GPU WORKLOAD DESIGN

AGENDA

GPU Architecture & Programming Model

Are there opportunities to improve performance?

WHY GPUS?

What is CUDA actually?

Depending on the context, "CUDA" can refer to different things

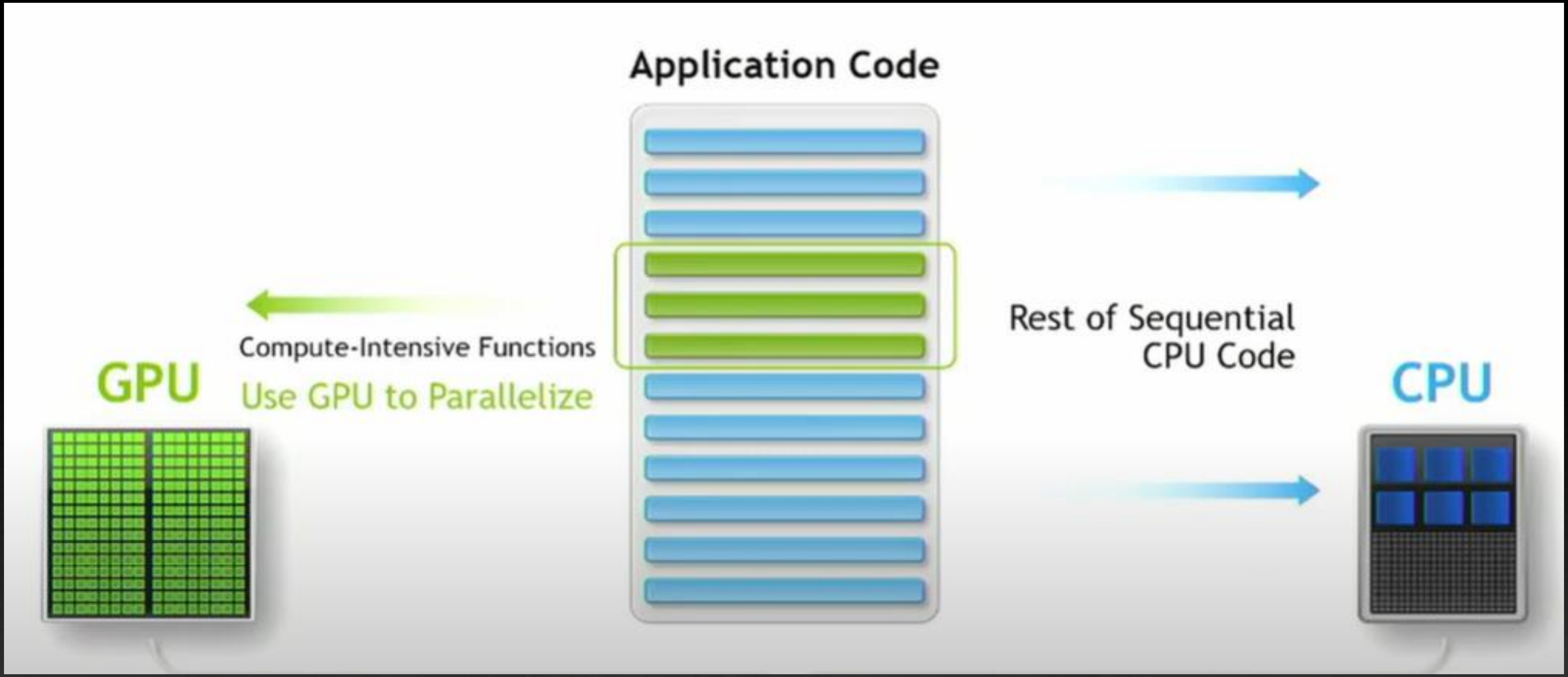
A high-level device architecture

Parallel programming model for architecture with that design

Software platform that extends high-level languages like C to add that programming model

WE NEED BOTH

Applications contain both sequential and parallel sections



GPU: throughput-Oriented design

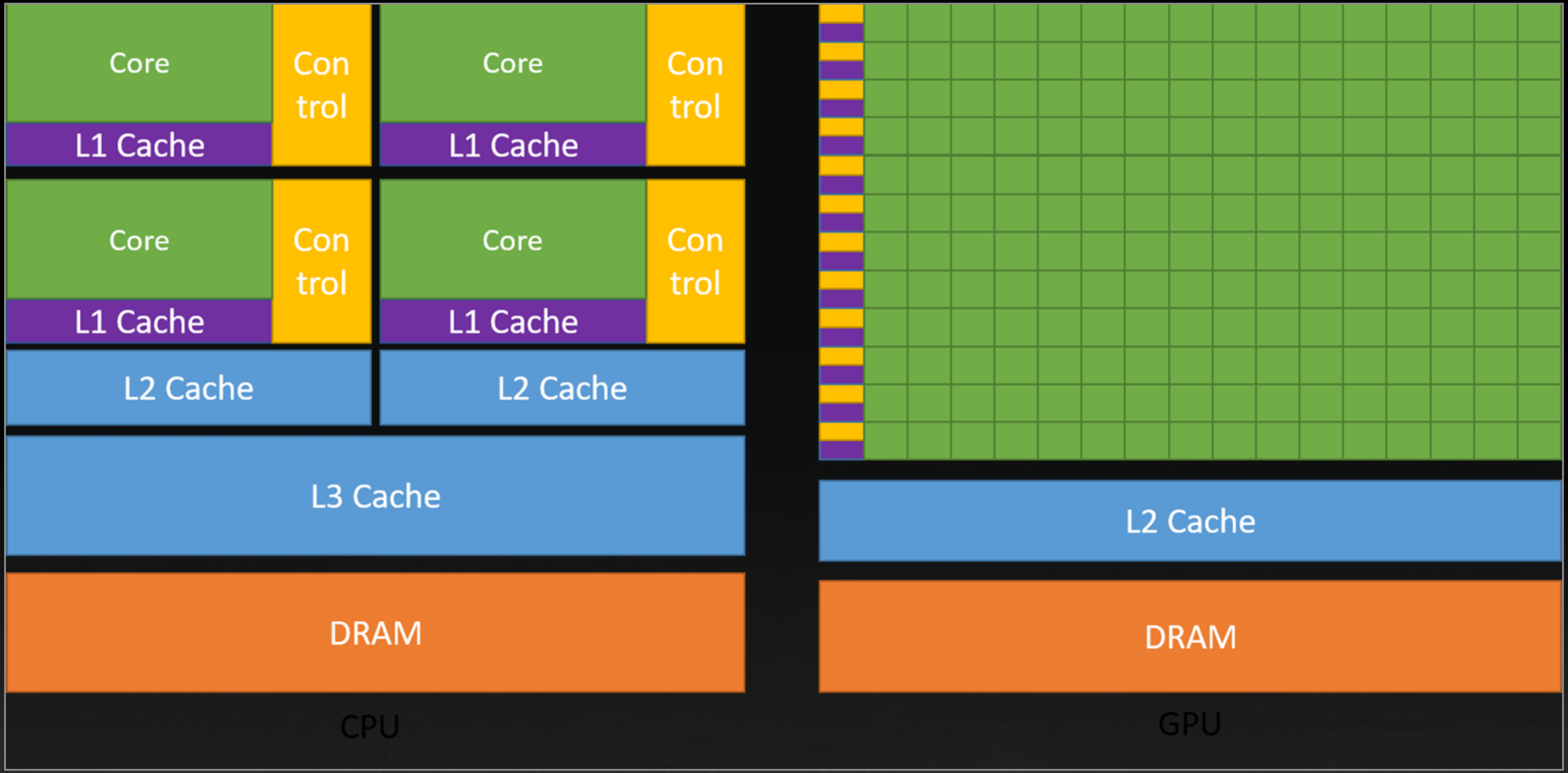
- **Focus**: Process massive numbers of parallel tasks efficiently (high throughput across thousands of threads)
- **Transistor Allocation**: majority dedicated to data processing rather than data caching and flow control
- **Execution:**
  - Many simple cores: No complex out-of-order execution or branch prediction, reducing per-core overhead
  - Optimized for parallel processing
  - Can hide memory access latencies with computation

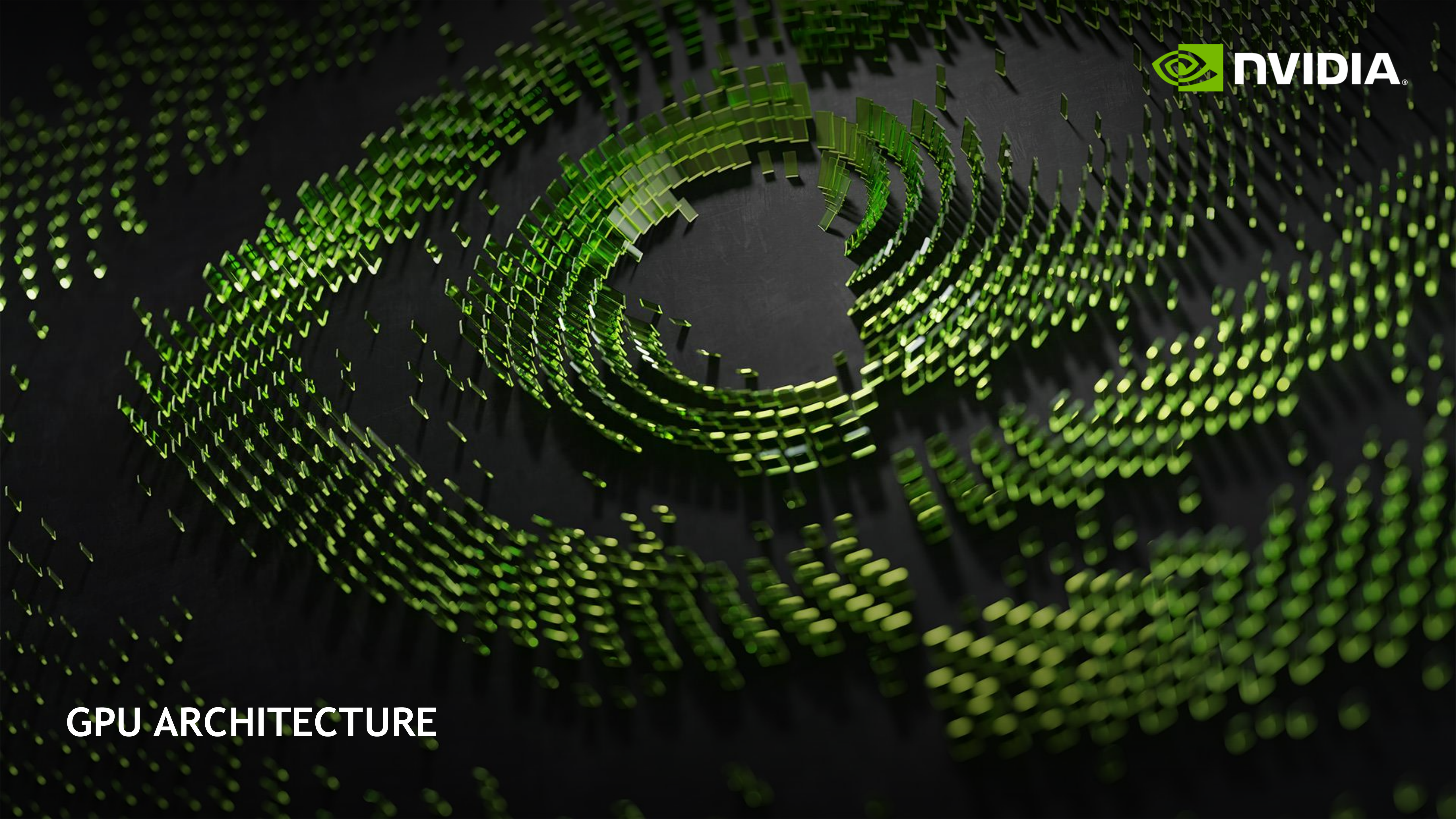
CPU: latency-oriented design

- **Focus**: Complete single tasks as quickly as possible
- **Transistor Allocation**: majority dedicated to flow control and large caches
- **Execution:**
  - Small number of high-quality, powerful, and efficient cores
  - Optimized for sequential processing (low latency per thread)
  - Relying on large data caches and complex flow control

https://en.wikipedia.org/wiki/Latency_oriented_processor_architecture
https://github.com/CisMine/Parallel-Computing-Cuda-C/blob/main/Chapter05/README.md

CPU vs GPU

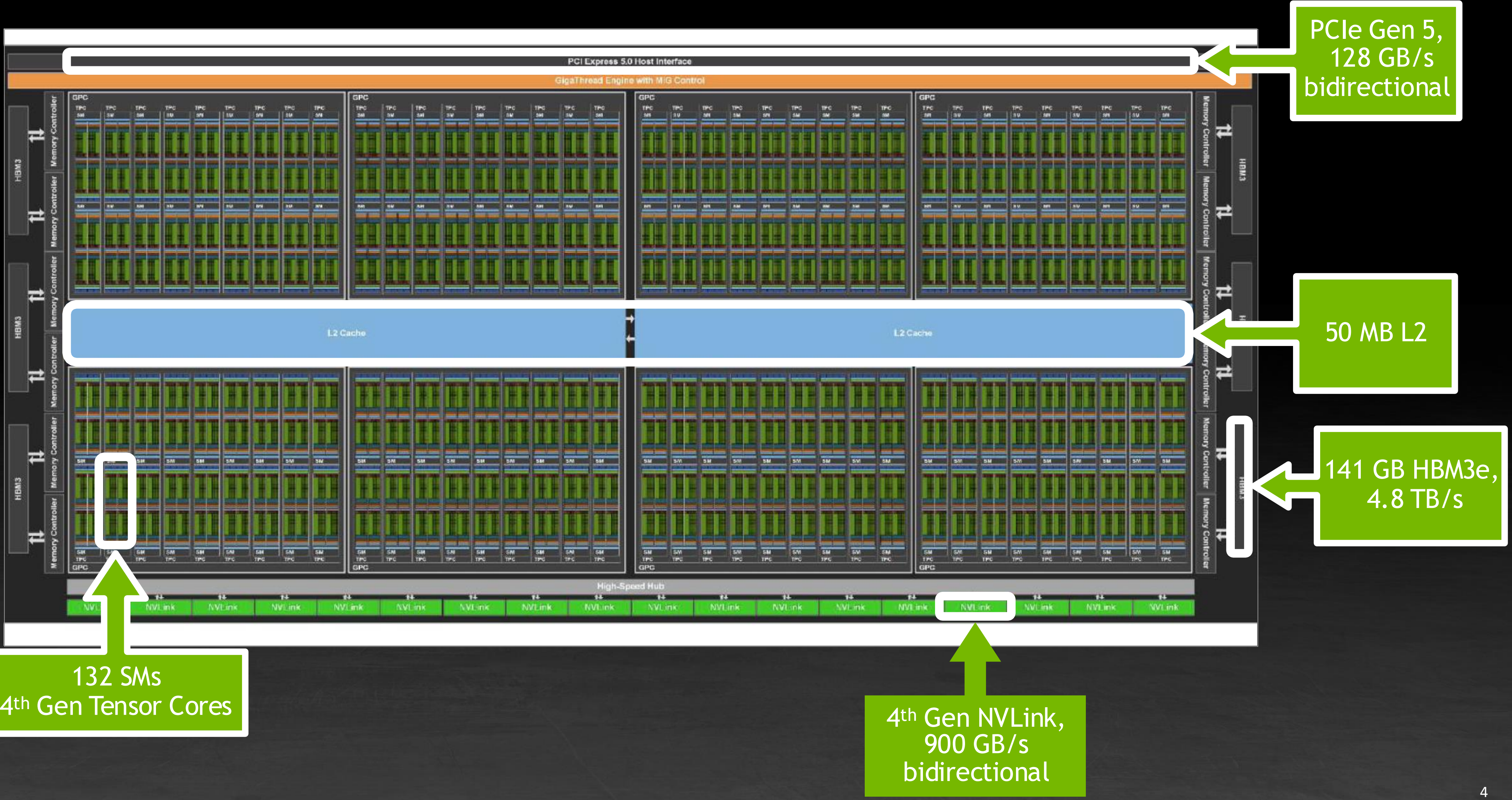

Core
Con trol
L1 Cache
Core
Con trol
L1 Cache
Core
Con trol
L1 Cache
Core
Con trol
L1 Cache
L2 Cache
L2 Cache
L3 Cache
DRAM
CPU
L2 Cache
DRAM
GPU



GPU ARCHITECTURE

GPU OVERVIEW

NVIDIA H200 SXM

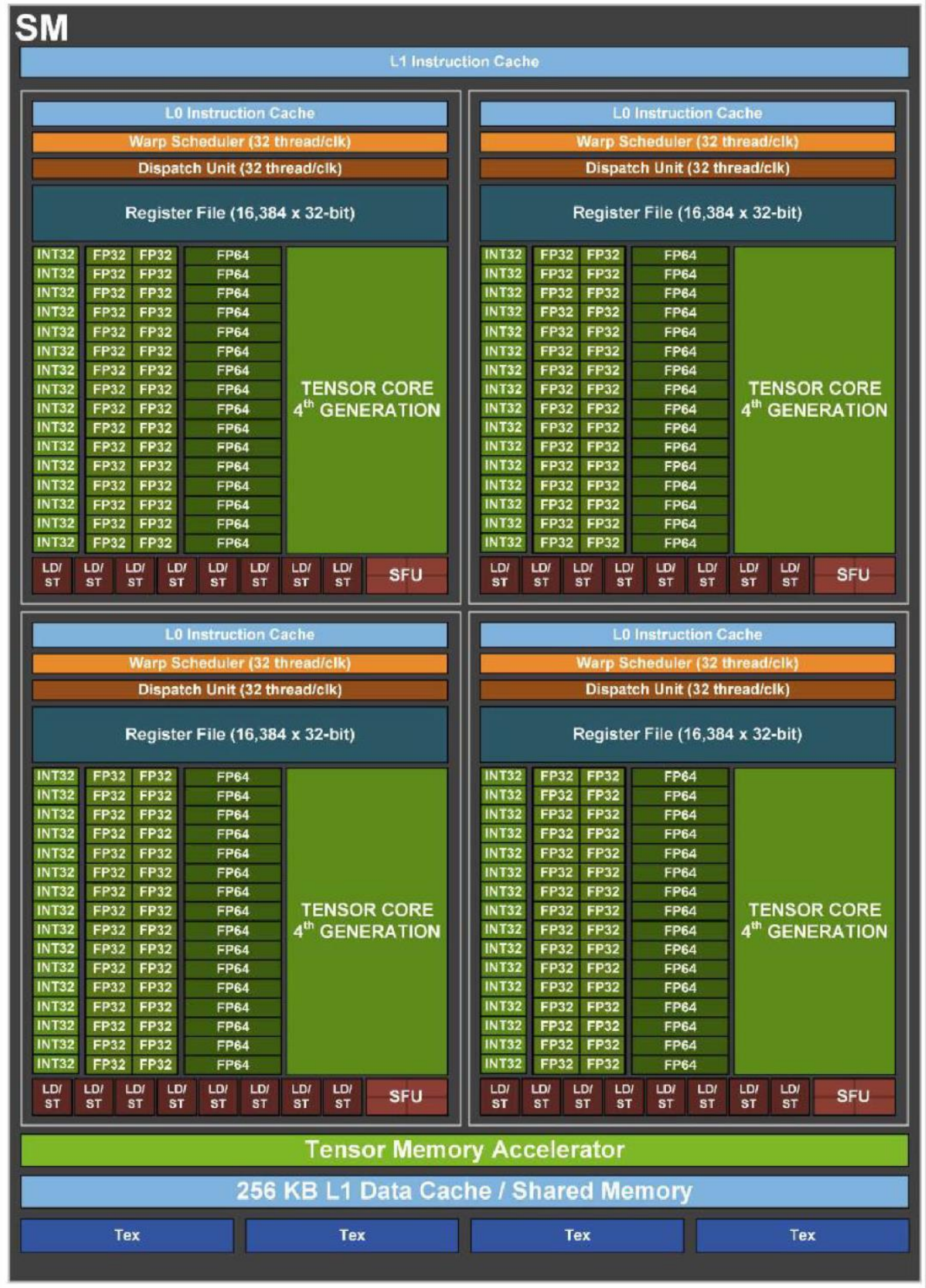

PCI Express 5.0 Host Interface
GigaThread Engine with MIG Control
GPC
TPC
SM
Memory Controller
HBM3
L2 Cache
High-Speed Hub
NVLink
PCIe Gen 5, 128 GB/s bidirectional
50 MB L2
141 GB HBM3e, 4.8 TB/s
132 SMs
4th Gen Tensor Cores
4th Gen NVLink, 900 GB/s bidirectional

STREAMING MULTIPROCESSOR (SM)

HOPPER ARCHITECTURE

- 128 FP32 cores
- 64 FP64 cores
- 64 INT32 cores
- 4 mixed-precision Tensor Cores
- 16 special function units (transcendentals)
- 4 warp schedulers

- 32 LD/ST units
- 64K 32-bit registers
- 256 KiB unified L1 data cache and shared memory
- Tensor Memory Accelerator (TMA)


SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
FP64
TENSOR CORE 4th GENERATION
LD/ST
SFU
Tensor Memory Accelerator
256 KB L1 Data Cache / Shared Memory
Tex

HARDWARE EXECUTION UNITS

- SMs are the fundamental compute units of NVIDIA GPUs, analogous to CPU cores but optimized for massive parallelism
- Key components:
  - CUDA cores: execute scalar arithmetic instructions
  - Tensor cores: operate on entire matrices. These cores are much larger and less numerous than CUDA cores
  - Warp schedulers: Manage thread execution by deciding which group of 32 threads (warps) to run
  - Context switching between warps occurs in one clock cycle

CUDA PROGRAMMING MODEL

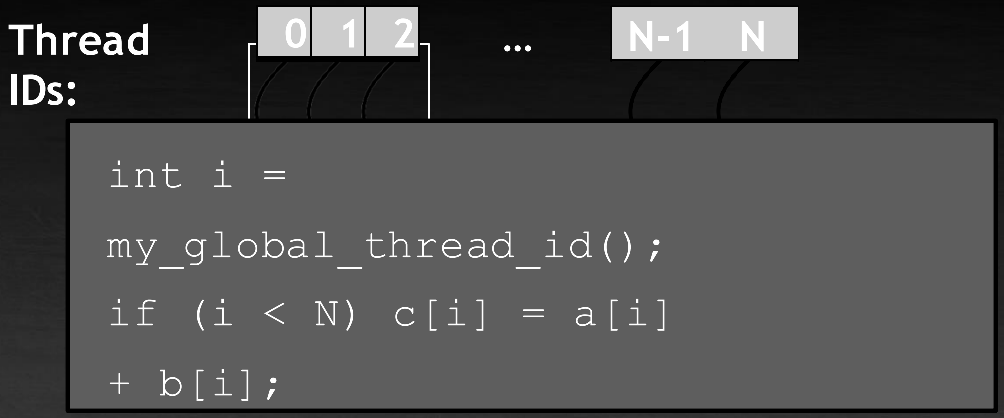
SINGLE-PROGRAM MULTIPLE-DATA

- SIMT instructions specify the execution of a **single** thread.
- A SIMT kernel is launched on many threads that execute in parallel.
- Threads use their thread index to work on disjoint data or to enable different execution paths.
- Three key software abstractions enable efficient programming through the CUDA programming model:
  - a hierarchy of **thread groups**,
  - **memory spaces**, and
  - **synchronization**.

Single-threaded CPU vector addition

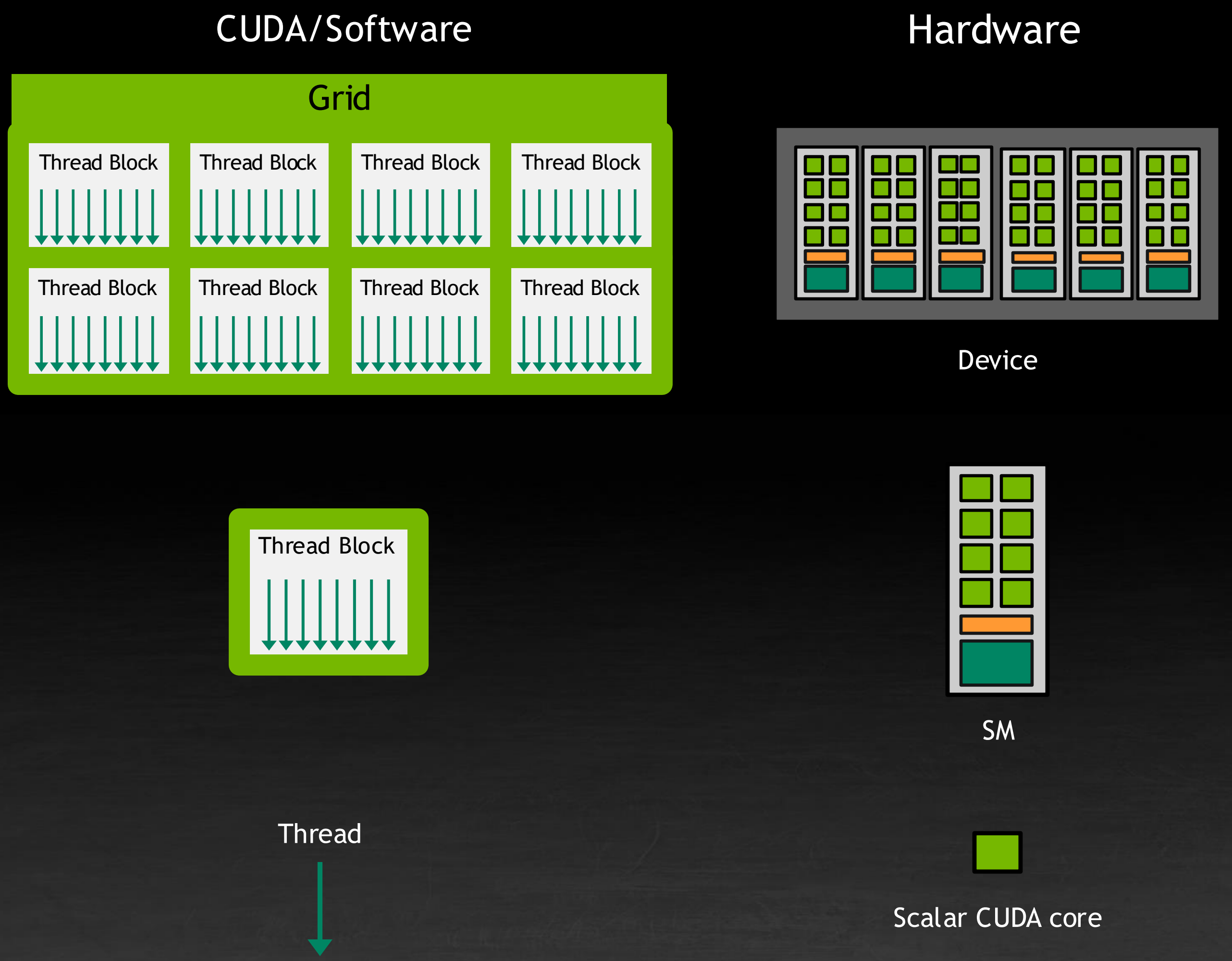
```
for (int i = 0; i <
   N; i++) { c[i] =
   a[i] + b[i];
}
```

GPU vector addition



```
int i =
my_global_thread_id();
if (i < N) c[i] = a[i]
+ b[i];
```

THREAD HIERARCHY

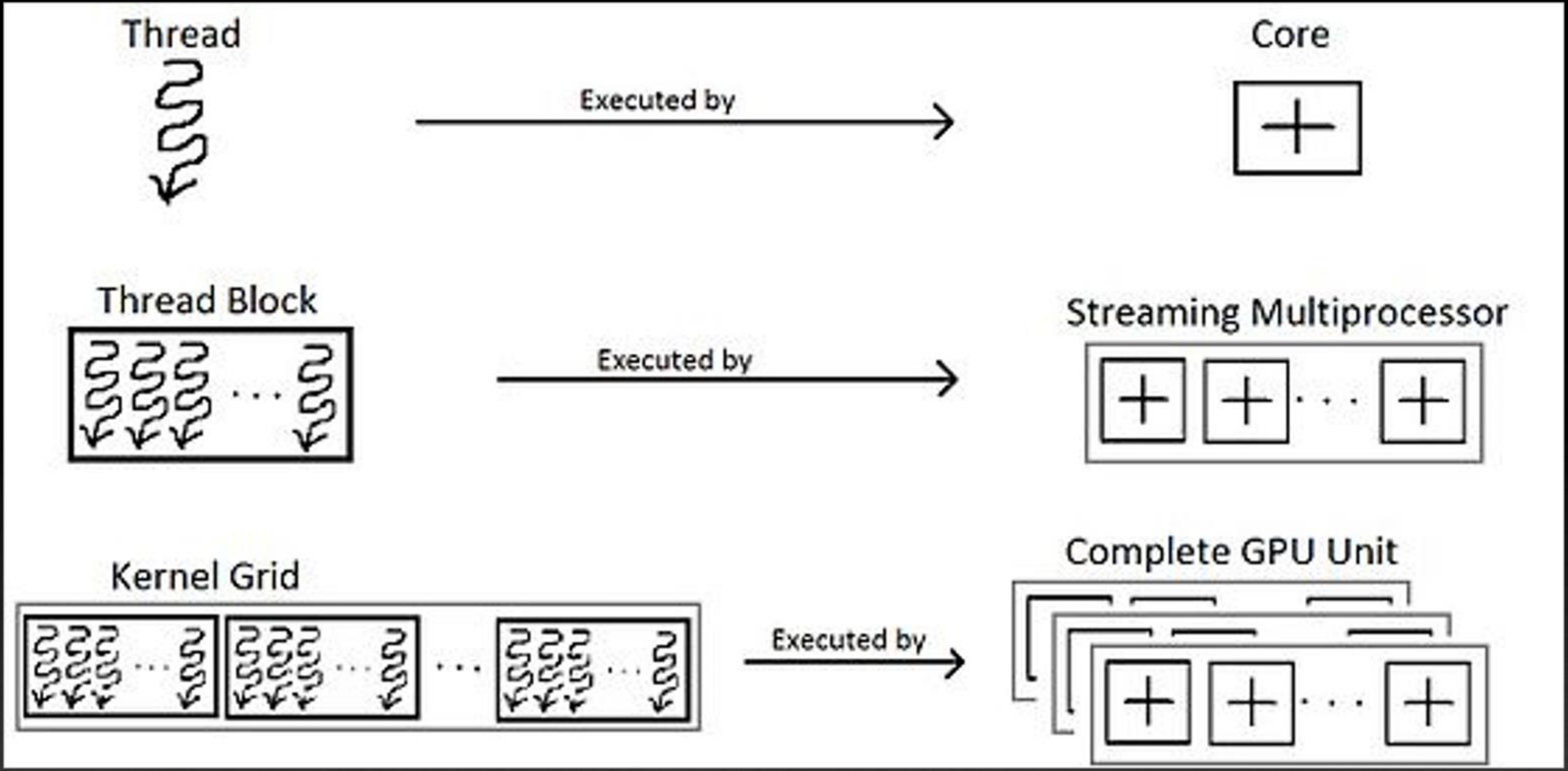


- A CUDA kernel is launched on a grid of thread blocks, which are completely independent.

- Thread blocks are executed on SMs.
  - Several concurrent thread blocks can reside on an SM.
  - Thread blocks do not migrate.
  - Each block can be scheduled on any of the available SMs, in any order, concurrently or in series.

- Individual threads execute on scalar CUDA cores.

HOW IS THIS EXECUTED?



ROOFLINE EXAMPLE STEP THROUGH

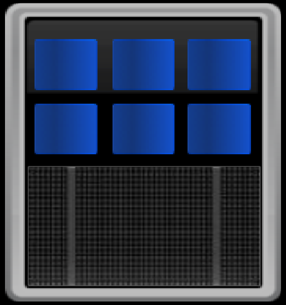
ROOFLINE EXAMPLE WALKTHROUGH

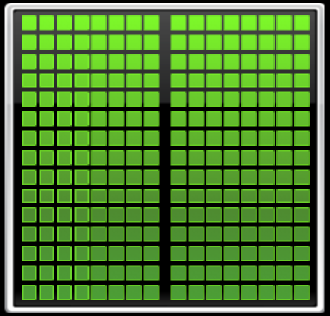
- Step through the CuPy code:

https://github.com/marcelo-alvarez/datasci211/blob/main/week-2/roofline.py

What is just-in-time compilation?

NUMERICAL COMPUTING IN PYTHON





- Mathematical focus
- Operates on arrays of data
    - *ndarray*, holds data of same type
- Many years of development
- Highly tuned for CPUs

- NumPy like interface
- Trivially port code to GPU
- Copy data to GPU
    - CuPy *ndarray*
- Data interoperability with DL frameworks, RAPIDS, and Numba
- Uses high tuned NVIDIA libraries
- Can write custom CUDA functions

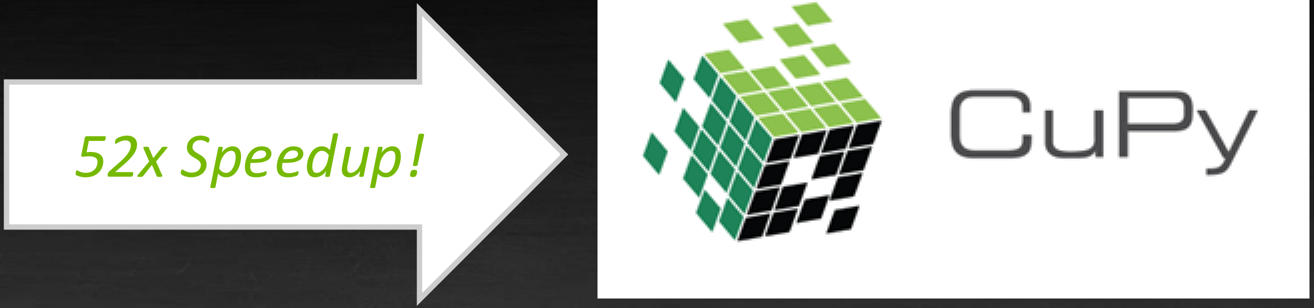
CUPY

BEFORE

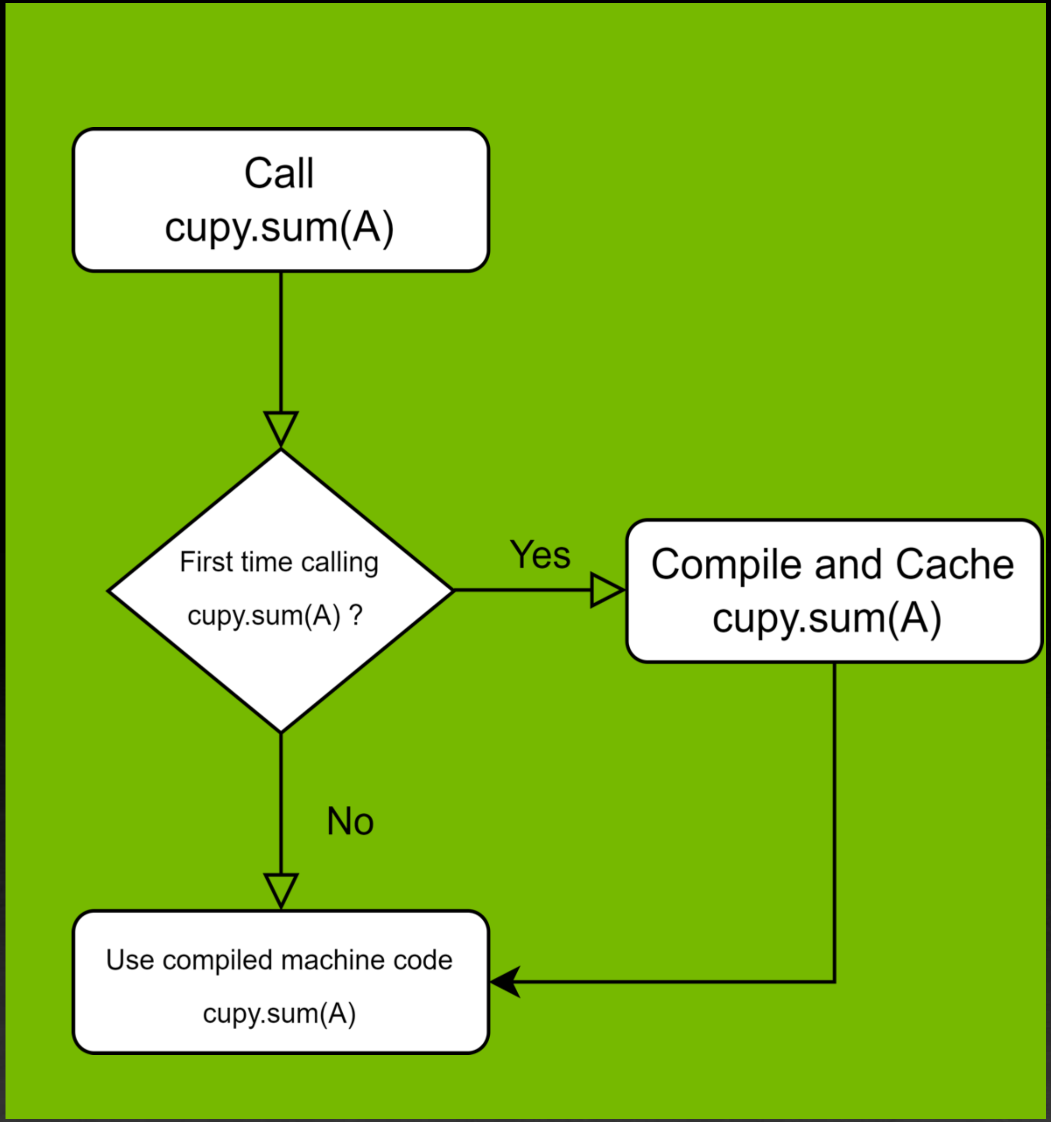
```
import numpy as np

size = 4096
A = np.random.randn(size,size)

Q, R = np.lingalg.qr(A)
```

AFTER

```
import cupy as cp

size = 4096
A = cp.random.randn(size,size)

Q, R = cp.lingalg.qr(A)
```



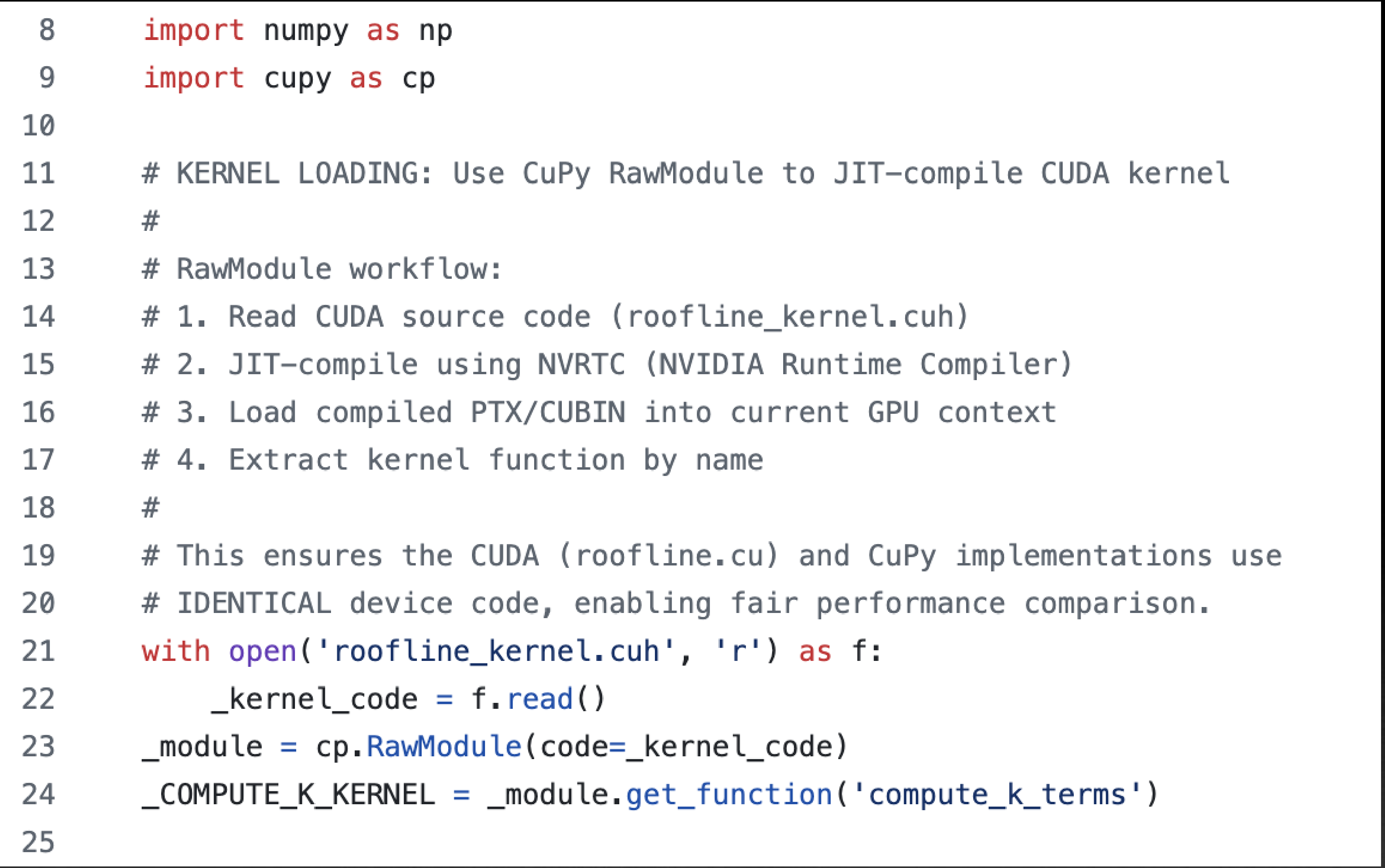
KERNEL OVERHEAD


Call cupy.sum(A)
First time calling cupy.sum(A) ?
Yes
Compile and Cache cupy.sum(A)
No
Use compiled machine code cupy.sum(A)

JIT Compilation

- What is the size of A?
- What is the datatype?
- Which GPU-accelerated libraries are available?
- Compiler optimizations for custom kernels

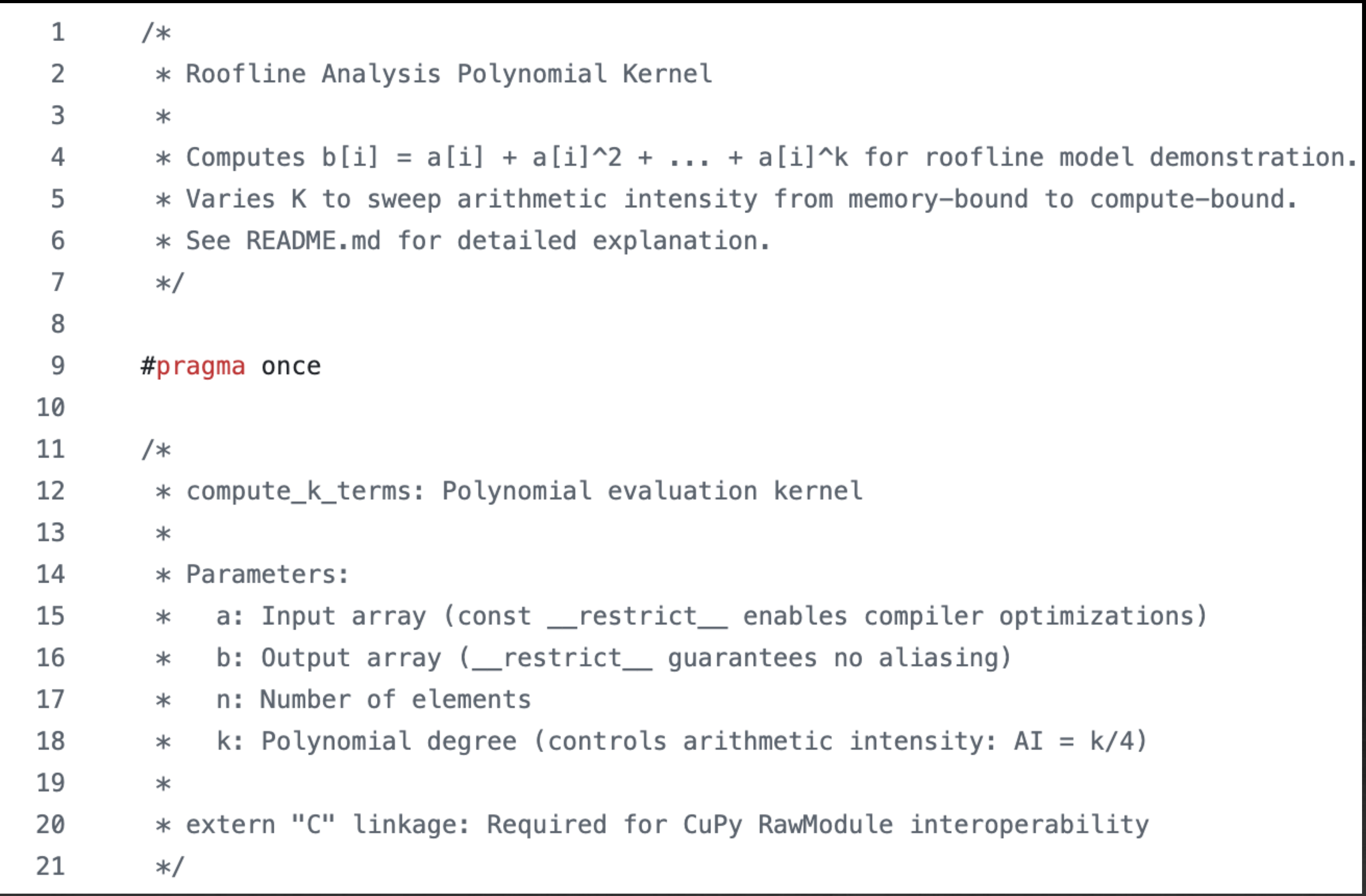
SETTING UP CUPY SCRIPT



```python
import numpy as np
import cupy as cp

# KERNEL LOADING: Use CuPy RawModule to JIT-compile CUDA kernel
#
# RawModule workflow:
# 1. Read CUDA source code (roofline_kernel.cuh)
# 2. JIT-compile using NVRTC (NVIDIA Runtime Compiler)
# 3. Load compiled PTX/CUBIN into current GPU context
# 4. Extract kernel function by name
#
# This ensures the CUDA (roofline.cu) and CuPy implementations use
# IDENTICAL device code, enabling fair performance comparison.
with open('roofline_kernel.cuh', 'r') as f:
    _kernel_code = f.read()
_module = cp.RawModule(code=_kernel_code)
_COMPUTE_K_KERNEL = _module.get_function('compute_k_terms')
```

- Loads CUDA kernel code from a file
- Just-in-time compiles it
- Picks the GPU kernel we want in our script

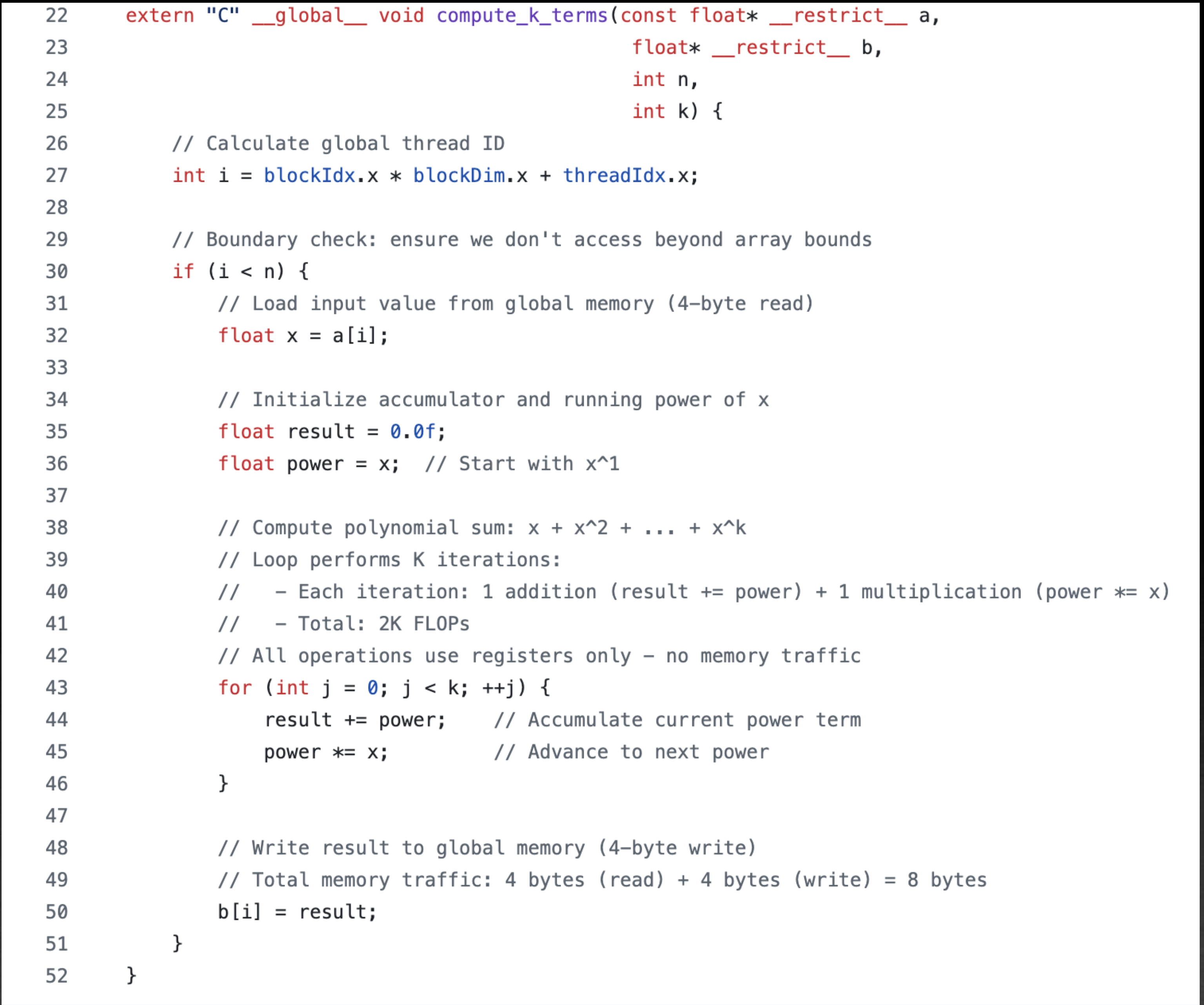
SOURCE CUDA CODE



```
/*
 * Roofline Analysis Polynomial Kernel
 *
 * Computes b[i] = a[i] + a[i]^2 + ... + a[i]^k for roofline model demonstration.
 * Varies K to sweep arithmetic intensity from memory-bound to compute-bound.
 * See README.md for detailed explanation.
 */

#pragma once

/*
 * compute_k_terms: Polynomial evaluation kernel
 *
 * Parameters:
 *   a: Input array (const __restrict__ enables compiler optimizations)
 *   b: Output array (__restrict__ guarantees no aliasing)
 *   n: Number of elements
 *   k: Polynomial degree (controls arithmetic intensity: AI = k/4)
 *
 * extern "C" linkage: Required for CuPy RawModule interoperability
 */
```

- Setup to do our roofline model computation

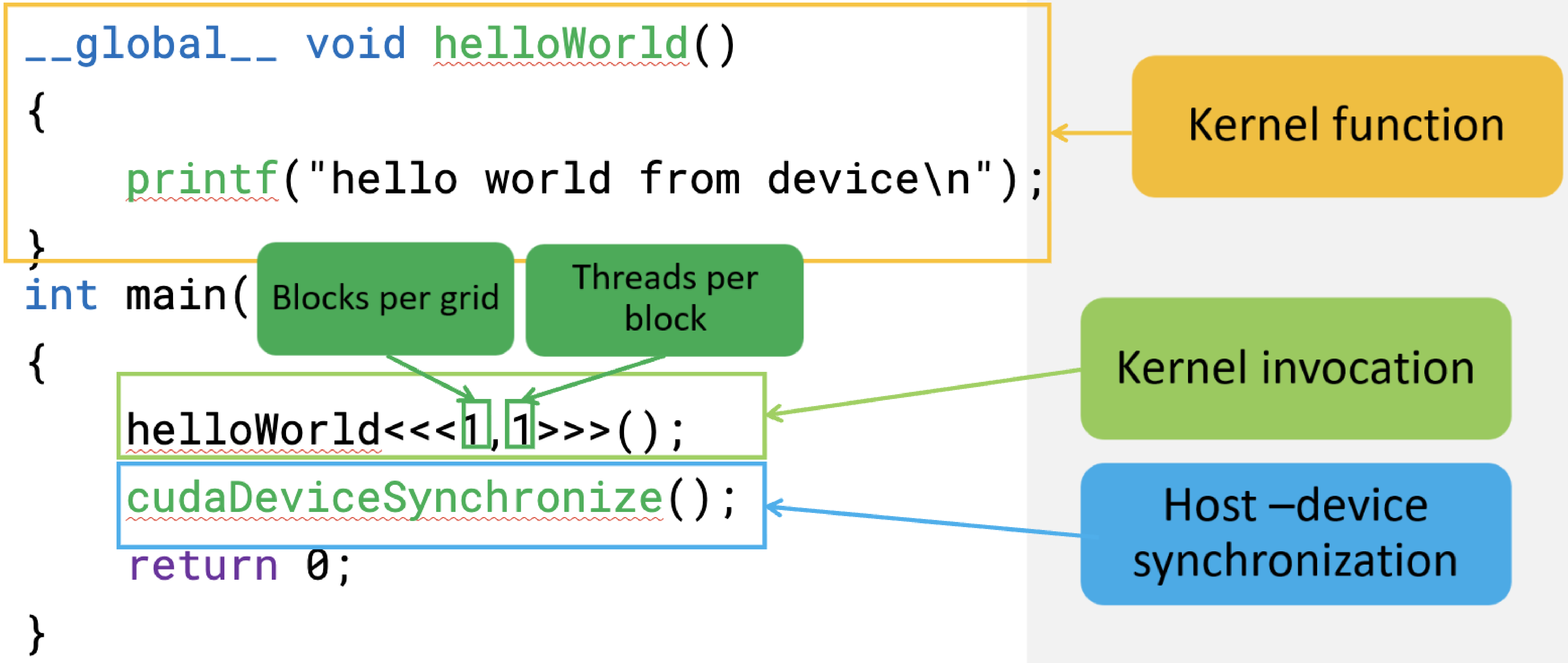
CUDA CODE IMPLEMENTATION



```
extern "C" __global__ void compute_k_terms(const float* __restrict__ a,
                                           float* __restrict__ b,
                                           int n,
                                           int k) {
    // Calculate global thread ID
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    // Boundary check: ensure we don't access beyond array bounds
    if (i < n) {
        // Load input value from global memory (4-byte read)
        float x = a[i];

        // Initialize accumulator and running power of x
        float result = 0.0f;
        float power = x;  // Start with x^1

        // Compute polynomial sum: x + x^2 + ... + x^k
        // Loop performs K iterations:
        //   - Each iteration: 1 addition (result += power) + 1 multiplication (power *= x)
        //   - Total: 2K FLOPs
        // All operations use registers only - no memory traffic
        for (int j = 0; j < k; ++j) {
            result += power;    // Accumulate current power term
            power *= x;         // Advance to next power
        }

        // Write result to global memory (4-byte write)
        // Total memory traffic: 4 bytes (read) + 4 bytes (write) = 8 bytes
        b[i] = result;
    }
}
```

- Identify unique thread running in a kernel
- Each thread computes a polynomial sum for one element
- Write results to global memory

CUDA SIDE STEP



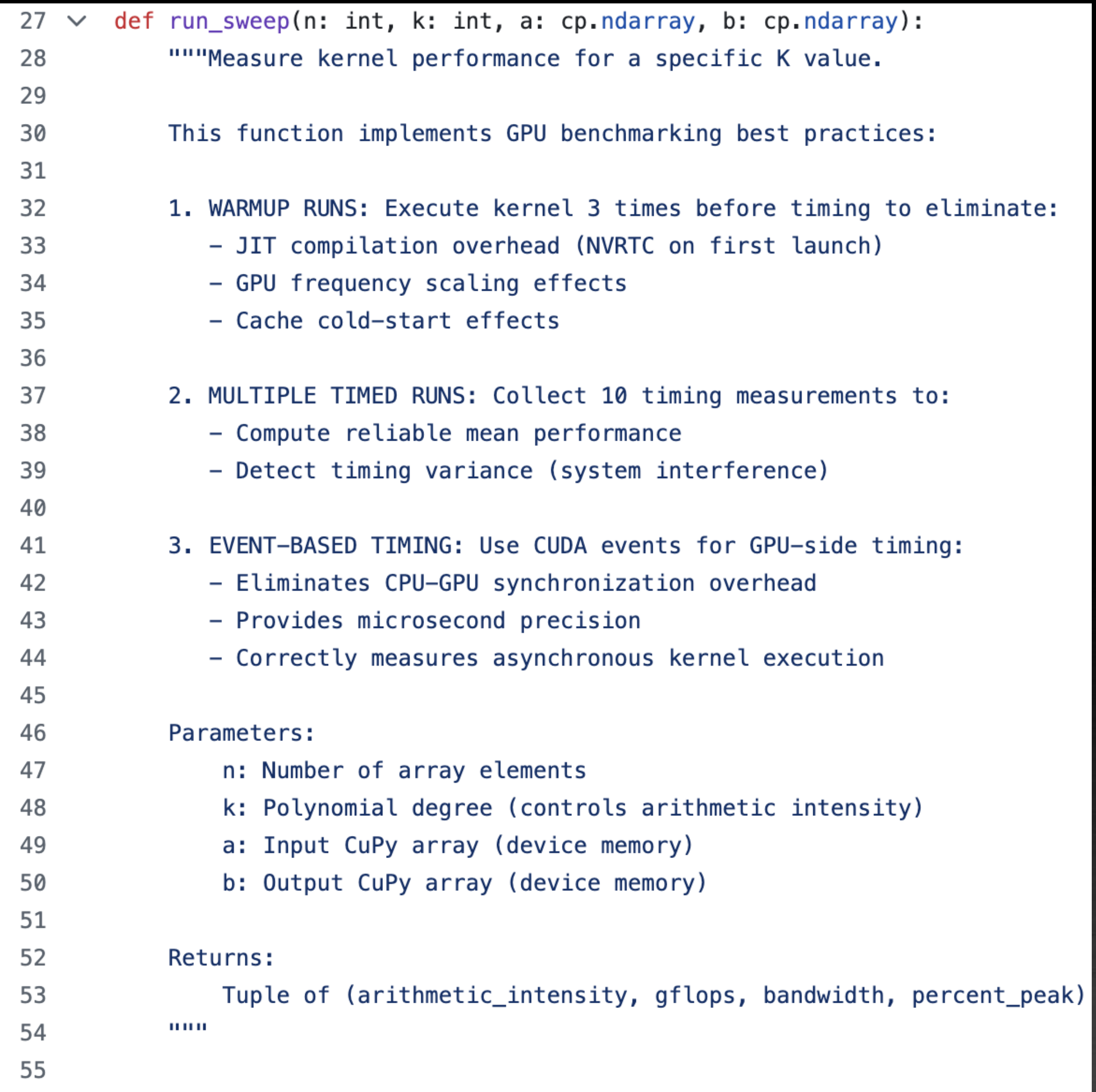
```
__global__ void helloWorld()
{
    printf("hello world from device\n");
}
int main(
{
    helloWorld<<<1,1>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

KERNEL

- Kernels are C++ functions prefixed with `__global__` declaration specifier
- `__global__` prefix defines a function that is called by the host (CPU) and executed by the device (GPU)
- Kernels are executed N times in parallel across N CUDA threads
- Kernels never return a value. Host and device cannot communicate directly. Instead, data needs to be copied back and forth between them

```
__global__ void helloWorld()
{
    printf("hello world from device \n");
}
```

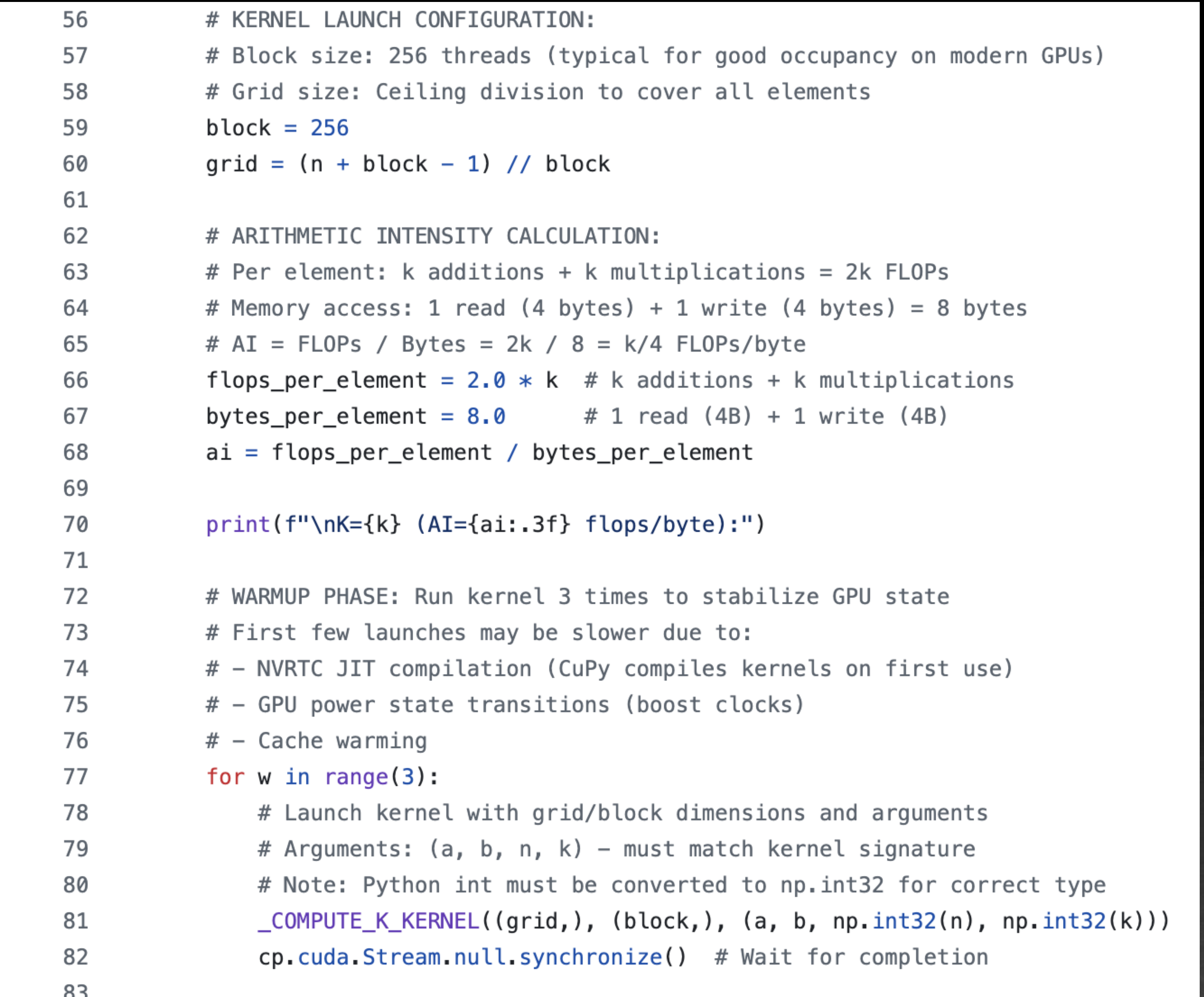
SETUP PYTHON TIMING CODE



```python
def run_sweep(n: int, k: int, a: cp.ndarray, b: cp.ndarray):
    """Measure kernel performance for a specific K value.

    This function implements GPU benchmarking best practices:

    1. WARMUP RUNS: Execute kernel 3 times before timing to eliminate:
       - JIT compilation overhead (NVRTC on first launch)
       - GPU frequency scaling effects
       - Cache cold-start effects

    2. MULTIPLE TIMED RUNS: Collect 10 timing measurements to:
       - Compute reliable mean performance
       - Detect timing variance (system interference)

    3. EVENT-BASED TIMING: Use CUDA events for GPU-side timing:
       - Eliminates CPU-GPU synchronization overhead
       - Provides microsecond precision
       - Correctly measures asynchronous kernel execution

    Parameters:
        n: Number of array elements
        k: Polynomial degree (controls arithmetic intensity)
        a: Input CuPy array (device memory)
        b: Output CuPy array (device memory)

    Returns:
        Tuple of (arithmetic_intensity, gflops, bandwidth, percent_peak)
    """

```

- Drop first couple of runs to account for compilation
- Calculate only the GPU compute timing with events

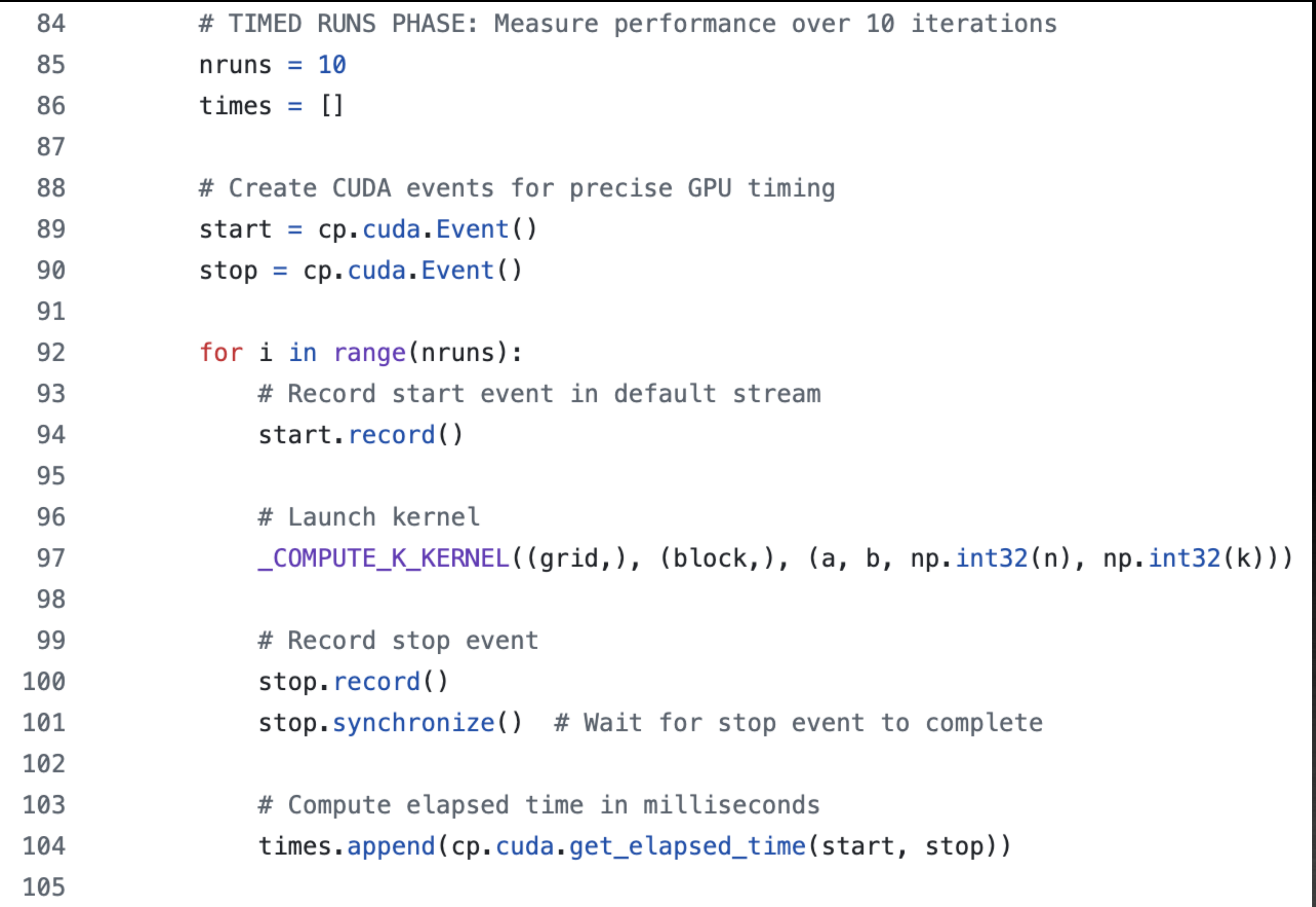
BENCHMARKING PYTHON CODE



```
    # KERNEL LAUNCH CONFIGURATION:
    # Block size: 256 threads (typical for good occupancy on modern GPUs)
    # Grid size: Ceiling division to cover all elements
    block = 256
    grid = (n + block - 1) // block

    # ARITHMETIC INTENSITY CALCULATION:
    # Per element: k additions + k multiplications = 2k FLOPs
    # Memory access: 1 read (4 bytes) + 1 write (4 bytes) = 8 bytes
    # AI = FLOPs / Bytes = 2k / 8 = k/4 FLOPs/byte
    flops_per_element = 2.0 * k  # k additions + k multiplications
    bytes_per_element = 8.0      # 1 read (4B) + 1 write (4B)
    ai = flops_per_element / bytes_per_element

    print(f"\nK={k} (AI={ai:.3f} flops/byte):")

    # WARMUP PHASE: Run kernel 3 times to stabilize GPU state
    # First few launches may be slower due to:
    # - NVRTC JIT compilation (CuPy compiles kernels on first use)
    # - GPU power state transitions (boost clocks)
    # - Cache warming
    for w in range(3):
        # Launch kernel with grid/block dimensions and arguments
        # Arguments: (a, b, n, k) - must match kernel signature
        # Note: Python int must be converted to np.int32 for correct type
        _COMPUTE_K_KERNEL((grid,), (block,), (a, b, np.int32(n), np.int32(k)))
        cp.cuda.Stream.null.synchronize()  # Wait for completion
```

- Set thread block/grid sizes for full GPU use
- Calculate FLOPS/byte
- Launches and syncs the benchmark kernel with CuPy for accurate results

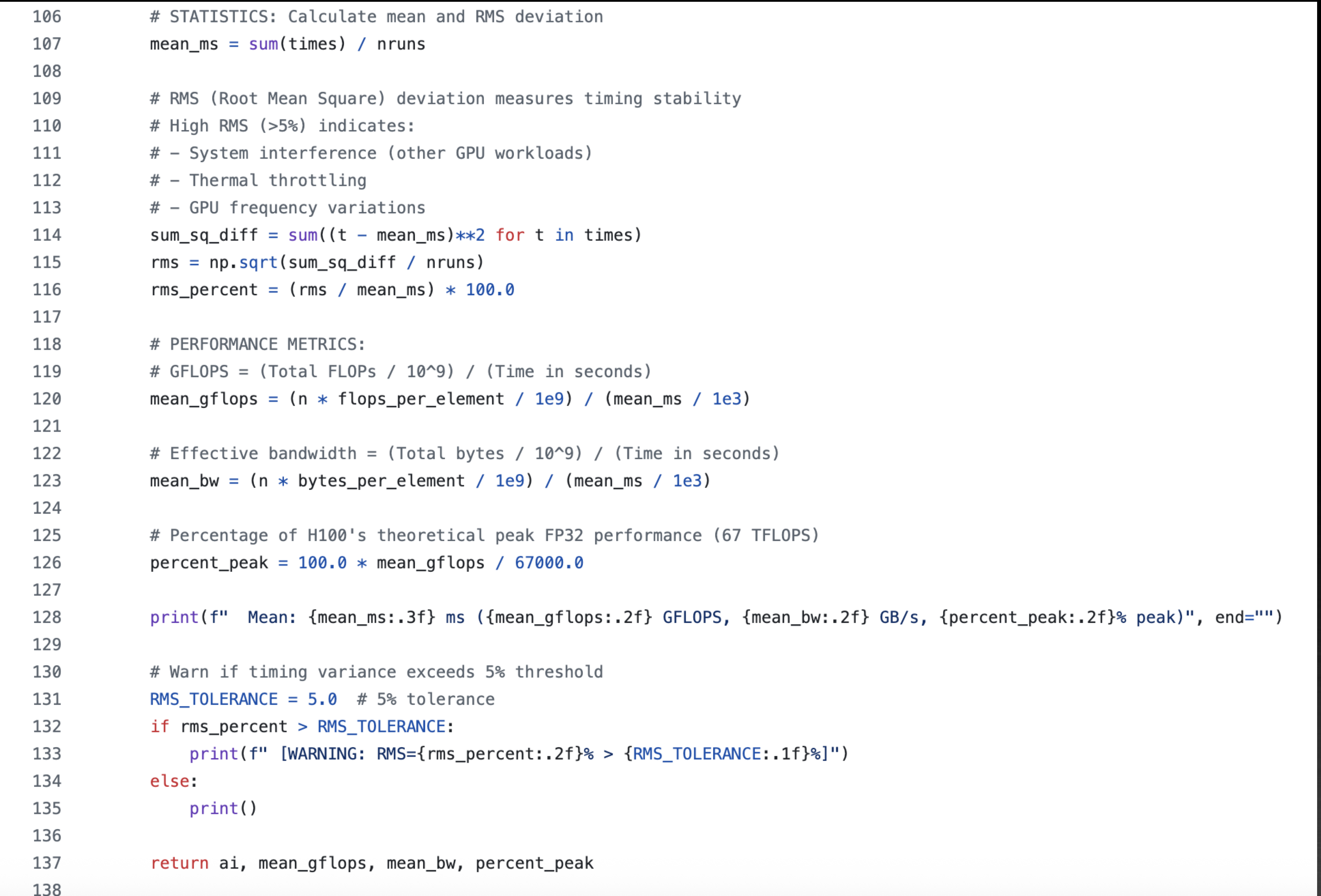
RUNNING OUR KERNEL



```python
    # TIMED RUNS PHASE: Measure performance over 10 iterations
    nruns = 10
    times = []

    # Create CUDA events for precise GPU timing
    start = cp.cuda.Event()
    stop = cp.cuda.Event()

    for i in range(nruns):
        # Record start event in default stream
        start.record()

        # Launch kernel
        _COMPUTE_K_KERNEL((grid,), (block,), (a, b, np.int32(n), np.int32(k)))

        # Record stop event
        stop.record()
        stop.synchronize()  # Wait for stop event to complete

        # Compute elapsed time in milliseconds
        times.append(cp.cuda.get_elapsed_time(start, stop))

```

- Tell the kernel how many threads to use, and how to arrange them
- Run our kernel 10 times
- Capture timing for each iteration

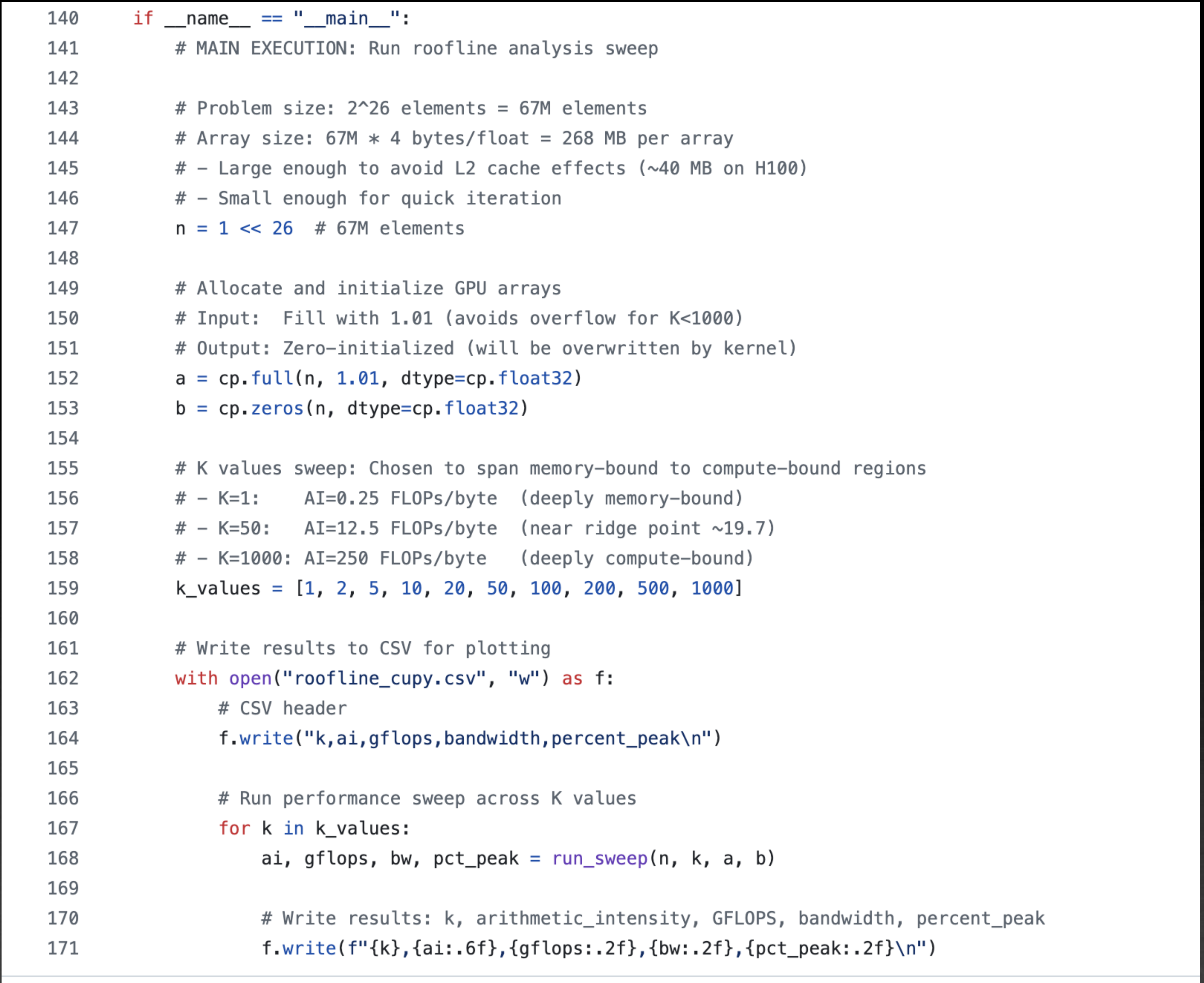
GATHER PERFORMANCE METRICS



```python
        # STATISTICS: Calculate mean and RMS deviation
        mean_ms = sum(times) / nruns

        # RMS (Root Mean Square) deviation measures timing stability
        # High RMS (>5%) indicates:
        # - System interference (other GPU workloads)
        # - Thermal throttling
        # - GPU frequency variations
        sum_sq_diff = sum((t - mean_ms)**2 for t in times)
        rms = np.sqrt(sum_sq_diff / nruns)
        rms_percent = (rms / mean_ms) * 100.0

        # PERFORMANCE METRICS:
        # GFLOPS = (Total FLOPs / 10^9) / (Time in seconds)
        mean_gflops = (n * flops_per_element / 1e9) / (mean_ms / 1e3)

        # Effective bandwidth = (Total bytes / 10^9) / (Time in seconds)
        mean_bw = (n * bytes_per_element / 1e9) / (mean_ms / 1e3)

        # Percentage of H100's theoretical peak FP32 performance (67 TFLOPS)
        percent_peak = 100.0 * mean_gflops / 67000.0

        print(f"  Mean: {mean_ms:.3f} ms ({mean_gflops:.2f} GFLOPS, {mean_bw:.2f} GB/s, {percent_peak:.2f}% peak)", end="")

        # Warn if timing variance exceeds 5% threshold
        RMS_TOLERANCE = 5.0  # 5% tolerance
        if rms_percent > RMS_TOLERANCE:
            print(f" [WARNING: RMS={rms_percent:.2f}% > {RMS_TOLERANCE:.1f}%]")
        else:
            print()

        return ai, mean_gflops, mean_bw, percent_peak

```

- Calculate and summarize GPU metrics for our kernel

FINALLY, TIME FOR THE MAIN!



```python
if __name__ == "__main__":
    # MAIN EXECUTION: Run roofline analysis sweep

    # Problem size: 2^26 elements = 67M elements
    # Array size: 67M * 4 bytes/float = 268 MB per array
    # - Large enough to avoid L2 cache effects (~40 MB on H100)
    # - Small enough for quick iteration
    n = 1 << 26  # 67M elements

    # Allocate and initialize GPU arrays
    # Input:  Fill with 1.01 (avoids overflow for K<1000)
    # Output: Zero-initialized (will be overwritten by kernel)
    a = cp.full(n, 1.01, dtype=cp.float32)
    b = cp.zeros(n, dtype=cp.float32)

    # K values sweep: Chosen to span memory-bound to compute-bound regions
    # - K=1:    AI=0.25 FLOPs/byte  (deeply memory-bound)
    # - K=50:   AI=12.5 FLOPs/byte  (near ridge point ~19.7)
    # - K=1000: AI=250 FLOPs/byte   (deeply compute-bound)
    k_values = [1, 2, 5, 10, 20, 50, 100, 200, 500, 1000]

    # Write results to CSV for plotting
    with open("roofline_cupy.csv", "w") as f:
        # CSV header
        f.write("k,ai,gflops,bandwidth,percent_peak\n")

        # Run performance sweep across K values
        for k in k_values:
            ai, gflops, bw, pct_peak = run_sweep(n, k, a, b)

            # Write results: k, arithmetic_intensity, GFLOPS, bandwidth, percent_peak
            f.write(f"{k},{ai:.6f},{gflops:.2f},{bw:.2f},{pct_peak:.2f}\n")
```

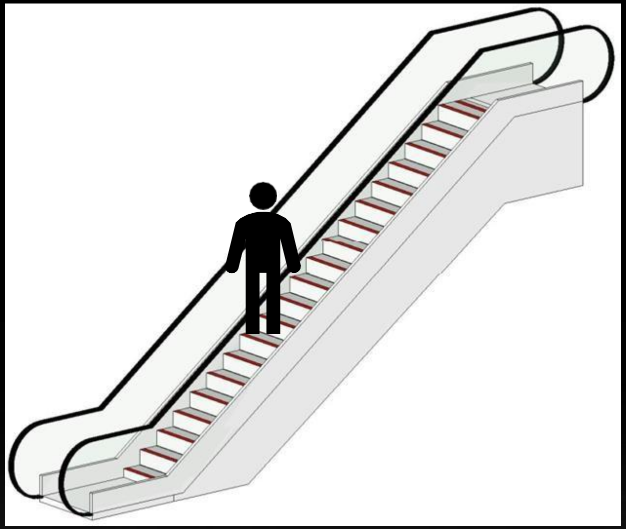
- Setup problem size
- Increasing k means more floating-point operations per memory access.
- Low k: memory-bound, limited by memory speed.
- High k: compute-bound, limited by GPU processing speed.

LITTLE'S LAW

LITTLE'S LAW
FOR ESCALATORS

Our escalator parameters:

- 1 person per step
- A step arrives every 2 seconds
  - **Bandwidth**: 0.5 person/s
- 20 steps tall
  - **Latency** = 40 seconds



One person in flight?

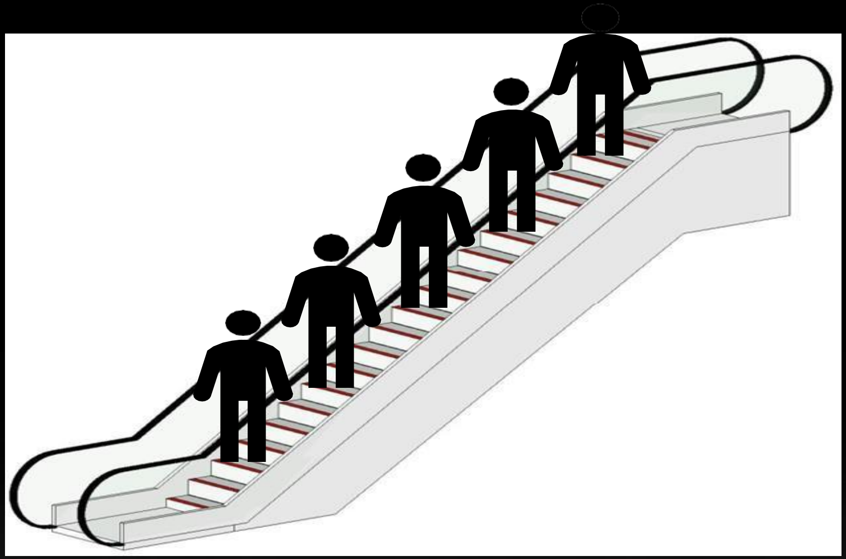
Throughput = 0.025 person/s

LITTLE'S LAW

FOR ESCALATORS

Our escalator parameters:

- 1 person per step
- A step arrives every 2 seconds
  - **Bandwidth**: 0.5 person/s
- 20 steps tall
  - **Latency** = 40 seconds



How many persons do we need in-flight to saturate bandwidth?

Concurrency = Bandwidth x Latency
= 0.5 persons/s x 40 s
= 20 persons

LITTLE'S LAW

FOR GPUS

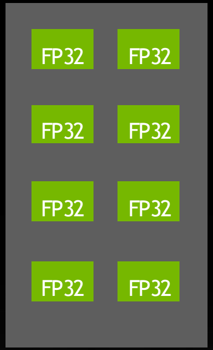
- How to maximize performance?

  1. Saturate compute units.
  2. Saturate memory bandwidth.

- Need to hide the corresponding latencies to achieve this.

  - Compute latencies.
  - Memory access latencies.

- Latencies can be hidden by having more instructions in flight.



FP32 Latency = 24 cycles 8 FP32 ops per cycle

Concurrency = Bandwidth x Latency = 8 x 24 operations in-flight

Is it always this clear cut?

WHAT OCCUPANCY DO I NEED?

GENERAL GUIDELINES

Rule of thumb: Try to maximize occupancy.

But some algorithms will run better at low occupancy.

More registers and shared memory can allow higher data reuse, higher ILP, higher performance.

Low Occupancy

Fewer threads per SM.

More resources per thread.

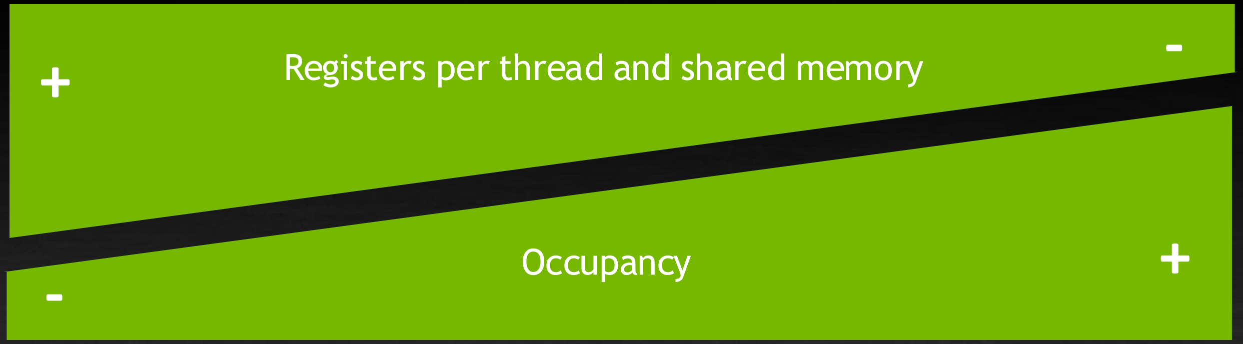
Enough instruction-level parallelism or GPU will starve!

High Occupancy

More threads per SM.

Fewer registers per thread.

Rely on thread parallelism to hide latencies!



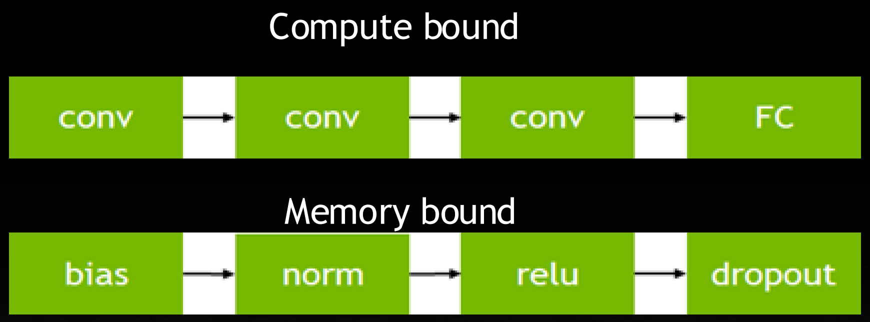
Complex algorithms ⟷ Simple algorithms

OTHER OPPORTUNITIES FOR OPTIMIZATION

FINDING PERFORMANCE OPPORTUNITIES

Models can be data bound by the data pipeline, compute or memory

- GPU utilization as it relates to model code
  - Time being spent on ops in every iteration
  - Time spent on GPU/CPU
  - Data types used for operations
- Bottlenecks could be attributed to
  - Input data pipeline: data loading, preprocessing etc
  - Compute (math) limited operations
  - Memory limited operations
  - Other aspects such as overall system tuning
- Categories of operations in DNNs based on bottleneck
  - Element wise: ReLU, memory bound
  - Reduction: Batch norm, memory bound
  - Dot product: Convolution, math bound



Compute heavy ops see speed-ups from GPUs

DEEP LEARNING OPTIMIZATION

Performance Analysis at System and DNN Level & Visualization

System Level Tuning

- System Tuning
  - Thread Synchronization, Multi GPU and node communication
  - Memory management & Kernel profiling
- Leveraging/Optimizing Hardware
- Input Pipeline Optimization
- Many others….

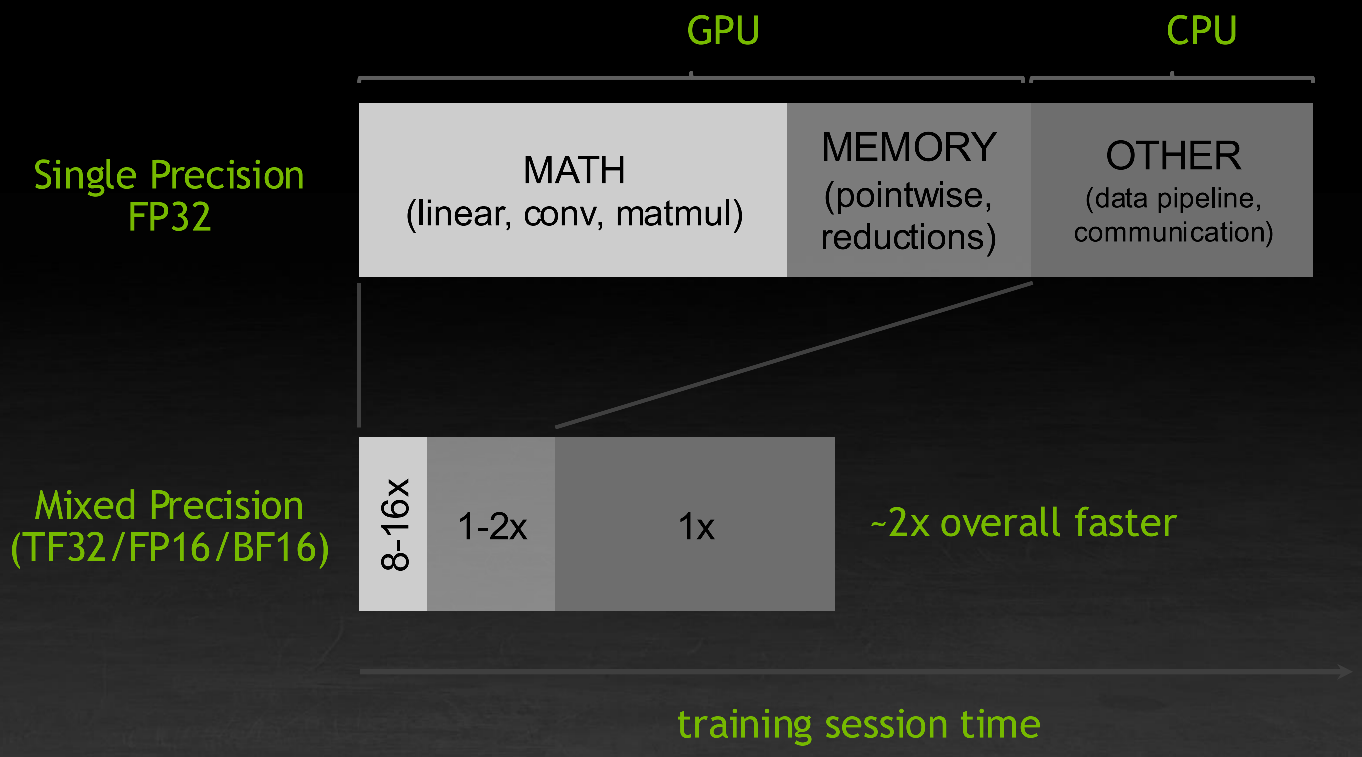
DNN Level Tuning

- Algorithm Techniques & Data Representations
- Pruning
- Calibration
- Quantization
- Many others….

LIMITS OF PERFORMANCE OPTIMIZATION

End-to-end perf depends on training composition

- **Amdahl's Law:**
If you speed up part of your training session (GPU work), then the remaining parts (CPU work) limit your overall performance


GPU
CPU
Single Precision FP32
MATH (linear, conv, matmul)
MEMORY (pointwise, reductions)
OTHER (data pipeline, communication)
Mixed Precision (TF32/FP16/BF16)
8-16x
1-2x
1x
~2x overall faster
training session time

DL PROFILING NEEDS OF DIFFERENT PERSONAS

Researchers

Fast development of best performant models for research, challenge and domains

Data Scientists & Applied Researchers

Reduce Training time, focus on data, develop and apply the best models for the applications

Sysadmins & DevOps

Optimized utilization and uptime, monitor GPU workloads, leverage hardware

WHICH OPTIMIZATIONS TO FOCUS ON?

SOLVING THE BOTTLENECKS

- Compute bound
  - Reduce instruction count.
    - E.g., use vector loads/stores.
  - Use tensor cores.
  - Use lower precision arithmetic, fast math intrinsics.
- Bandwidth bound
  - Reduce the amount of data transferred.
    - Optimize memory access patterns.
    - Lower precision datatypes.
    - Kernel fusion.
- Latency bound
  - Increase number of instructions and memory accesses in-flight.
  - Increase parallelism, occupancy.

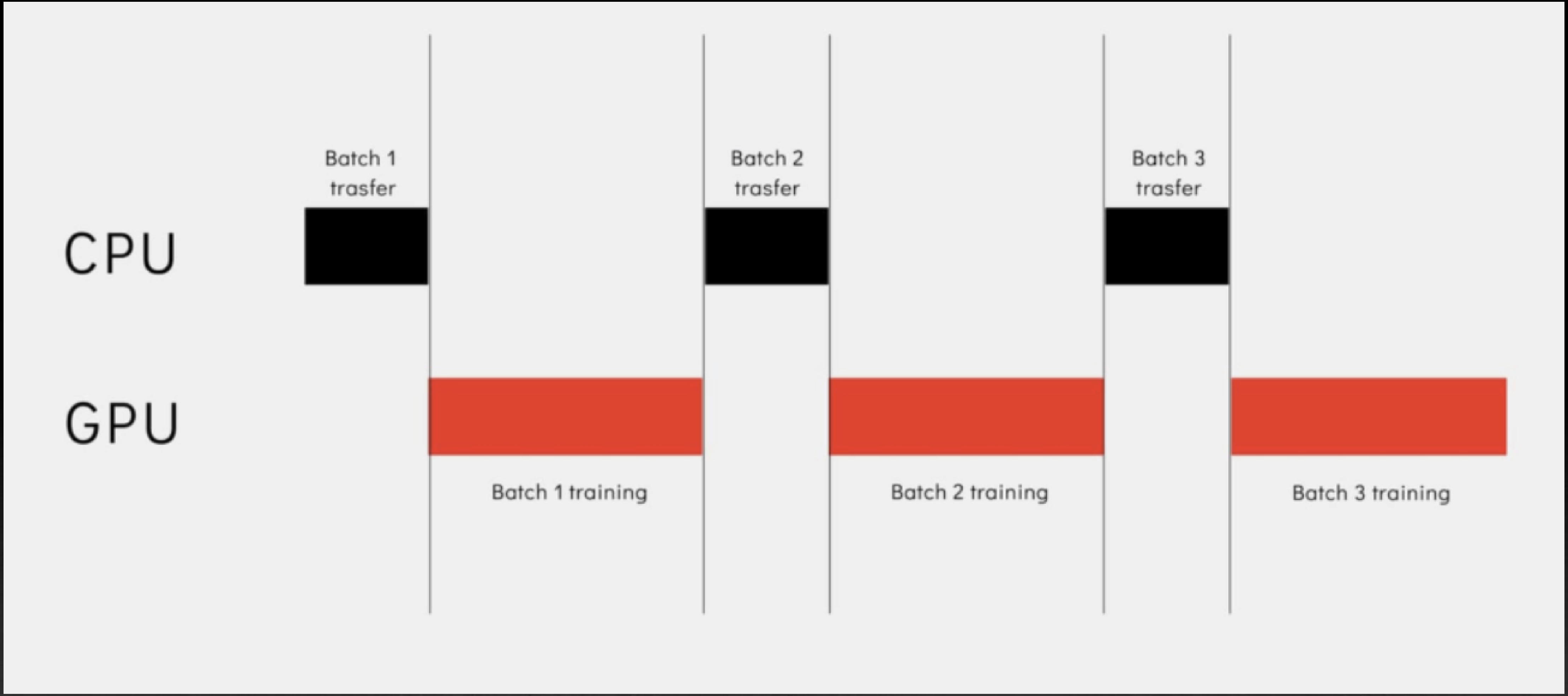
PYTORCH PERFORMANCE

WHAT ARE COMMON PROBLEMS IN PYTORCH

- I/O and Data:
  - **GPU starvation**
  - GPU sits idle waiting for the CPU to load and preprocess data.
  - This is common with large data sets that cannot be fully loaded into RAM.
- GPU utilization:
- Fully saturated with a heavy model
- Underutilized due to a lack of data

- Memory:
  - `RuntimeError: CUDA out of memory`
  - Forces a reduction in batch size.

WHAT ARE COMMON PROBLEMS IN PYTORCH?



https://towardsdatascience.com/improve-efficiency-of-your-pytorch-training-loop/

WHAT CAN WE DO ABOUT IT?

- I/O and Data:
  - Use the fundamental abstractions called Dataset and Dataloader
  - Dataset: This is the base class that represents a set of samples and their labels.
  - Dataloader: Wraps the dataset and makes it efficiently iterable.

- GPU utilization:
  - PyTorch Profiler to see Detailed analysis of CUDA operators and kernels.
  - Export results in .jsoninteractive format or visualization with TensorBoard.

- Memory:
  - Fuse operations to reduce memory access and kernel launch times
  - Only one kernel is launched for multiple pointwise operations
-

```
@torch.compile
def gelu(x):
    return x * 0.5 * (1.0 + torch.erf(x / 1.41421))
```

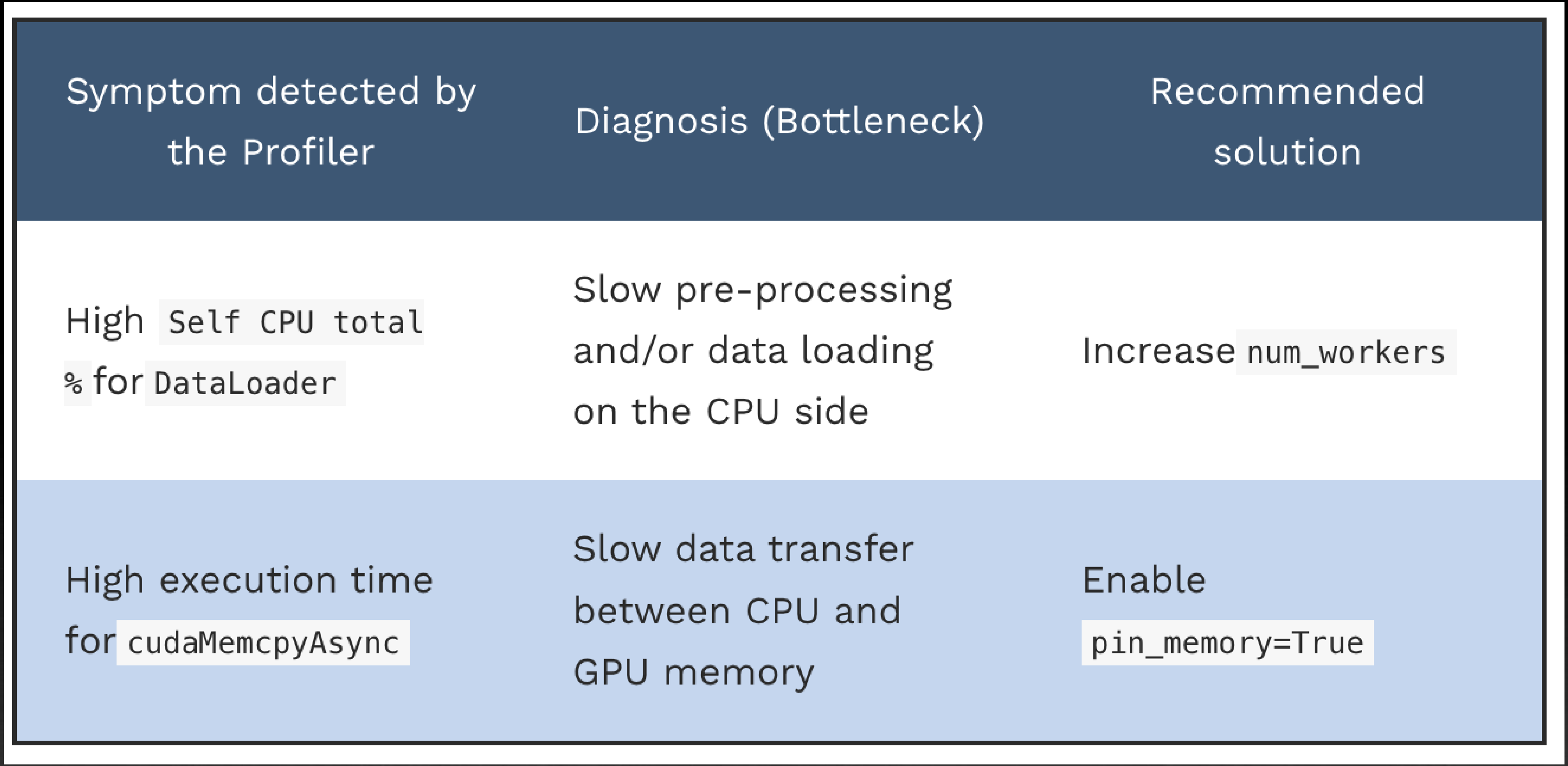
WHAT CAN WE DO ABOUT IT?

- **GPU utilization:**
  - Automatic Mixed Precision
  - Reduced precision can mean faster computations

- **GPU utilization:**
  - Increase batch size during training
  - Number of training examples utilized in one iteration.
  - Better utilization of GPU parallelism and faster convergence.

- **Memory:**
  - Memory Pinning for optimizing data transfer between the CPU and GPU
-

WHAT CAN WE DO ABOUT IT



Symptom detected by the Profiler	Diagnosis (Bottleneck)	Recommended solution
High `Self CPU total %` for `DataLoader`	Slow pre-processing and/or data loading on the CPU side	Increase `num_workers`
High execution time for `cudaMemcpyAsync`	Slow data transfer between CPU and GPU memory	Enable `pin_memory=True`

https://towardsdatascience.com/improve-efficiency-of-your-pytorch-training-loop/

WHERE CAN WE LEARN MORE?

- PyTorch Performance Tuning Guide
- PyTorch Optimization Guide
- NVIDIA Deep Learning Performance Guide
- Ultimate guide to PyTorch library in Python
- LLM Training on Grace Hopper