Clusters, Data types, inline PTX, pointers

Prateek Shukla

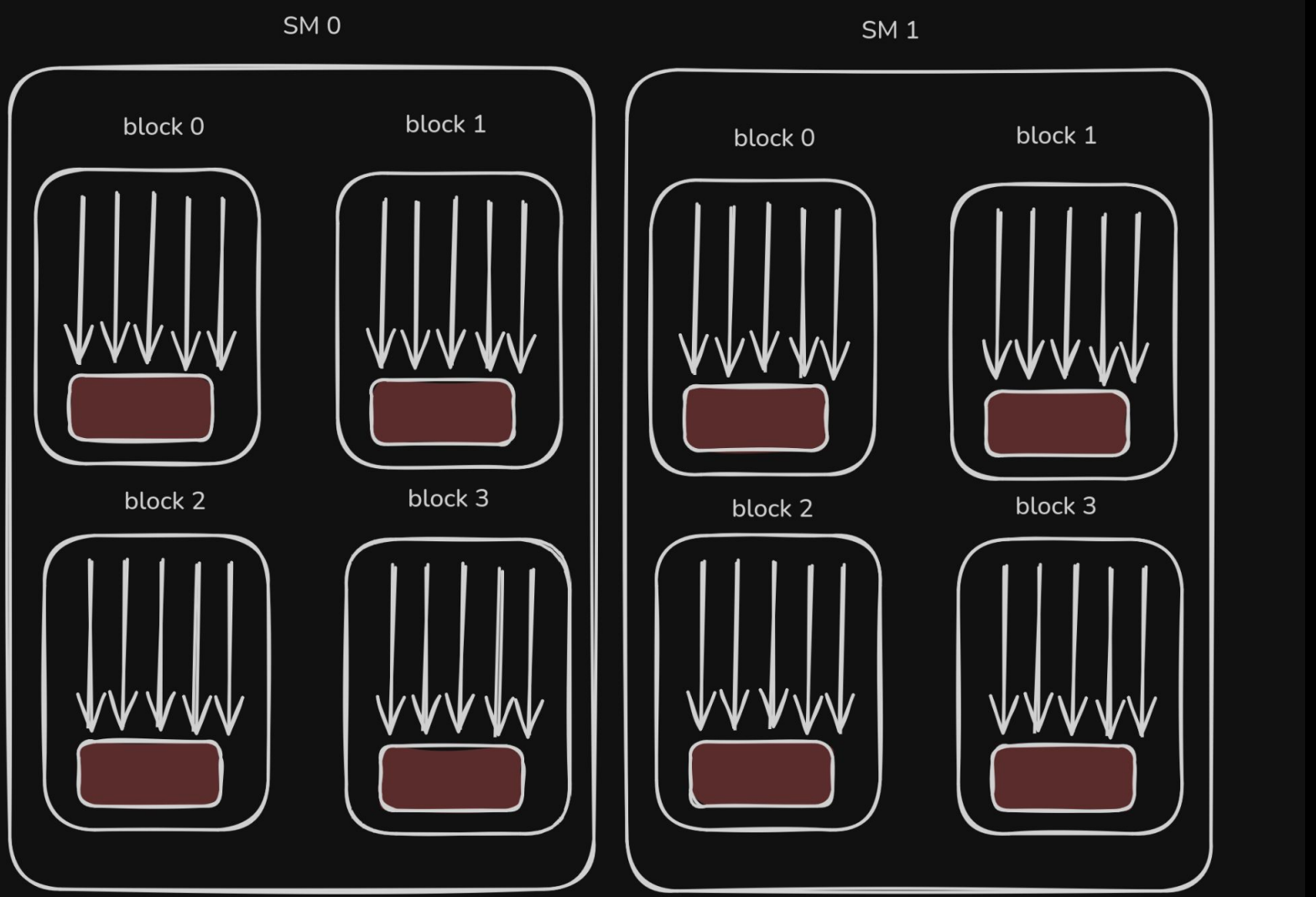
What are thread block clusters

Cluster is a collective of up to 16 thread blocks that are guaranteed to be co-scheduled concurrently on adjacent SMs in a GPC

This extends the traditional CUDA programming hierarchy from three levels to four explicit levels: threads → thread blocks → thread block clusters → grid

All blocks in a cluster run concurrently on SMs that are physically close together, enabling efficient cross-SM cooperation

The reason of using a cluster is for most of the time using distributed shared memory for different purpose.

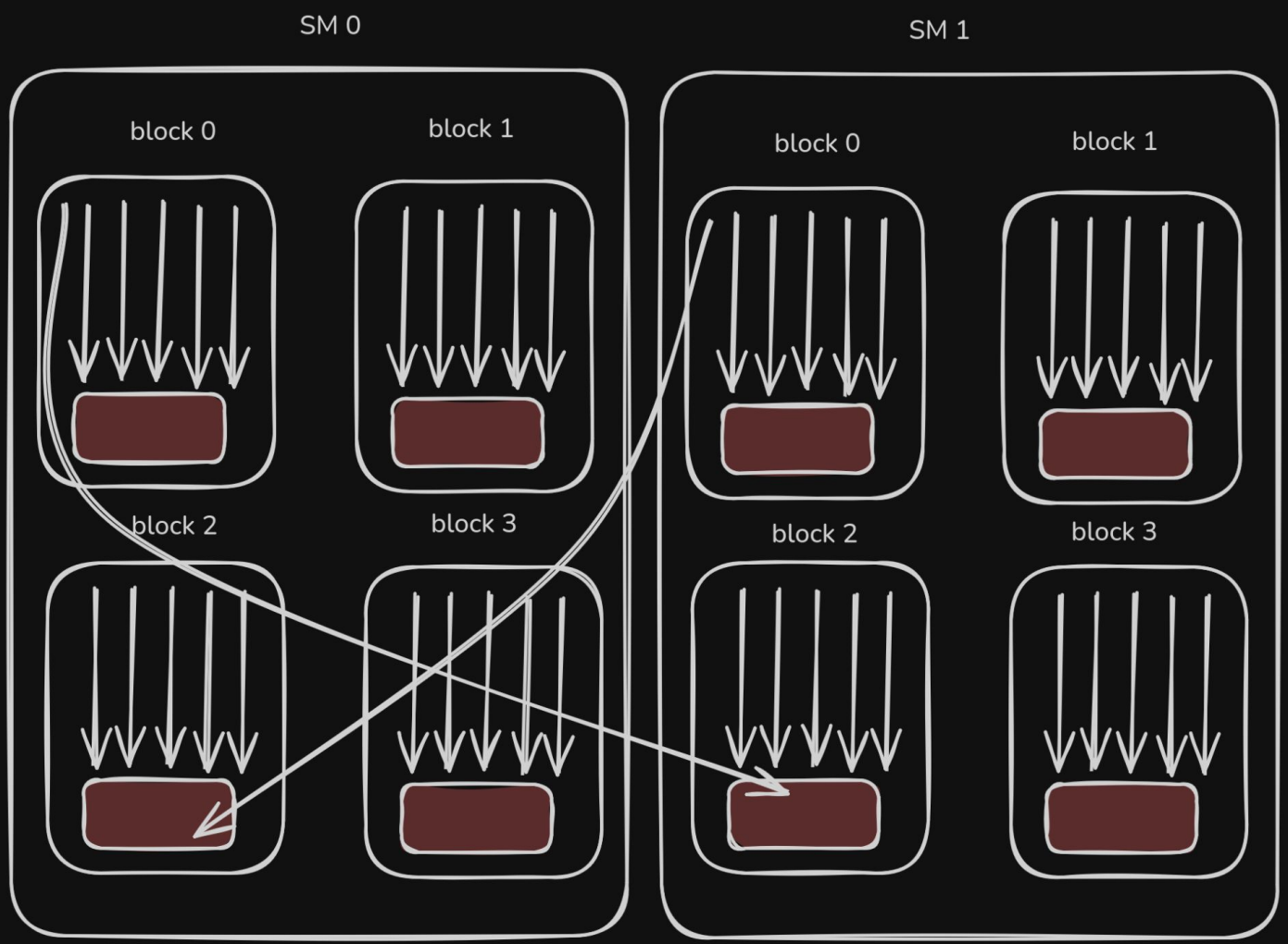

SM 0
SM 1
block 0
block 1
block 2
block 3
block 0
block 1
block 2
block 3

Each thread block is constrained by the limited shared memory and computational resources available on a single SM.

If the thread wants to access more data it needs to go to global memory to fetch it which is expensive

For operations like top-k or matrix multiplication that require accessing data from thread blocks executing on different SMs, thread block clusters provide a solution.

They enable algorithms to trade modest memory bandwidth for significantly expanded data accessibility across multiple SMs



All that we do is take all the thread blocks and pool some of their memory. Now threads from different blocks can access the data which they need without needing to access the global memory

This pool is not pool in the sense that all the threads just pour all their shared memory allocated into a single thing. Each block still owns its piece of shared memory, the difference is that now the threads can go down and access other block's memory

Generally this pool is small and constrained inside a GPC

A few important points

A threadblock still runs on a single SM - the one-to-one relationship hasn't changed

SMs can run multiple threadblocks simultaneously - this was always possible and remains true

Clusters are a software concept, GPC is a hardware concept

No automatic assignment: You must explicitly define clusters in code; the scheduler won't group blocks automatically

On cluster sizes

With a cluster size of 8 thread blocks, the hardware can fit two clusters per GPC, utilizing roughly 16 SMs per GPC efficiently

Mention that using cluster size > 8 requires explicitly setting cudaFuncAttributeNonPortableClusterSizeAllowed at kernel launch. For size 8 or smaller, the code remains portable

With a cluster size of 16 thread blocks, only one cluster fits per GPC, leaving approximately 1-2 SMs idle per GPC or roughly 18 SMs unused across the entire GPU

Cluster size 2 have been found a lot more optimal as there is a hidden synchronization cost to larger clusters (87 cycles for size 4+ and 150 cycles for size 16)

What is Distributed Shared Memory

Feature in H100 which allows a direct memory accesses between SMs within a thread block cluster

H100 implements a dedicated SM-to-SM network for clusters that provides fast, low-latency access to remote shared memory which enables SMs to perform load, store and Atomic operations across the shared memory of other SMs

DSMEM can be used simultaneously with L2 cache accesses. This allows applications to utilize the combined bandwidth of both pathways when communicating data between SMs

Multicast and TMA with DSMEM

We can use TMA for asynchronous copy operations with DSMEM

One of the most important features of DSMEM are multicast operations that deliver data to multiple SMs' shared memory simultaneously

TMA multicast bypasses the SM-to-SM network bottleneck we discussed earlier. Instead of threads explicitly writing and reading across DSMEM (causing congestion and synchronization overhead), a single thread per cluster can issue a TMA multicast operation to distribute data to all SMs at once

Creating thread block clusters

You can define cluster dimensions at compile time using the __cluster_dims__ attribute in your kernel declaration

```
__global__ void __cluster_dims__(2, 2, 1) cluster_kernel(float* input, float* output) {
    // Kernel code here
}
```

Once defined you can launch the kernel normally but the grid dimensions must be a multiple of cluster size

```
dim3 threadsPerBlock(vx: 16, vy: 16);
dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
cluster_kernel<<<numBlocks, threadsPerBlock>>>(input, output);
```

Cluster handlers

```
#include <cooperative_groups.h> (cooperative groups help manage clusters)

namespace cg = cooperative_groups;

cg::cluster_group cluster = cg::this_cluster();
```

creates a handle to the thread block cluster that the currently executing thread belongs to

```
int* remote_smem = cluster.map_shared_rank(smem, target_block_rank);

remote_smem[idx] = value;

unsigned int cluster_size = cluster.num_blocks();

unsigned int cluster_rank = cluster.block_rank();
```

But we are not gonna use it a lot. We will generally use mapa to convert the shared memory address to address in clusters

PTX special registers

These provide information about thread blocks in the cluster

`%cluster_ctaid`: CTA ID within the cluster

`%cluster_nctaid:` the dimensions of the cluster

`%cluster_ctarank`: linearized rank of the CTA in the cluster

`%cluster_nctarank`: Total number of CTAs in the cluster

`%is_explicit_cluster`: Differentiates explicit vs implicit (1×1×1) cluster launch

Parallel Thread Execution

PTX is the assembly language of Nvidia GPUs and serves as an Instruction set architecture in the CUDA ecosystem

When CUDA C++ files are compiled they translate into PTX which then jit compiles into SASS which is then assembled into binaries

We are learning PTX because it enables direct access to low-level GPU features that are not directly exposed in CUDA C/C++ which will help us to hand optimize performance critical sections

Its widely used in a lot of famous repositories and learning PTX allows you to get to know all kinds of optimizations in highly optimized libraries

Inline PTX

Inline PTX is PTX assembly code that is embedded directly within CUDA C/C++ using asm keyword

We use inline PTX because it reduces a lot of heavy lifting in hand writing everything from scratch in PTX and allows writing PTX instructions inside c++ code

Templates enable to embed parameterized compile-time constants directly into GPU instructions which is why often times we would use templates with inline instructions

The format of PTX instructions

- `asm()` inserts the ptx code into your cuda program
- Adding `volatile` keyword after asm prevents the compiler from deleting or moving your ptx instruction
- If you don't want compiler to see the way you access memory use `memory` keyword in the clobbers section

General format of a PTX instruction

```
asm("ptx instruction" : output operands : input operands : clobbers)
```

Input and output operands

Input and output operands are the bridge between your C++ variables and the assembly instructions. They use a constraint-based syntax to tell the compiler how to map variables to registers or memory

Output operands are declared first after the instruction string, then input operands and then clobbers

Operands are numbered sequentially across outputs and then inputs

If you don’t have output you can leave fields empty like `asm(“string”:: inputs)`

Constraint modifiers

`=` - Write only

`+` - read and write only

`&` - Early clobber - prevents the compiler from using the same register for an output and a subsequent input. Critical when output is written before all inputs are consumed

```
// BAD: May fail if %0 and %1 use same register
asm("add.u32 %0, %1, %0;" : "=r"(x) : "r"(y));

// GOOD: Forces separate registers
asm("add.u32 %0, %1, %2;" : "=&r"(x) : "r"(y), "r"(x));
```

Constraints

`h` - 16 bit unsigned integers

`r` - 32 bit unsigned integer used for 32 bit addresses and unsigned integers

`l` - 64 bit unsigned integer and 64 bit addresses

`f` - 32 bit floating point number

`d` - 64 bit floating point numbers

`n` - immediate integer(the compile time constant)

Some more points about operands

We represent the operands in text order like `%0, %1, %2`

So %0 would represent the first variable after the instruction string

We represent the variables using the constraint modifier and constraints(`“+r”, “=f”, “+l”` for output and `“r”, “l”, “f”` for inputs)

Curly braces are used for local register scoping {}

```
asm volatile("{\n"
             "wgmma.mma_async.sync.aligned.m64n32k16.f32.bf16.bf16 "
             "{%0,   %1,   %2,   %3,   %4,   %5,   %6,   %7,   "
             " %8,   %9,   %10,  %11,  %12,  %13,  %14,  %15},  "
             " %16,"
             " %17,"
             " %18, %19, %20, %21, %22;\n"
             "}\n"
             : "+f"(d[0][0]), "+f"(d[0][1]), "+f"(d[0][2]), "+f"(d[0][3]),
               "+f"(d[0][4]), "+f"(d[0][5]), "+f"(d[0][6]), "+f"(d[0][7]),
               "+f"(d[1][0]), "+f"(d[1][1]), "+f"(d[1][2]), "+f"(d[1][3]),
               "+f"(d[1][4]), "+f"(d[1][5]), "+f"(d[1][6]), "+f"(d[1][7])
             : "l"(desc_a), "l"(desc_b), "n"(int32_t(ScaleD)),
               "n"(int32_t(ScaleA)), "n"(int32_t(ScaleB)),
               "n"(int32_t(TransA)), "n"(int32_t(TransB)));
```

The instruction string

- This is the string containing actual instructions which we want to run on the gpu

- Instructions can be written in a single line or multiple lines using newline

- You can use placeholders like `%0, %1` etc to represent the operands which will be replaced by the actual operands used in the assembly which we after :

```
asm volatile("wgmma.mma_async {%0, %1, %2, ...}")
```

Output operands

They come after the instruction string

Use special modifiers and constraints to represent the output

Use comma separated outputs to represent multiple outputs

If there is no output operand leave the field empty (like ::)

```
asm("add.s32 %0, %2, %3; sub.s32 %1, %2, %3;"
    : "=r"(sum), "=r"(diff)
    : "r"(a), "r"(b));
```

Input operands

Represented after output operands separated by colon

You can have either read only input operands (“n”, “r”, “f”, “l”) or read and write operands (“+n”, “+r”, “+f”) but not write only operands

You can use “+n” constraint to get the compile time operands

You can use multiple input operands by separating them with commas and putting them after the output operands

Clobbers

They tell the compiler what resources might get modified apart from the output operands

This prevents the compiler from making wrong assumption and modify the code incorrectly.

```
asm volatile("mbarrier.arrive.release.cta.shared::cta.b64 _, [%0], %1;\n"
             :
             : "r"(mbar_ptr), "r"(count)
             : "memory");
```

What are state spaces in CUDA

State spaces in CUDA refer to different memory regions that are accessible by GPU threads, each optimized for specific purposes

When writing code in PTX we need to specify for which state spaces we are launching instructions

They're not just a low-level PTX detail. They're the reason we can write high performance code by making memory placement a first class decision, not an afterthought

Register State Space (.reg)

The register memory

you need explicit .reg declarations when you require temporary registers that aren't covered by the input/output constraint system.

Spilling mechanism: When register usage exceeds hardware limits, variables automatically spill to local memory, which can significantly impact performance.

```
__device__ int cube(int x) {
    int y;
    asm("{\\n\\t"
        ".reg .u32 t1;\\n\\t"          // Declare temp register
        "mul.lo.u32 t1, %1, %1;\\n\\t"  // t1 = x * x
        "mul.lo.u32 %0, t1, %1;\\n\\t"  // y = t1 * x
        "}"
        : "=r"(y) : "r"(x));
    return y;
}
```

Global State Space (.global)

Represents the global memory

For different operations which use data from global memory

```
    asm volatile (
        "cp.async.bulk.tensor.3d.shared::cluster.global.tile.mbarrier::complete_tx::bytes.multicast::cluster"
        " [%0], [%1, {%3, %4, %5}], [%2], %6;"
        :
        : "r"(dst_ptr), "l"(tma_ptr), "r"(mbar_ptr),
        "n"(0), "r"(global_row_idx), "r"(global_col_idx/64), "h"(cluster_mask)
```

```
asm volatile (
    "cp.async.bulk.tensor.3d.global.shared::cta.tile.bulk_group"
    " [%0, {%2, %3, %4}], [%1];"
    :
    : "l"(tma_ptr), "r"(src_ptr),
    "n"(0), "r"(global_row_idx), "r"(global_col_idx / 64)
    : "memory"
);
```

Local State Space (.local)

Per-thread private memory: Each thread has its own private local memory space for storing data that doesn't fit in registers.

Rarely used

Parameter State Space (.param)

Dual-purpose usage: Serves as kernel input parameters (read-only, per-grid) and device function parameters (read/write, per-thread).

Argument passing mechanism: Used to pass arguments to kernels and functions without using the general register file.

Again not used a lot

Shared State Space (.shared)

Represents the shared memory

Cluster wide accessibility: With sub-qualifiers (::cta or ::cluster), can be accessed by threads in other CTAs within the same cluster for advanced communication

The data types

Unsigned integer types - .u8, .u16, .u32, .u64

Signed integer types - .s8, .s16, .s32, .s64

Floating point types - .f16, .f32, .f64

Raw bit patterns - .b8, .b16, .b32, .b64, .b128

Predicate type - store TRUE or FALSE values

Some important datatypes in deep learning

bf16(stored as .b16): 8-bit exponent, 7-bit mantissa (16 bits total)

e4m3, e5m2(stored as .b8): 8-bit Float Formats (4-bit exponent + 3-bit mantissa, or 5-bit exponent + 2-bit mantissa)

tf32(stored as .b32) - 32-bit format with same range as .f32 but reduced precision (≥10 bits mantissa)

4-bit Float (e2m1)

Ultra-compact: 2-bit exponent, 1-bit mantissa (4 bits total)

Packed data types

These pack multiple values into a single register for parallel operations:

.u16x2, .s16x2 - Holds two 16-bit integers in a 32-bit register (.b32)

.f16x2 - Holds two .f16s

.bf16x2 - holds two BF16

.e4m3x2 - holds two e4m3

.e5m2x2 - holds two e5m2

What are memory addresses really

The entire memory of the GPU as one infinitely long ruler. The smallest unit of memory you can have an address for is a Byte (8 bits) this is also called an atom.

The address is the distance in bytes from the start of the ruler. Byte Address is the raw integer index on the ruler

In CUDA, you rarely work with raw bytes. You work with types (float, int, bf16). When you write ptr + 1, the compiler does not move 1 byte. It moves 1 element.

$$\text{Byte Address} = \text{Base Address} + (\text{Index} \times \text{SizeOf}(\text{Type}))$$

The shared memory and the banks

To allow 32 threads (a warp) to access memory simultaneously, shared memory is divided into 32 discrete memory banks. The bank indexes from 0 to 31 and each bank is 4 bytes wide.

The GPU determines which bank an address belongs to based on the 4-byte word index.

$$\text{Bank Index} = \frac{\text{Byte Address}}{4} \pmod{32}$$

(Bytes 0-3): Stored in Bank 0

(Bytes 4-7): Stored in Bank 1

(Bytes 8-11): Stored in Bank 2

(Bytes 12-15): Stored in Bank 3 and so on

Bank Conflicts

The hardware can only serve one unique address per bank per clock cycle. If 2 threads conflict, the access is serialized (takes 2x as long). If 32 threads conflict (worst case), it takes 32x as long.

If multiple threads read the exact same address, there is no conflict. The hardware performs a multicast/broadcast, serving all threads in 1 cycle. A Bank Conflict occurs when multiple threads in the same warp try to access different addresses that map to the same bank.

The H100 memory controller processes requests in 128-byte transactions

If your warp requests more than 128 bytes total (e.g., every thread loads a 16-byte float4), the request is split into multiple transactions (waves).

Origin of swizzling

Because the H100 relies on Tensor Cores (which read 64x64 tiles of data), standard linear addressing often creates massive bank conflicts (strided access).

If you have a 64×64 tile of bf16 numbers.

Row size: 64 elements×2 bytes=128 bytes.

Bank capacity per row: 32 banks × 4 bytes = 128 bytes.

Every column in your matrix lines up perfectly in the same bank((128/4) mod(32))

We want to make sure that `Matrix[0][0]` and `Matrix[1][0]` land in different banks, even though their stride is a perfect multiple of the bank width

Pointers and state spaces

The Unified Virtual Addressing (UVA) system maps each state space to a non-overlapping region:

0x0000000000000000 -> 0x0000FFFFFFFFFFFF (Reserved)

0x0001000000000000 -> 0x0001FFFFFFFFFFFF (Global memory)

0x0002000000000000 -> 0x0002FFFFFFFFFFFF (Local memory per context)

0x0003000000000000 -> 0x0003FFFFFFFFFFFF (Shared memory per block)

0x0004000000000000 -> 0x0004FFFFFFFFFFFF (Constant memory)

Generic pointers

In modern CUDA (Compute Capability ≥ 3.5), all pointers are generic by default - they're 64-bit addresses in a single unified virtual address space spanning all state spaces

There is no runtime checking to determine what state space we’re in

Can do all the normal arithmetic like the pointers in CPU

When a generic pointer is dereferenced, the hardware checks the high bits to route the request to the correct memory subsystem, this results in a few lost cycles in runtime space detection

State Space pointers

When you write inline PTX or compiler generates it, pointers can be qualified with their state space

NOT a pointer just raw bits meaningful to specific hardware instructions

Cannot be dereferenced in C++ (*(float*)shared_bits would crash or give garbage)

No runtime space detection hardware knows it's shared memory immediately

Enables specialized instructions that require explicit space qualification

cvta convert to address

cvta (Convert To Address) is a PTX (Parallel Thread Execution) instruction that converts pointers between generic addresses and specific memory space addresses.

```
// convert const, global, local, or shared address to generic address
cvta.space.size  p, a;        // source address in register a
cvta.space.size  p, var;      // get generic address of var
cvta.space.size  p, var+imm;  // generic address of var+offset

// convert generic address to const, global, local, or shared address
cvta.to.space.size  p, a;

.space = { .const, .global, .local, .shared{::cta, ::cluster}, .param{::entry} };
.size  = { .u32, .u64 };
```

The cuda c++ API for this

__cvta_generic_to_global

__cvta_generic_to_shared

__cvta_generic_to_local

__cvta_generic_to_constant

Where do we need these

Tensor Cores require explicit shared memory addresses for ldmatrix

TMA needs explicit space qualifications

Thread Block Clusters need precise space control for cross-SM data sharing, we use cluster specific addresses to work with them

In Multi-Instance GPU (MIG) configurations , explicit address space conversions help ensure proper memory isolation between GPU instances.

mapa and thread block clusters

Used for memory address conversion from shared memory in cta to distributed shared memory

```
mapa.shared::cluster.u64  %dst, %src_generic, %rank;
mapa.shared::cluster.u32  %dst, %src_generic, %rank;
```

mapa converts a shared memory address from the current CTA's shared memory space into the corresponding address in another CTA's shared memory within the same cluster

Crucial point: It takes a rank, not a block ID!

Rank vs Block Id

Block ID (%ctaid): Global position in the grid (0 to N-1 across all clusters)

Rank (%cluster_ctarank): Position within the cluster (0 to cluster_size-1)