GPU Architecture + CUDA Notes

Ananya Pagadala

1 TFLOPS, Cores, and Performance (car count + speed analogy)

Think of car count + speed ⇒ $\boxed{\textbf{TFLOPS}}$
⇒ cores execute instructions in one cycle (think parallel processing)

Watch out / Important

But number of cores doesn't mean better performance.
You need more context because cycle speed can differ in the GPU w/ more cores.

Two knobs you were thinking about:

1. **The car count ⇒ core count**
2. **Core speed** ⇒ clock / frequency

TFLOPS is derived from these (plus how many FLOPs per instruction).

Key idea

Memory recall shortcut:
TFLOPS ≈ *(cores)* × *(clock)* × *(FLOPs per cycle)*.

So higher speed + core count means more energy consumption.
So even if a GPU has ↑ energy consumption, even if it has higher perf, some company might opt for one with ↓ **perf/watt**.

Table: Why "more cores" isn't always faster

Factor	What it means	Why it can bottleneck
Core count	How many lanes of compute	More lanes don't help if traffic (memory) is slow
Clock speed	How fast lanes run	Higher clock can raise power/heat and throttle
Memory bandwidth	Bytes/sec from HBM/DRAM	If data can't feed cores, cores sit idle
Latency hiding	Extra warps ready to run	If not enough warps, stalls show up
Tensor cores	Special matrix units	Huge speedups only when workload matches (GEMM/conv)

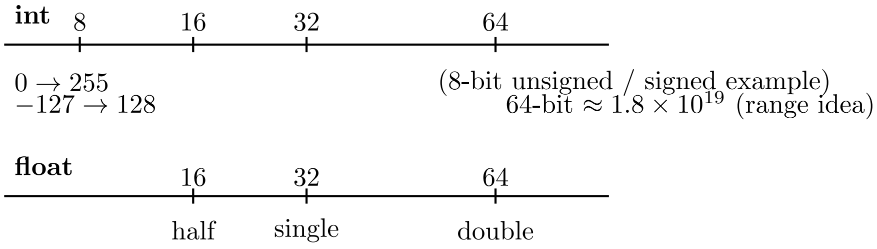
You wrote: New features (supporting new data types) → tensor cores

So perf impacted by:

1. memory bandwidth
2. TFLOPS
3. tensor cores

2 Data Types + Sizes (different ways to store numeric values)

Your sketch: ints + floats by bit-width


int
8
16
32
64
0 → 255
−127 → 128
(8-bit unsigned / signed example)
64-bit ≈ 1.8 × 10^19 (range idea)
float
16
32
64
half
single
double

CUDA data types you noted

cuda:

- **intx** → 32 bits
- log y → 64 bits

Floating point types you wrote (and why)

Type	Bits	You wrote	Clarification (why it matters)
FP16	16	half precision → lower numerical acc → less computational resources	Uses fewer bytes and can use tensor cores heavily; best when model tolerates it
FP32	32	single precision → good for general computing	Default for many CUDA kernels; balance of speed/accuracy
FP64	64	double precision → good for scientific computing	More accurate but typically slower / fewer FP64 units (depends on GPU)

Typically double precision needs more cycles.

Watch out / Important

Memory recall: Lower precision ⇒ **less memory traffic** + often **more throughput**, but can hurt stability/accuracy.

3 Compute Capability (CC)

Compute capability (CC):

- measurement of how powerful of GPU (NVIDIA based)
- CC enables different features
- if CC of GPU is not enough they don't have access to different features
- different GPUs need different version of code

Table: What CC practically changes (intuition)

CC affects...	Example consequences
Instruction support	Newer CC can run newer instructions (or faster variants)
Tensor core modes	Which data types / matrix shapes are accelerated
Memory features	Unified memory improvements, async copies, etc. (arch-dependent)
Compilation target	You compile for a specific `sm_XY` / `compute_XY`

4 GPU white papers / architecture notes

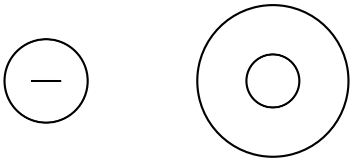
GPU white papers:
⇒ just search: "chip name" white paper and search

Volta vs Ampere (SIMT, SMs, perf knobs)

Volta ampere simd

- Transistors ⇒ indicate avail computation resource (hardware units)
- Streaming Multiprocessors (SMs) ⇒ we look at energy efficiency, FP32/FP64 perf ⇔ power, tensor cores (FLOPS)
- Latency ⇒ look at latency (how many cycles) an instruction takes

Your drawing: circles (recreated)



5 GPU → SM → Partitions → Units (your hierarchy sketch)

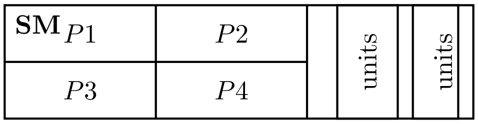
Your labeled chain

GPU ↓ (1)
SM ↓ (2)

main units (3)

Your SM partition sketch (recreated)

“4 partitions in SM”



You listed these parts

1. L0 cache → 4
2. 32 thread in a warp ⇒ **SIMT**
3. warp scheduler
4. dispatcher unit
5. register files
6. computational units

Key idea

Memory recall:
Scheduler picks *which warp* runs; **dispatcher** issues the instruction to the pipelines/cores.

6 Tensor core perf / architecture + CUDA toolchain

You wrote:

- Tensor core perf can vary based on arch (ex. Volta vs Ampere)
- It is possible that FP16 product can be FP32 (depending on the arch)
- GPUs have max threads depending on the arch
- NVLink vs PCIe

CUDA toolchain (your notes)

CUDA ⇒ compute unified device arch
CUDA compiler = NVCC

- transforms CUDA to HW PTX code
- then obtains an executable file

CUDA libraries (your notes)

CUDA libraries:

- cuBLAS ⇒ GPU accelerated linear alg
- cuFFT ⇒ fast fourier transform library
- cuDNN ⇒ DNN used for deep learning acceleration

CUDA runtime + driver APIs (your notes)

CUDA runtime + driver APIs:

- managing devices, mem, program exec on GPU
- `cudaMalloc()` allocate mem (→ plain malloc() is just on CPU)
- `cudaMemcpy()` (→ copying data GPU ↔ CPU)

Tools for debugging + perf analysis (your notes)

- Nsight systems ⇒ system wide perf (→ compute: kernel profiling)
- CUDA GDB ⇒ debugging CUDA
- cuda memcheck ⇒ memory errors

Table: Quick "which tool for what?"

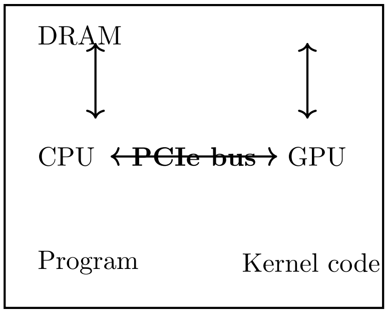
Tool	Best for
Nsight Systems	End-to-end timeline: CPU+GPU overlap, launches, syncs, memcpy
Nsight Compute	Deep kernel analysis: occupancy, memory throughput, instruction mix
cuda-gdb	Debugging kernels (breakpoints, stepping)
cuda-memcheck	Memory correctness (OOB, race-ish issues, leaks)

7 Mapping SW from CPU to HW (your diagrams)

Mapping SW from CPU to HW

- GPGPU ⇒ general purpose on GPU
- Based on C, compiler NVCC (but is not just a compiler help use GPU resources)

Your CPU/GPU + DRAM + PCIe sketch (recreated)

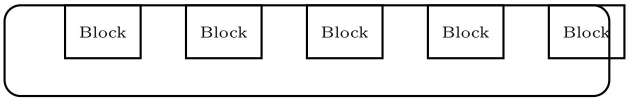


Your “GPU has 4 levels of hierarchy / SM 2 level / partitions / core”

1. GPU → 4 levels of hierarchy
2. SM in 2 level
   each SM is divided in partitions (level 3)
   SM has 4 partitions
3. FP32 core

So we need to understand how each part program maps to the different levels of GPU using threads or blocks.

Visualization: Grid → Blocks → SMs



Each Block maps to an SM (worker)

8 Blocks, warps, scheduling, dispatch (your “level 2/3” notes)

Block → think of each block of tasks (job)
which is sent to a SM (worker)

So who is the manager coordinating jobs ⇔ workers?
the GIGATHREAD (level 2!!)
⇒ thread is connected to all SMs!

Watch out / Important

But... how do we deal with the SM partitions?
can’t just assign to one partition and leave other 3 partitions

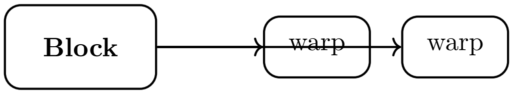
So... blocks further division into warps will allow us to divide into the four partitions.

Your “what happens when we ↑ # of warps”

What happens when we ↑ # of warps (level 3)
we have a **warp scheduler** for scheduling each warp for each cycle

So what happens in each core?
warp → 32 threads
A **DISPATCHER** makes sure that every core in the partition is actively engaged in the computational process

Small diagram: Block split into Warps



warps enable partitioning + scheduling

Your summary (preserved)

SUMMARY

1. GPU
2. SM (block / giga threads)
3. partition (warp / warp scheduler)
4. core (thread / dispatcher)

9 Need to specify # of threads and # of blocks (your occupancy notes)

Need to specify # of threads and # of blocks
For warps? Divide block size #threads/blocks by 32
Warps are fixed and are 32 threads

Volta arch (your limits)

Volta arch:

- max threads per block = 1024
- max threads per SM = 2048

So... think like this:

- 1024 threads/block = max 2 blocks per SM
- 512 → 4 blocks
- 256 → 8 blocks

SM must be busy every cycle!
⇒ if 1 warp stalls, SM needs to make another warp ready = Latency Hiding
but... how many regs + shared mem a block uses can effect

Large Blocks

Large Blocks

1. more thread can share shared mem ✓
2. fewer redundant global mem loads ✓
3. easier reductions, tiling, reuse ✓

Better data sharing + coordination

↓ If 2 blocks per SM then if 1 block stalls?
⇒ fewer active warps ⇒ less latency hiding
↓ occupancy drops ⇒ not bad but... ↓ flexibility

Occupancy = active warps / max possible warps
bc of more reg/shared mem per thread ⇒ ↓ # of blocks per SM ⇒ ↓ active warps

Small Blocks

Small Blocks

1. more blocks → more warps → better latency hiding ✓

fit more easily on SM ↑ pool of runnable warps
↓ less coordination + reuse

1. fewer threads cooperating
2. smaller shared mem tiles
3. more global mem traffic

What does this mean?

1. ↑ mem bandwidth usage
2. ↓ reduce arithmetic intensity
3. hurt cache locality

Basically... may be loading data mult times

Table: Large vs Small blocks (memory recall)

	Large blocks	Small blocks
Reuse / shared memory	Higher (good tiling)	Lower
Latency hiding	Can be worse if fewer warps	Often better (more runnable warps)
Global memory traffic	Often lower	Often higher
Occupancy	May drop (regs/shmem limit)	May rise (more blocks fit)

Watch out / Important

Important: High occupancy is a tool for *latency hiding*, not a guarantee of max performance.

10 Vector size / chunking notes (your “8x smaller chunk” idea)

Large $\#'s$ continued: remember when you change vector size change config of application
remember mult by 4 to get total of # of bytes

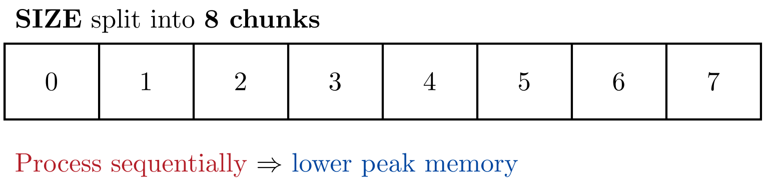
$$\texttt{\#define SIZE } 1024 \cdot 1024 \cdot 128$$

need a

$$\texttt{\#define CHUNK : smaller vector chunks : } \ 1024 \cdot 1024 \cdot 1024$$

8x smaller chunk, doing each piece sequentially
use a for loop to iterate over the 8 chunks
↑ this process of using chunks reduces the mem being used at a certain time

Visualization: Chunking timeline


SIZE split into **8 chunks**
0
1
2
3
4
5
6
7
Process sequentially ⇒ lower peak memory

11 Extra quick "memory recall" tables

Table: Vocabulary mapping (CPU analogy)

Term	Meaning (in your words)
Block	block of tasks (job) sent to an SM (worker)
Warp	fixed group of 32 threads (SIMT unit)
Warp scheduler	schedules each warp for each cycle
Dispatcher	keeps cores engaged / issues work into partition pipelines
GigaThread engine	manager coordinating blocks across SMs (level 2!!)

Table: Perf drivers you listed (fast checklist)

Driver	What to ask yourself
Memory bandwidth	Is this kernel waiting on memory? Are loads coalesced?
TFLOPS	Is the kernel compute-heavy and using the right math units?
Tensor cores	Can I use FP16/BF16/TF32 + matrix ops to hit tensor cores?
Occupancy	Do regs/shmem limit active warps and reduce latency hiding?

12 Runtime API Functions

Key idea

Goal of this section: use CUDA **runtime API** calls to **query the GPU(s)** on your system and understand key hardware limits (threads, grid dims, bandwidth, SM count, CC).

```
cudaGetDeviceCount(&nDevices);
```

Runtime API function that gets the number of CUDA-capable devices on the system. It stores the total number of GPUs in `nDevices`.

```
cudaGetDeviceProperties(&prop, i);
```

Use this in a loop for all devices listed (i = 0 to nDevices-1) to query each GPU's properties.

Example output (device query)

You get something like:

```
Device Number: 0
Device name: Tesla V100-PCIE-32GB
Memory Clock Rate (KHz): 877000
Memory Bus Width (bits): 4096
Peak Memory Bandwidth (GB/s): 898.048000

Total global memory: 34072559616 : 34 gb
Compute capability: 7.0
Number of SMs: 80
Max threads per block: 1024
```

Key idea

Memory recall: Peak memory bandwidth is **roughly** derived from memory clock + bus width.
Idea: wider bus and faster memory clock ⇒ more bytes/sec ⇒ potentially better throughput for memory-bound kernels.

Warp note (preserved)

```
// NOTES each WARP is 32 so WARP = 32
```

Watch out / Important

Warp = 32 threads is a **hardware scheduling unit** in NVIDIA GPUs (SIMT). Many performance concepts (like occupancy + latency hiding) depend on how many **warps** are active.

Occupancy (INSERT filled): what is it and why important?

Occupancy is the fraction of the maximum possible number of **active warps** on an SM that your kernel actually has active:

$$\textbf{Occupancy} = \frac{\text{active warps per SM}}{\text{max warps per SM}}$$

- **Why it matters:** Occupancy helps **hide latency**. When one warp stalls (memory, dependency), the SM can switch to another ready warp.
- **What it does NOT guarantee:** High occupancy does **not automatically mean fast**. If you're memory-bandwidth limited, more warps may not help.

Max thread dimensions (INSERT filled): what does this mean simply?

```
Max threads dimensions:  x = 1024, y = 1024, z = 64
```

Key idea

Simple meaning: this is the **maximum size of a CUDA block** in each dimension. You can launch a block as `dim3(blockX, blockY, blockZ)` but each axis has a limit. Also, **total threads per block** is still limited (e.g., **1024** threads total on V100).

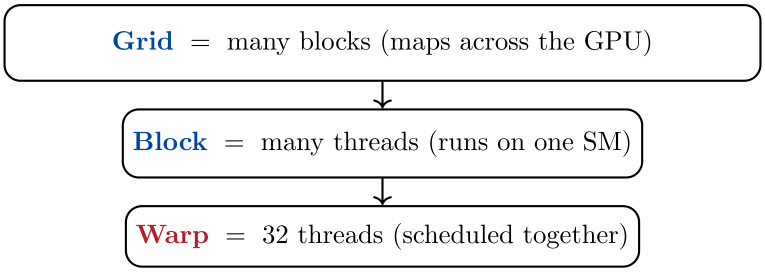
Max grid dimensions (INSERT filled): what does this mean simply?

```
Max grid dimensions:  x = 2147483647, y = 65535, z = 65535
```

Key idea

Simple meaning: this is the maximum number of **blocks in the grid** you can launch along each axis.
Grid size = `dim3(gridX, gridY, gridZ)`. This controls how many blocks exist overall.

Visualization: Thread hierarchy (grid → block → warp)



Grid = many blocks (maps across the GPU)

Block = many threads (runs on one SM)

Warp = 32 threads (scheduled together)

13 NVIDIA-SMI → System Management Interface

Key idea

What is NVIDIA-SMI? A command-line tool to **monitor**, **control**, and **query** NVIDIA GPU state.

Three key features (cleaned + preserved)

- **Performance monitoring:** utilization, memory usage, temperature, and power
- **Settings management:** controlling clock speeds and power limits
- **Device info + query:** GPU name, driver version, CUDA version, MIG status, ECC, etc.

Example output (INSERT filled: color coding by feature)

Watch out / Important

Color guide for reading the output:
Blue = monitoring metrics (util/temp/mem/power)   Red = settings/limits (P-states, power caps)   **Black** = device identity + versions

```
nvidia-smi
Sat Feb  7 23:27:50 2026
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 580.105.08             Driver Version: 580.105.08     CUDA Version: 13.0     |
+-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  Tesla V100-PCIE-32GB           On  |   00000000:D8:00.0 Off |                    0 |
| N/A   28C    P0             26W /  250W |       0MiB /  32768MiB |      0%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+

+-----------------------------------------------------------------------------------------+
| Processes:                                                                              |
|  GPU   GI   CI              PID   Type   Process name                        GPU Memory |
|        ID   ID                                                               Usage      |
|=========================================================================================|
|  No running processes found                                                             |
+-----------------------------------------------------------------------------------------+
```

Key idea

Why show driver + CUDA versions?
Some libraries depend on a minimum (or specific) **driver version** and **CUDA version**. If something fails to load, checking these is step 1.

Monitoring GPU continuously (cleaned)

Monitoring GPU continuously (Linux command): `nvidia-smi -l`
This refreshes output every second so you can see GPU util + memory change live.

INSERT: useful nvidia-smi commands (filled)

Goal	Command
Show summary (default view)	`nvidia-smi`
Refresh every 1s	`nvidia-smi -l 1`
Show specific fields (custom query)	`nvidia-smi --query-gpu=name,uuid,utilization.gpu,utilization.m --format=csv`
Save output to a file	`nvidia-smi -q > smi_report.txt`
Check clocks in use (graphics/SM/mem)	`nvidia-smi -q -d CLOCK`
Check all supported clocks	`nvidia-smi -q -d SUPPORTED_CLOCKS`
Set power limit (requires permission)	`sudo nvidia-smi -pl 200`
Set application clocks (if supported + permitted)	`sudo nvidia-smi -ac <memClockMHz>,<smClockMHz>`
Permission note (why it might fail)	**If you get "Insufficient Permissions", you need admin or cluster policy support.**

Watch out / Important

Cluster reality: On shared HPC systems, power/clock changes are often disabled for users. You can still monitor everything.

14 GPU Occupancy and Utilization

Theoretical vs achieved occupancy (fixed spelling, preserved meaning)

There's a difference between **achieved occupancy** and **theoretical occupancy**:

- **Theoretical occupancy (ideal):** calculated as *warps used in kernel / max warps per SM*. It refers to the maximum percentage of compute resources a kernel **could** use under ideal conditions (no bottlenecks: memory, compute dependencies, synchronization).
- **Achieved occupancy (actual):** what the kernel **really gets** at runtime. Achieved occupancy is affected by scheduling overhead, imbalance across warps/blocks, stalls, and resource limits (registers/shared memory).

Your mental model (preserved + clarified)

Achieved: actual usage. Since GPUs are made of SMs, each SM has multiple partitions. A block runs on an SM, and warps execute on the partition.

Key idea

Helpful framing:
Warps = software execution groups **Partitions** = hardware execution lanes
Scheduler picks which warp runs; dispatcher issues its instructions.

Each partition has a **scheduler** that allocates a warp to execute each cycle.
A **dispatcher unit** takes ready instructions and sends them to the appropriate computation units.

Stalls + latency hiding (fixed grammar, preserved content)

There are examples where warps are not ready. A warp can stall anywhere from **30 to 300 cycles**. During this wait, another warp may run.

However, if there are a lot of memory requests or scheduling dependencies, there are situations where **no warp can be issued** — this is a **STALL cycle**.

> **Watch out / Important**
>
> **Latency hiding:** the whole point of having many warps is that when one stalls, another runs.
> If **all** warps stall, you get visible idle cycles.

Your NVBit note (preserved, clarified)

Scenario: (1) no memory dependency / dependency.
NVBit: assembly is SASS instructions.

We are able to see the software for each warp in assembly, and observe what warps are doing.

Say you have four warps on one partition:
How many warps are utilized per partition?
In this case you can see there is

warps/p = 4 warps arps/sm = 4*4=16 warps

say you ha ve the assembly code, warm schduel chooses one of the warps to excuted

ex) a flop instruction , warp sheduleder says i have warp 0 and its ready then the dispatcher with thake the first intrsuction here and send it to al the core heres that are FP32,

so ii its once instrc why is it sent to all fp32? thats because each warp is 32 threads but on diffrent data, so each assemply linea fter the first istruct it will be sent to all the core with idffrent reg,

in the next cylce, scheduler as the warps which one is ready goes into the cycle into each warp,

15 Profiling with NCU (Nsight Compute) — vecAdd example

> **Key idea**
>
> **Goal:** If you run a file, you can profile the app with **NCU** to understand whether you are compute-bound or memory-bound, and to see occupancy + launch stats.

So if you run a file, all you need to do is profile the app. You can use the command:

```
ncu ./vecAdd
```

Example output (cleaned, preserved)

```
==PROF== Profiling "vecAdd" - 0: 0%
==WARNING== Backing up device memory in system memory. Kernel replay might be slow.
```

```
           Consider using "--replay-mode application" to avoid memory save-and-restore.
....50%....100% - 19 passes
Execution time: 4944.171387 ms

Finished
==PROF== Disconnected from process 1141543
[1141543] vectorAdd@127.0.0.1
  vecAdd(int *, int *, int *, int) (442368, 1, 1)x(1024, 1, 1), Context 1, Stream 7, Device 0,
```

Watch out / Important

INSERT filled: total number of blocks

Grid size `(442368, 1, 1)` means **442,368 blocks total** are launched (in X dimension).

Visualization: Launch configuration

Launch: `(gridX,gridY,gridZ)` × `(blockX,blockY,blockZ)`
Example: `(442368, 1, 1)` × `(1024, 1, 1)`
Threads per block: 1024 **Total threads launched:** 442368 × 1024

GPU Speed Of Light Throughput (table version)

Metric Name	Unit	Value
DRAM Frequency	MHz	876.40
SM Frequency	GHz	1.24
Elapsed Cycles	cycles	8,215,186
Memory Throughput	%	91.11
DRAM Throughput	%	91.11
Duration	ms	6.65
L1/TEX Cache Throughput	%	25.88
L2 Cache Throughput	%	34.03
SM Active Cycles	cycles	8,159,381
Compute (SM) Throughput	%	10.78

Watch out / Important

Quick read: Memory throughput is **very high** (91%) while compute throughput is **low** (10.78%).
This strongly suggests the kernel is **memory-bound** (DRAM is the limiter).

Launch Statistics (table version)

Metric	Value
Block Size	1024
Grid Size	442368
Registers Per Thread	16
Static Shared Memory Per Block	0 bytes
Dynamic Shared Memory Per Block	0 bytes
# SMs	80
Threads (total)	452,984,832
Waves Per SM	2764.80

Occupancy section (preserved + clarified + extra table)

Section: Occupancy // here is what we are looking at:

Metric Name	Metric Value
Block Limit SM	32
Block Limit Registers	4
Block Limit Shared Mem	32
Block Limit Warps	2
Theoretical Active Warps per SM	64
Theoretical Occupancy	100%
Achieved Occupancy	83.02%
Achieved Active Warps Per SM	53.13

Key idea

Memory recall: why achieved ¡ theoretical?
Even if resources allow 100% occupancy, real execution can be lower due to: warp scheduling overhead, uneven work distribution, divergence, or stalls.

Your note (cleaned, preserved): We are using a good amount, but the number of threads can affect the occupancy of your application. Changing the number of threads per block can change occupancy. We are going to execute 32 blocks simultaneously per SM. Let's take this value, and calculate theoretical occupancy and get a percentage.

Watch out / Important

Important clarification:
Occupancy is **not** "blocks per SM". It is **active warps per SM** divided by max warps per SM.
Blocks per SM influences occupancy because each block contributes warps, but registers/shared memory can cap blocks and warps.

GPU + Memory Workload Distribution (table version)

Metric Name	Metric Value
Average DRAM Active Cycles	5,306,905.59
Total DRAM Elapsed Cycles	186,398,080
Average L1 Active Cycles	8,159,381
Total L1 Elapsed Cycles	656,324,412
Average L2 Active Cycles	7,782,946.30
Total L2 Elapsed Cycles	499,117,376
Average SM Active Cycles	8,159,381
Total SM Elapsed Cycles	656,324,412
Average SMSP Active Cycles	8,119,229.41
Total SMSP Elapsed Cycles	2,625,297,648

Key idea

Interpretation hint: If DRAM throughput is high and compute throughput is low, optimize memory behavior first: coalescing, fewer global loads, more reuse (tiling/shared memory), better cache locality.