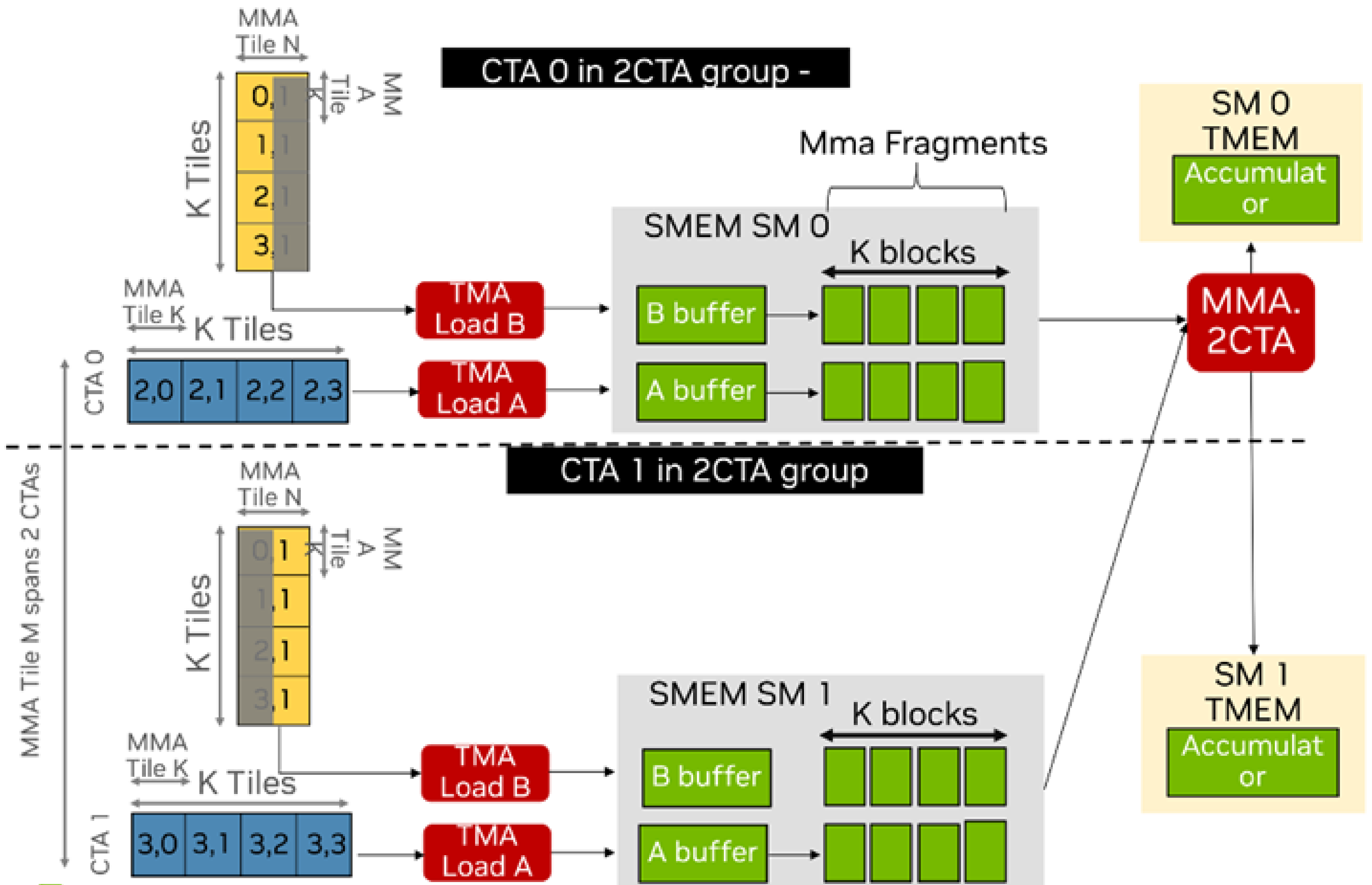
- CTA 0 in the CTA pair is the “leader” CTA, and elects 1 thread issue the MMA for both CTAs.

MMA.2SM + TMA.2SM

Blackwell GEMM

- Execute the **`cute::gemm`** on the leader CTA.

That's it!



```cpp
// Construct the MMA grid coordinate from the CTA grid coordinate
auto mma_coord_vmnk = make_coord(
  blockIdx.x % size<0>(cta_layout_vmnk), // Peer CTA coordinate
  blockIdx.x / size<0>(cta_layout_vmnk), // MMA-M coordinate
  blockIdx.y,                            // MMA-N coordinate
  _);                                    // MMA-K coordinate

auto mma_coord = select<1,2,3>(mma_coord_vmnk);
// (MmaTile_M,MmaTile_K,Tiles_K)
Tensor gA = local_tile(mA, mma_tiler, mma_coord, Step<_1, X,_1>{});
// (MmaTile_N,MmaTile_K,Tiles_K)
Tensor gB = local_tile(mB, mma_tiler, mma_coord, Step< X,_1,_1>{});
// (MmaTile_M,MmaTile_N)
Tensor gC = local_tile(mC, mma_tiler, mma_coord, Step<_1,_1, X>{});

auto mma_v = get<0>(mma_coord_vmnk);
ThrMMA mma = tiled_mma.get_slice(mma_v); // Use Peer CTA coordinate
Tensor tCgA = mma.partition_A(gA); // (MmaA,NumMma_M,NumMma_K,Tiles_K)
Tensor tCgB = mma.partition_B(gB); // (MmaB,NumMma_N,NumMma_K,Tiles_K)
Tensor tCgC = mma.partition_C(gC); // (MmaC,NumMma_M,NumMma_N)

// Construct MMA Fragments (SMEM Descriptors + TMEM Tensor)
Tensor tCrA = mma.make_fragment_A(tCsA); // (1,NumMma_M,NumMma_K,Tiles_K)
Tensor tCrB = mma.make_fragment_B(tCsB); // (1,NumMma_M,NumMma_K,Tiles_K)
Tensor tCtC = mma.make_fragment_C(tCgC); // (C,NumMma_M,NumMma_N)

uint32_t elect_one_cta = get<0>(cta_in_cluster_coord_vmnk) == 0;

for (int k_tile = 0; k_tile < size<3>(tCgA); ++k_tile)
{
  copy(tma_atom_A.with(tma_barrier), tAgA(_,k_tile), tAsA);
  copy(tma_atom_B.with(tma_barrier), tBgB(_,k_tile), tBsB);
  // TMA sync

  if (elect_one_cta)
    gemm(tiled_mma, tCrA, tCrB, tCtC);
  // MMA sync
}
```

cutlass/examples/cute/tutorial/04_mma_tma_2sm_sm100.cu

GTC Deep dive into Blackwell and CUTLASS details

Programming Blackwell Tensor Cores with CUTLASS [**S72720**]

- Come to out GTC talk for a Blackwell deep dive!!

- Deep dive into all new features of Blackwell and how to use them
  - 2SM MMA + TMA
  - TMEM
  - CLC scheduler
  - Preferred cluster shapes
  - New warp-specialized kernel recipes

- Many SOL kernels already available in `cutlass/examples`
  - **Attention**, GEMMs with various dtypes/fusions etc, different Blackwell features

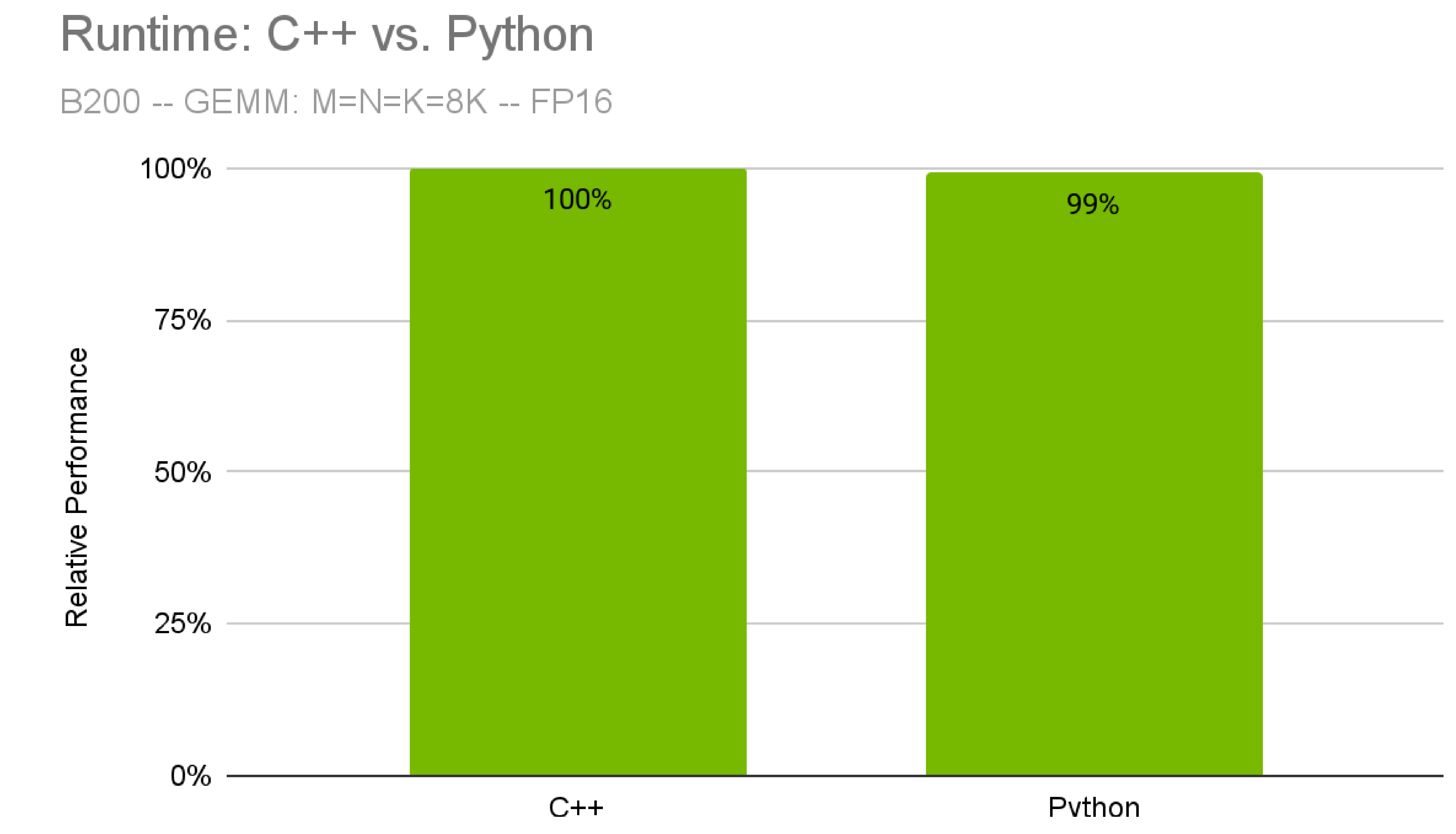
- A new series of CuTe tutorials specifically for Blackwell
  - Look out for `cutlass/examples/cute/tutorial/*sm100.cu`

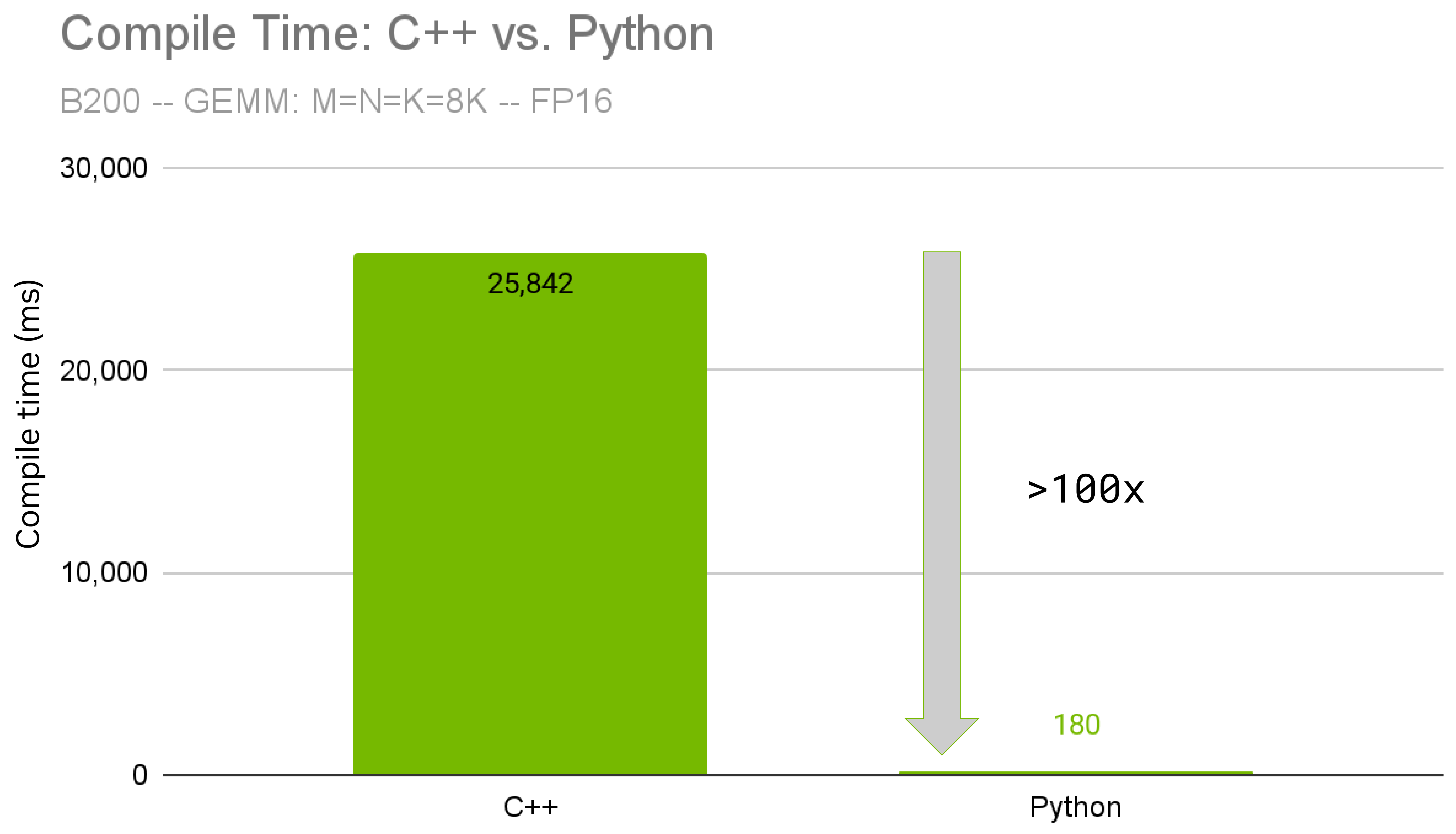
But wait... where is the future part?

CUTLASS 4.0 is coming!

A shift to Python DSLs

- Performance parity with CUTLASS C++
- Dramatically reduced compile times compared to C++
- Much lower barrier to entry and usage
- Native integration with existing Python ecosystem
- Simpler for automatic LLM based kernel generation





CUTLASS 4.x

Hierarchical exposure like CUTLASS C++

CuTe DSL (releasing at GTC)

- Intended to be a 1:1 analog of CuTe C++
- Designed for both productivity and peak performance
- Low level control with a clean programming model
- Support for all architectures starting with Ampere

More in the future ...

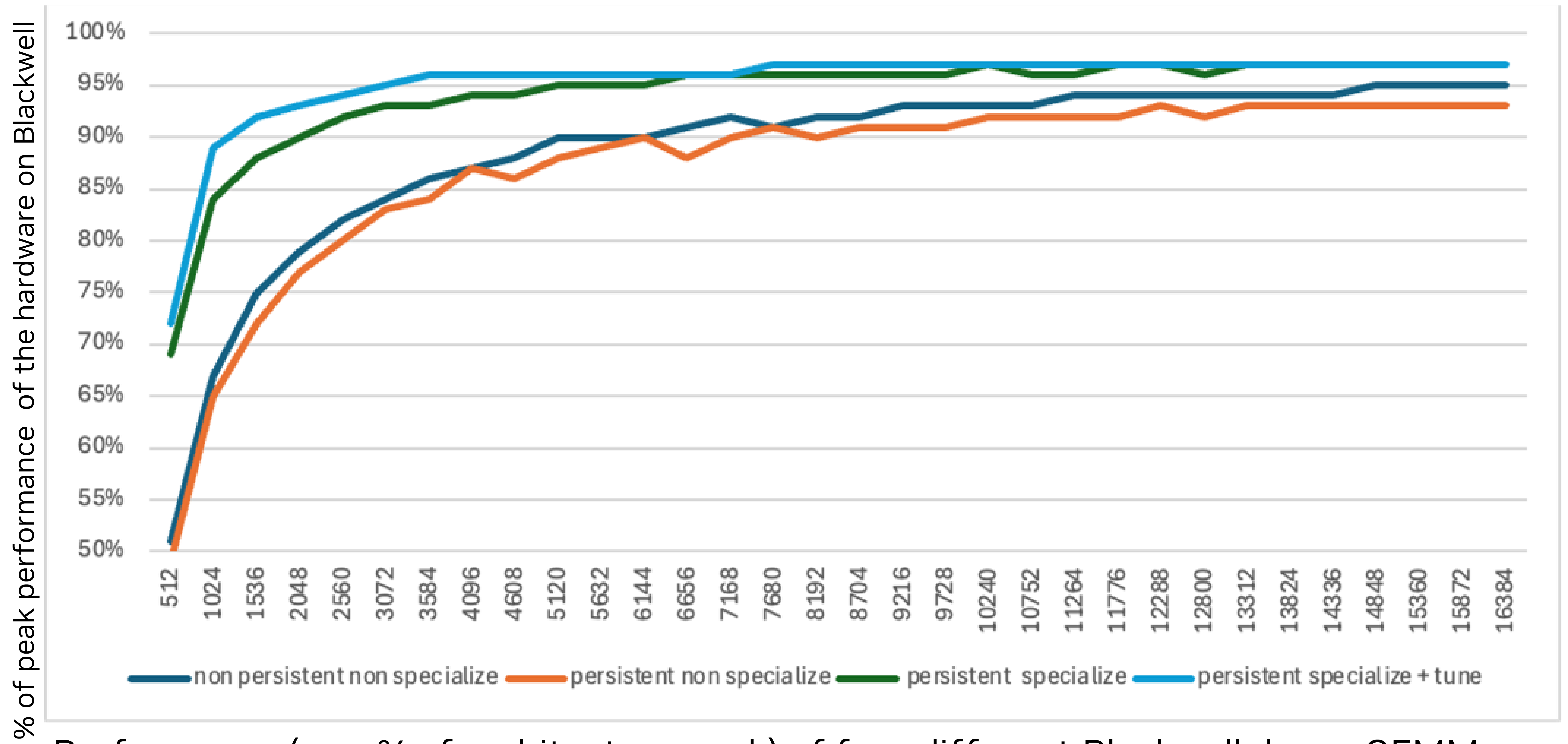
- For higher levels of abstractions

```python
@cute.kernel
def vector_add_kernel(
      gA: cute.Tensor,
      gB: cute.Tensor,
      gC: cute.Tensor,
      cC: cute.Tensor,  # coordinate tensor
      shape: cute.Shape,
      tv_layout: cute.Layout):
    tidx, _, _ = cutlass.nvvm.thread_idx()
    bidx, _, _ = cutlass.nvvm.block_idx()

    # slice for CTAs (logical id -> address)
    cta_coord = ((None, None), bidx)
    ctaA = gA[cta_coord]    # (TileM,TileN)
    ctaB = gB[cta_coord]    # (TileM,TileN)
    ctaC = gC[cta_coord]    # (TileM,TileN)
    ctaCrd = cC[cta_coord]  # (TileM,TileN)

    # declare the atoms which will be used later for memory copy
    copy_atom_ldg = cute.make_copy_atom(cute.nvgpu.CopyUniversalOp(), gA.element_type)
    copy_atom_stg = cute.make_copy_atom(cute.nvgpu.CopyUniversalOp(), gC.element_type)

    tiled_copy_A = cute.make_tiled_copy_tv(copy_atom_ldg, tv_layout[0], tv_layout[1])
    thr_copy_A = tiled_copy_A.get_slice(tidx)
    thrA = thr_copy_A.partition_S(ctaA)
      # repeat above for B

    tiled_copy_C = cute.make_tiled_copy_tv(copy_atom_stg, tv_layout[0], tv_layout[1])
    thr_copy_C = tiled_copy_C.get_slice(tidx)
    thrC = thr_copy_C.partition_S(ctaC)

    # allocate fragments for gmem->rmem
    frgA = cute.make_fragment(thrA.element_type, thrA.shape)
    frgB = cute.make_fragment(thrB.element_type, thrB.shape)
    frgC = cute.make_fragment(thrC.element_type, thrC.shape)

    thrCrd = thr_copy_C.partition_S(ctaCrd)
    # Move data to reg address space
    cute.copy(copy_atom_ldg, thrA, frgA)
    cute.copy(copy_atom_ldg, thrB, frgB)

    # Load data before use. The compiler will optimize the copy and load
    result = frgA.load() + frgB.load()

    # Save the results back to registers and copy the results back to gmem
    frgC.store(result)
    cute.copy(copy_atom_stg, frgC, thrC)
```

Introduction to CuTe Python DSL

A pythonic analog to CuTe C++ built on CuTe IR

- Reference/buffer semantics for granular control
- Correctness by construction
- Support for both dynamic and static shapes (and mixed)
- Support for both host and device code JIT
- Support for TMA as a first class citizen
- Metaprogramming looks just like imperative programming
- NumPy-style comprehensive documentation on docs.nvidia.com



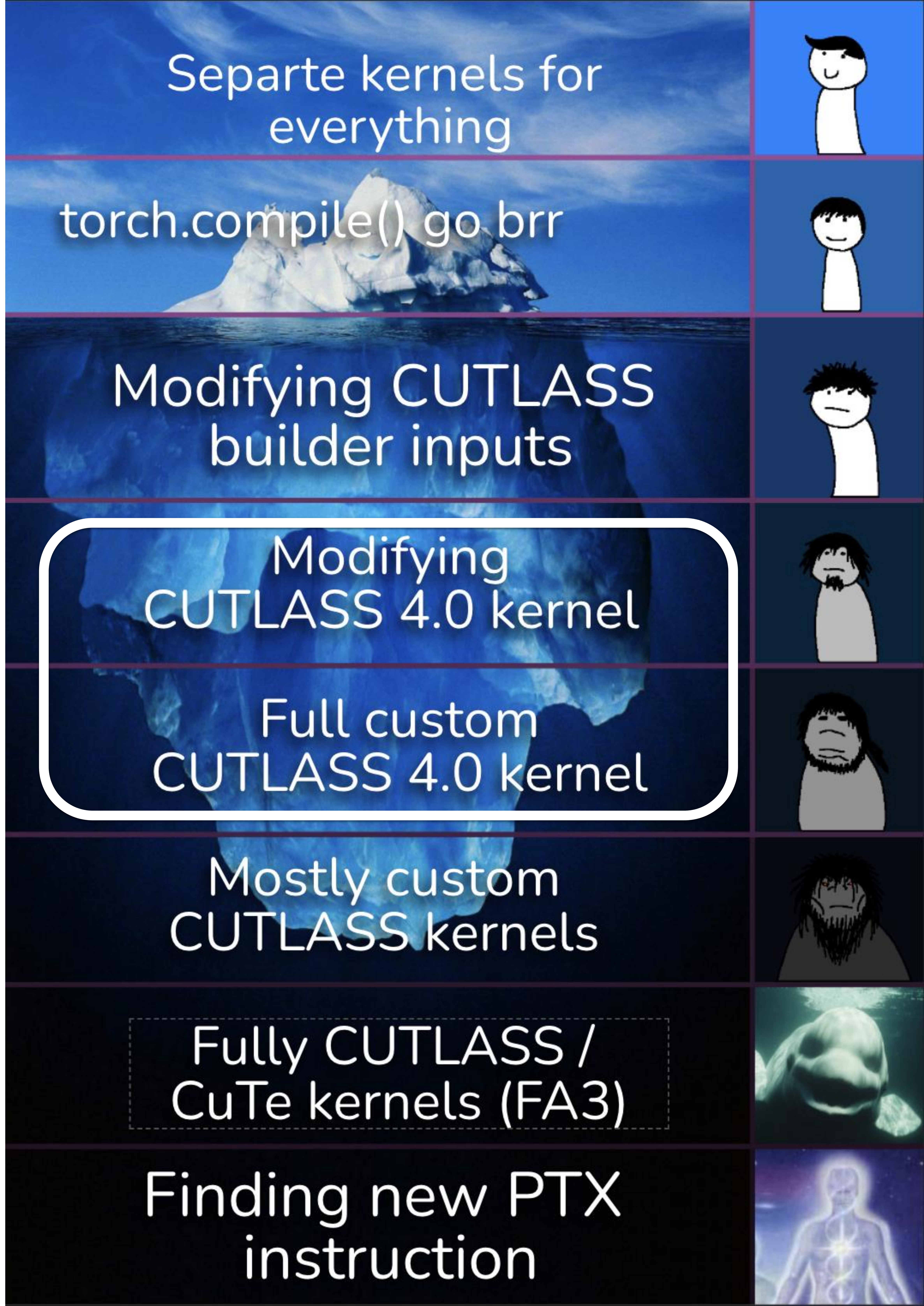
Performance (as a % of architecture peak) of four different Blackwell dense GEMM kernels developed in CuTe DSL with increasing levels of optimization

Key Advantages of CuTe DSL

- Parity with CuTe C++ in interfaces and concepts
  - Full freedom to design novel kernels

- No C++ templates!
  - Blazing fast compile times
  - Much better debug messages
  - Intuitive metaprogramming that looks imperative
  - Much faster prototyping loop
  - Much wider auto-tuning space

- Significantly easier integration into python frameworks
  - JIT compilation with caching for reduced overhead
  - No NVCC or CUDA toolkit dependencies
  - Support for DLPack and Torch tensor formats
  - Auto-tuning and benchmarking within the framework
  - Testing code can just be written in PyTorch as well!


Separte kernels for everything
torch.compile() go brr
Modifying CUTLASS builder inputs
Modifying CUTLASS 4.0 kernel
Full custom CUTLASS 4.0 kernel
Mostly custom CUTLASS kernels
Fully CUTLASS / CuTe kernels (FA3)
Finding new PTX instruction

Getting Started

- GTC talk for Python DSL: Enable Tensor Core Programming in Python With CUTLASS 4.0 [**S74639**]
- Comprehensive tutorials from “Hello World” to advanced implementations
- Interactive Jupyter notebooks for hands-on learning
- Copy-paste ready examples for quick implementation
- API documentation with clear examples and use cases

Need to know

- CUTLASS C++ (2.x and 3.x) is here to stay!
- The DSL will be available GitHub repo for issues / bug fixes
- Provided as a pip wheel with nightly builds for prompt bug fixes
- CUTLASS 4.0 will deprecate the existing python interface for instantiating device wide GEMMs

CUDA Developer Sessions

General CUDA

- S72571 - What's CUDA All About Anyways?
- S72897 - How To Write A CUDA Program: The Parallel Programming Edition

CUDA Python

- S72450 - Accelerated Python: Tour of the Community and Ecosystem
- S72448 - The CUDA Python Developer's Toolbox
- S72449 - 1001 Ways to Write CUDA Kernels in Python
- S74639 - Enable Tensor Core Programming in Python With CuTe

CUDA C++

- S72574 - Building CUDA Software at the Speed-of-Light
- S72572 - The CUDA C++ Developer's Toolbox
- S72575 - How You Should Write a CUDA C++ Kernel

Developer Tools

- S72527 - It's Easier than You Think – Debugging and Optimizing CUDA with Intelligent Developer Tools

Connect with the Experts

- CWE72433 - CUDA Developer Best Practices
- CWE73310 - Using NVIDIA CUDA Compiler Tool Chain for Productive GPGPU Programming
- CWE72393 - What's in Your Developer Toolbox? CUDA and Graphics Profiling, Optimization, and Debugging Tools
- CWE75384 – Connect with Dr. Wen-mei Hwu, Author of *Programming Massively Parallel Processors*

Multi-GPU Programming

- S72576 - Getting Started with Multi-GPU Scaling: Distributed Libraries
- S72579 - Going Deeper with Multi-GPU Scaling: Task-based Runtimes
- S72578 - Advanced Multi-GPU Scaling: Communication Libraries

Performance Optimization

- S72683 - CUDA Techniques to Maximize Memory Bandwidth and Hide Latency
- S72685 - CUDA Techniques to Maximize Compute and Instruction Throughput
- S72686 - CUDA Techniques to Maximize Concurrency and System Utilization
- S72687 - Get the Most Performance from Grace Hopper

nvidia.com/gtc/sessions/cuda-developer

NVIDIA

Thank you and happy hacking!