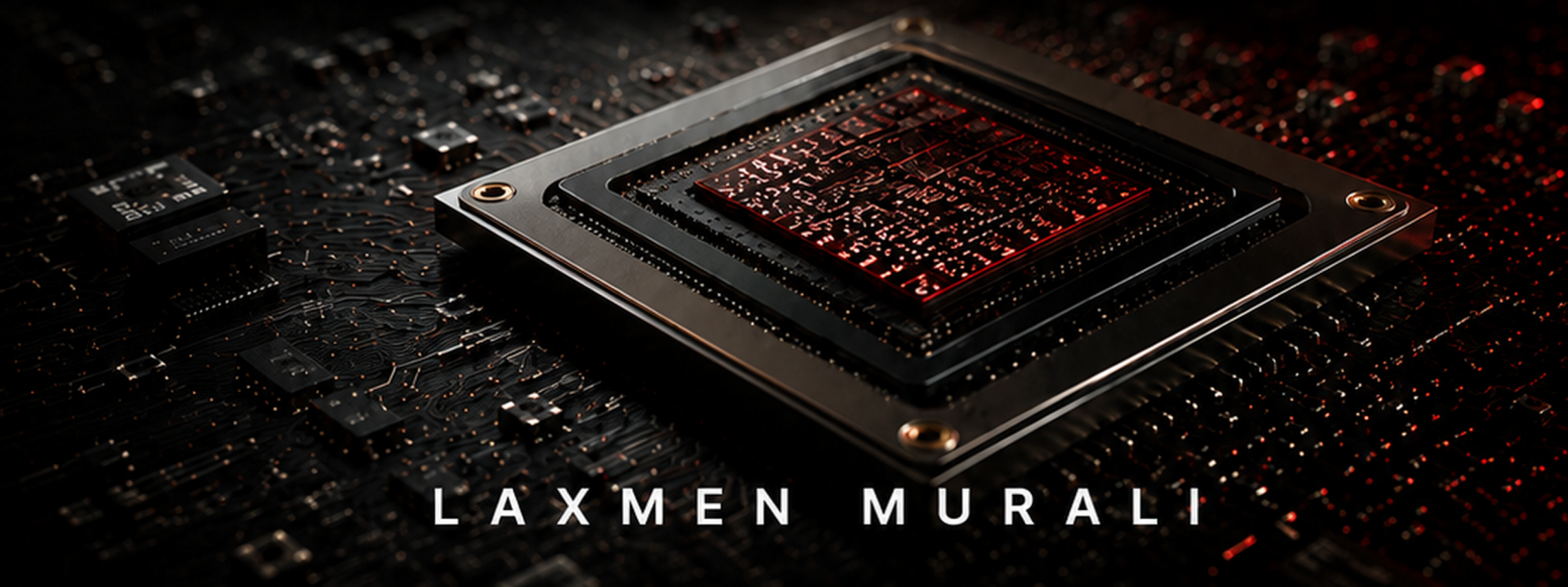
MACHINE LEARNING INFERENCE AND KERNELS



LAXMEN MURALI

Machine Learning Inference and Kernels

First Edition

Laxmen Murali

Copyright © 2026 by Laxmen Murali

All rights reserved. No part of this publication may be reproduced, stored or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning, or otherwise without written permission from the publisher. It is illegal to copy this book, post it to a website, or distribute it by any other means without permission.

Laxmen Murali has no responsibility for the persistence or accuracy of URLs for external or third-party Internet Websites referred to in this publication and does not guarantee that any content on such Websites is, or will remain, accurate or appropriate.

First edition

Contents

Preface **15**
A note on the numbers . . . 15
How to read this book . . . 16

Part I Understanding LLM Inference **17**

1 What Happens Between a Prompt and a GPU **18**
1.1 The stack . . . 18
1.2 The request to follow . . . 20
1.3 Text becomes tokens . . . 20
1.4 The request joins a queue . . . 21
1.5 Prefill: the whole prompt at once . . . 21
1.6 Decode: one token at a time, two hundred times . . . 22
1.7 Sampling, and text again . . . 22
1.8 Where the time actually went . . . 23
1.9 The five places a millisecond can hide . . . 24
1.10 How to read the numbers . . . 26
1.11 Running it yourself . . . 26
1.12 Where this request ends up . . . 28
1.13 Exercises . . . 28

2 The Transformer as an Inference Workload **30**
2.1 Notation . . . 30
2.2 One layer, drawn honestly . . . 31
2.3 Counting parameters . . . 31
2.4 Counting arithmetic . . . 33
2.5 Where attention actually sits . . . 34
2.6 The shape space, on models people actually ship . . . 35
2.6.1 What changes when you change one symbol . . . 36
2.7 Memory, not just parameters . . . 36
2.8 The reference implementation . . . 38
2.9 The same layer, two memory plans . . . 39
2.10 Checking the arithmetic against the code . . . 40
2.11 Exercises . . . 41

3 Prefill and Decode: Two Different Machines **43**
3.1 The same matrix, twice . . . 43
3.2 Arithmetic intensity . . . 44
3.3 What decode is actually limited by . . . 45
3.4 Batching, and where it stops working . . . 46
3.5 Throughput and latency are different questions . . . 47
3.6 The two machines end up in one step . . . 48

3.7 Measuring it . . . 48
3.8 Exercises . . . 51

4 The KV Cache and the Memory Math of Decoding 53
4.1 The redundancy . . . 53
4.2 The formula . . . 54
4.3 Real models . . . 55
4.4 Architectures that change the formula itself . . . 56
4.4.1 Sliding window attention: capping S . . . 56
4.4.2 Cross-layer sharing: cutting L . . . 57
4.4.3 What none of them change . . . 57
4.5 What this does to a real GPU . . . 58
4.6 Reducing it . . . 59
4.7 In InferEngine . . . 60
4.8 Exercises . . . 62

Part II GPU Architecture for AI Engineers 64

5 Latency, Throughput, and Metrics You Can Defend 65
5.1 Four clocks, one request . . . 65
5.1.1 Where the clock starts, and why two honest benchmarks disagree . . . 66
5.2 Throughput, and the question it hides . . . 67
5.3 The frontier . . . 68
5.4 Little's law, and why the batch is not free . . . 69
5.5 The mean is the wrong statistic . . . 70
5.6 The harness . . . 70
5.7 Warmup has two jobs . . . 72
5.7.1 Then check the clock reading itself . . . 72
5.7.2 The spread column, and the day it earned its place . . . 73
5.8 Exercises . . . 76

6 The Shape of a GPU 78
6.1 Inside an SM . . . 79
6.2 Four levels, two of which are fictional . . . 80
6.2.1 A fifth level, on newer parts . . . 81
6.3 What actually issues, and how often . . . 81
6.4 Divergence . . . 82
6.5 Filling the machine . . . 84
6.6 The tail . . . 84
6.7 Reading it off your own card . . . 85
6.8 Exercises . . . 86

7 The Memory Hierarchy 88
7.1 The levels . . . 89
7.2 Sectors, and why access pattern is everything . . . 91
7.3 Bank conflicts . . . 92
7.4 What the cache does that the model does not . . . 92
7.5 Latency, and how it is hidden . . . 93
7.6 Address translation, the level nobody draws . . . 94
7.7 Measuring your own card . . . 95
7.8 Exercises . . . 97

8 Tensor Cores and the Arithmetic of AI **99**
8.1 What a Tensor Core actually is . . . 99
8.2 Precision, and what each format buys . . . 101
8.3 The numbers . . . 102
8.4 Why this does not automatically help decode . . . 103
8.5 The smallest honest example . . . 103
8.6 Exercises . . . 104

9 Roofline, Arithmetic Intensity, and Bottleneck Reasoning **106**
9.1 The model . . . 106
9.2 Placing an LLM on it . . . 107
9.3 Same kernel, different verdict . . . 108
9.4 What the model cannot see . . . 108
9.4.1 The hierarchical version, which is worth drawing once . . . 109
9.4.2 Three ways the model lies, and what to do about each . . . 111
9.5 Using it in practice . . . 111
9.6 Exercises . . . 113

10 Reading a Spec Sheet Without Being Lied To **114**
10.1 Three kinds of number . . . 114
10.2 Reproducing the numbers . . . 115
10.3 The sparsity asterisk . . . 116
10.4 What the sheet cannot tell you . . . 116
10.5 The sheet against the machine . . . 116
10.6 The three numbers to ask for that are never printed . . . 118
10.7 Reading a sheet you have never seen before . . . 119
10.8 The generational trend that matters . . . 120
10.9 The most underrated line on the sheet . . . 120
10.10 Blackwell and beyond . . . 121
10.11 Exercises . . . 122

11 Your First CUDA Kernels **123**
11.1 What nvcc does . . . 123
11.2 The index . . . 124
11.3 Decoupling the grid from the data . . . 125
11.4 Wider loads . . . 126
11.5 The walkthrough . . . 127
11.6 Results . . . 128
11.6.1 The ceiling all four are chasing . . . 129
11.7 Getting the data there in the first place . . . 130
11.8 Launch geometry . . . 131
11.9 Error handling that catches errors . . . 132
11.10 InferEngine gets its first kernel . . . 133
11.11 Exercises . . . 134

Part III CUDA and GPU Kernel Engineering **136**

12 Global Memory Access Patterns **137**
12.1 Counting transactions . . . 137
12.2 Alignment . . . 138
12.3 Vectorized loads . . . 138
12.3.1 The same argument in FP16, where the book's models live . . . 139

12.4 Structure of arrays, and where the KV cache actually lives . . . 140
12.5 Layout is a performance decision . . . 140
12.6 The transpose, four ways . . . 142
12.6.1 Where the conflict becomes visible . . . 144
12.6.2 The floor and the ceiling this ladder sits between . . . 145
12.7 The read-only path and cp.async . . . 146
12.7.1 The double-buffered pipeline, written out . . . 147
12.8 Exercises . . . 149

13 Shared Memory, Tiling, and Bank Conflicts 151
13.1 Tiling is arithmetic intensity . . . 151
13.2 The tile size is a constrained optimization . . . 152
13.3 The sweep, measured . . . 154
13.4 Bank conflicts, properly . . . 156
13.5 Swizzling, and why production kernels do not pad . . . 157
13.5.1 The reason swizzle is not optional . . . 158
13.6 Dynamic shared memory and the opt-in limit . . . 158
13.7 A tiled reduction . . . 159
13.8 Exercises . . . 161

14 Warp-Level Programming 162
14.1 Shuffles . . . 162
14.2 The block reduction . . . 164
14.3 What the barriers actually cost . . . 165
14.4 Ballots and votes . . . 167
14.5 Scans, and the operation a reduction cannot do . . . 167
14.6 What the hardware does for you . . . 168
14.7 Warp specialisation, defined here because Part VI assumes it . . . 168
14.8 Independent thread scheduling . . . 169
14.9 Cooperative groups . . . 170
14.10Exercises . . . 172

15 Occupancy, Registers, and Latency Hiding 174
15.1 What occupancy is for . . . 174
15.2 The quantitative version . . . 175
15.3 The same ladder, run twice . . . 176
15.4 The four limits, and the constants behind them . . . 179
15.5 The other quantisation, at grid level . . . 180
15.6 The register cliff . . . 181
15.7 Spilling . . . 181
15.8 When occupancy genuinely is the problem . . . 182
15.8.1 Diagnosing a spill, and the cause that is not register pressure . . . 184
15.9 Exercises . . . 184

16 Synchronization, Streams, Graphs, and Launch Overhead 186
16.1 Barriers . . . 186
16.2 Atomics . . . 187
16.2.1 What the contention model gets right, and by how much it misses . . . 188
16.2.2 Compare-and-swap, and the atomics the hardware does not have . . . 189
16.3 Streams . . . 190
16.3.1 Timing, since every measured table in this book depends on it . . . 191
16.4 Launch overhead . . . 192
16.5 CUDA graphs . . . 193

16.5.1 The other answer to launch overhead: never return . . . 196
16.5.2 Capturing a graph is not the end of it . . . 196
16.6 Exercises . . . 197

17 GEMM I: From Naive to Tiled **199**
17.1 The shapes are not square . . . 199
17.2 v1: naive, and how bad it is . . . 200
17.3 v2: the same arithmetic, coalesced . . . 201
17.4 v3: shared memory tiling . . . 202
17.5 v4: rectangular tiles and register accumulators . . . 203
17.6 The ladder, in bytes per FMA . . . 205
17.7 What the roofline predicts, before you build anything . . . 206
17.8 The same ladder at four shapes, and why the ranking moves . . . 207
17.9 Results . . . 208
17.9.1 What the ladder is chasing . . . 210
17.9.2 Building this yourself . . . 211
17.10Exercises . . . 212

18 GEMM II: Register Blocking, Double Buffering, and Tensor Cores **214**
18.1 Two-dimensional register tiling . . . 214
18.2 Double buffering . . . 216
18.2.1 cp.async makes it real . . . 217
18.3 Tensor Cores . . . 217
18.3.1 The fragment layout, derived rather than looked up . . . 218
18.3.2 `wmma` is the teaching API, not the shipping one . . . 220
18.4 Split-K . . . 220
18.5 Epilogue fusion . . . 221
18.6 Where CUTLASS fits . . . 222
18.7 Results . . . 222
18.7.1 The ceiling these versions are chasing . . . 224
18.8 Where the ladder goes after this card . . . 225
18.9 Exercises . . . 226

Part IV Writing High-Performance LLM Kernels **228**

19 Reductions and the Online Softmax **229**
19.1 Softmax, and why the obvious version is wrong . . . 229
19.2 The online formulation . . . 230
19.3 Combining partial states . . . 231
19.4 The kernels . . . 232
19.4.1 What the prediction says, and where it is wrong on purpose . . . 232
19.4.2 What the four versions actually did . . . 232
19.4.3 The same operation through the framework . . . 233
19.5 The batch-one problem . . . 235
19.5.1 The exponential, and the reformulation every production kernel uses . . . 236
19.5.2 The statistic worth returning . . . 237
19.6 Reductions in general . . . 237
19.7 Exercises . . . 238

20 Memory-Bound Kernels: RMSNorm, RoPE, and Activations **240**
20.1 RMSNorm . . . 241
20.2 RoPE . . . 242

20.2.1 The kernel . . . 243
20.2.2 Long context changes the table, not the kernel . . . 244
20.3 SwiGLU and the activation family . . . 245
20.4 The general theory of the memory-bound kernel . . . 246
20.5 The same chain, measured at three shapes . . . 247
20.6 The same chain again, with Python removed . . . 248
20.7 Exercises . . . 250

21 Kernel Fusion **252**
21.1 What fusion is . . . 252
21.2 What can and cannot be fused . . . 253
21.3 The costs . . . 254
21.4 The three fusions that matter for LLM inference . . . 255
21.5 Horizontal fusion, which comes first and is usually forgotten . . . 257
21.6 Dequantization in the prologue, which is the one that pays now . . . 258
21.7 A decision procedure . . . 259
21.8 Exercises . . . 259

22 Profiling: Nsight Systems, Nsight Compute, and Compute Sanitizer **261**
22.1 Four tools, four questions . . . 261
22.2 Correctness first . . . 262
22.3 The timeline . . . 262
22.4 The kernel . . . 263
22.5 The metric table . . . 264
22.6 Distinguishing the two memory-bound failures . . . 265
22.7 The PyTorch profiler . . . 266
22.8 Driving the tools . . . 266
22.8.1 Nsight Compute . . . 266
22.8.2 Nsight Systems . . . 267
22.8.3 Compute Sanitizer, and the cost of using it . . . 268
22.9 Producing these views on your own machine . . . 268
22.10Exercises . . . 269

Part V Triton and GPU Code Generation **271**

23 The Triton Programming Model **272**
23.1 Blocks, not threads . . . 272
23.2 Masking, and the `other` argument . . . 273
23.3 What the compiler decides . . . 274
23.4 What you still decide . . . 274
23.5 Telling the compiler what it cannot prove . . . 275
23.6 Block pointers, and why modern kernels do not compute addresses . . . 276
23.7 What Triton does not give you . . . 276
23.8 Softmax, side by side . . . 277
23.9 Exercises . . . 278

24 Writing LLM Kernels in Triton **280**
24.1 Matmul, and the one optimization Triton cannot do for you . . . 280
24.2 The fusions from Chapter 21, in eight lines each . . . 282
24.3 Rows wider than a block . . . 282
24.4 Attention in Triton, which is the kernel everyone writes second . . . 283
24.4.1 The kernel above, against every kernel the library has . . . 285

24.5 What goes where . . . 286
24.6 Exercises . . . 289

25 Autotuning and the Triton Compiler Pipeline **291**
25.1 Autotuning . . . 291
25.2 Measuring a Triton kernel without measuring the cache . . . 293
25.3 The cache, and getting the JIT out of the serving path . . . 294
25.4 The pipeline . . . 294
25.5 Layout attributes . . . 295
25.6 The passes that matter . . . 296
25.7 A worked failure . . . 297
25.8 Exercises . . . 297

26 PTX and SASS: Reading What the GPU Actually Runs **299**
26.1 Why a virtual ISA exists . . . 299
26.2 Reading PTX . . . 300
26.3 The instructions worth recognising . . . 301
26.4 Writing PTX, not just reading it . . . 302
26.4.1 `lop3`, and why every INT4 dequantizer contains one . . . 302
26.4.2 `prmt`, the byte shuffle . . . 303
26.4.3 Cache hints, which are a serving lever rather than a kernel one . . . 303
26.5 SASS . . . 304
26.6 A worked reading . . . 305
26.7 Exercises . . . 306

27 torch.compile, TorchInductor, and Graph Capture **307**
27.1 The pipeline . . . 307
27.2 Graph breaks . . . 308
27.3 Recompilation . . . 309
27.4 What Inductor actually fuses . . . 310
27.5 Compiling once, and serving without Python . . . 311
27.6 Interaction with CUDA graphs . . . 312
27.7 Exercises . . . 315

Part VI Attention, FlashAttention, and KV Cache **317**

28 Attention as a Memory Problem **318**
28.1 The tensor that should not exist . . . 318
28.2 Why the intensity does not improve with scale . . . 320
28.3 Causal masking does not halve the cost . . . 321
28.4 MHA, GQA, MQA . . . 321
28.4.1 The fourth variant, and why it is not simply a smaller `H_kv` . . . 322
28.4.2 And the variant that removes the cache entirely . . . 322
28.5 Where the difficulty is, by phase . . . 323
28.6 Exercises . . . 324

29 FlashAttention: Tiling and Online Softmax in a Kernel **326**
29.1 The structure . . . 326
29.2 The rescale . . . 327
29.3 The IO complexity argument . . . 329
29.4 The tile size is chosen for you . . . 329
29.5 Causal masking, properly . . . 330
29.6 The kernel . . . 330

29.6.1 What fraction of the machine the fused kernel uses . . . 332
29.6.2 The controlled version of that observation . . . 334
29.6.3 The instrument, diagnosed . . . 337
29.7 Exercises . . . 339

30 FlashAttention-2 and FlashAttention-3 341
30.1 The non-matmul problem . . . 341
30.2 Parallelism at batch one . . . 342
30.3 FlashAttention-3 and Hopper . . . 343
30.4 The variants a serving system actually needs . . . 344
30.5 The published numbers . . . 345
30.6 What to implement . . . 346
30.7 Exercises . . . 347

31 Decode Attention: GQA, FlashDecoding, and the Batch-of-One Problem 349
31.1 Two operations . . . 349
31.2 GQA is a decode optimization . . . 350
31.3 The parallelism problem . . . 351
31.4 FlashDecoding . . . 352
31.5 Combining partials, which is where the correctness lives . . . 353
31.6 How many splits, and the cost of getting it wrong . . . 354
31.7 The layout matters more here than anywhere . . . 354
31.8 Exercises . . . 357

32 Paged KV Cache and PagedAttention 359
32.1 The reservation problem . . . 359
32.2 Mental model: the KV cache is virtual memory . . . 361
32.3 Blocks and block tables . . . 362
32.4 The allocator . . . 363
32.5 What the kernel has to do differently . . . 365
32.6 What the discontinuity actually costs . . . 366
32.7 Choosing the block size . . . 368
32.8 Sharing: where paging pays for itself twice . . . 370
32.9 What the allocator has to be, as code . . . 371
32.10 What paging does not fix . . . 373

33 Prefix Caching, Radix Trees, and Cache Reuse 376
33.1 KV is a pure function of the prefix . . . 376
33.2 The key is the hard part . . . 378
33.3 A radix tree over token IDs . . . 379
33.4 Eviction . . . 381
33.5 How long is a block worth keeping . . . 382
33.6 What it is worth, measured on a stated workload . . . 383
33.7 What a cache hit is worth depends on who is asking . . . 384
33.8 Hit rate is not latency . . . 385
33.9 Isolation between tenants . . . 386

Part VII Quantization and Advanced Inference 390

34 Number Formats and Numerical Behaviour 391
34.1 The two things a format can spend bits on . . . 391
34.2 Integer quantization is one equation . . . 393
34.3 Granularity is where the error actually goes . . . 394

34.4 Measuring what each scheme costs . . . 395
34.5 The scale does not have to be the maximum . . . 396
34.6 Accumulation precision . . . 397
34.7 The formats that arrived after FP8 . . . 399
34.8 The distribution that makes activations hard . . . 400

35 Weight-Only Quantization: INT8, INT4, GPTQ, and AWQ 404
35.1 What the bytes say . . . 404
35.2 Round-to-nearest, and its floor . . . 406
35.3 GPTQ: compensate as you go . . . 406
35.4 AWQ: change which weights get the precision . . . 408
35.5 Which one is better . . . 410
35.6 Where the quantized weights meet the kernel . . . 411
35.7 INT8 is a different decision from INT4 . . . 412
35.8 What the checkpoint has to record . . . 412
35.9 Calibration, which is where a quantized model actually goes wrong . . . 414
35.10 Below four bits . . . 415
35.11 The two places the prediction overstates . . . 415

36 Activation Quantization, SmoothQuant, and FP8 419
36.1 Why the fix that worked for weights does not work here . . . 419
36.2 FP8 solves this differently, and only on some hardware . . . 421
36.3 SmoothQuant: move the difficulty to where it can be handled . . . 423
36.4 What to leave in FP16 . . . 425
36.5 Static or dynamic scales . . . 426
36.6 A third answer: leave the outliers alone . . . 427
36.7 The three things to check before believing a quantized deployment . . . 428
36.8 FP8 KV cache . . . 429

37 Quantized Kernels, and Why Low Precision Is Not Automatically Fast 433
37.1 Packing . . . 433
37.2 The kernel . . . 435
37.3 Counting the unpack in CUDA . . . 436
37.4 The scales are traffic too . . . 437
37.5 Where the advantage ends . . . 438
37.6 What the profile shows . . . 439
37.7 Dispatch is part of the kernel . . . 440

38 Speculative Decoding 445
38.1 The exactness argument . . . 446
38.2 The cost model . . . 447
38.3 Drafters . . . 449
38.4 The drafters that are actually deployed . . . 450
38.5 Trees . . . 451
38.6 What the verification pass actually does . . . 453
38.7 The scheduling consequences . . . 454
38.8 Speculation in the finished engine . . . 454
38.9 What goes wrong . . . 455

Part VIII Inference Runtimes and Serving 458

39 Anatomy of an Inference Runtime 459
39.1 The components . . . 459

39.2 Request state . . . 461
39.3 The step loop . . . 462
39.4 The parts of a request that are not the model . . . 464
39.5 The control plane is a latency budget . . . 464
39.6 Where the time goes . . . 465
39.7 One step, end to end . . . 467
39.8 The two components nobody plans for . . . 468
39.9 What the executor should not know . . . 469

40 Batching, Continuous Batching, and the Scheduler Loop 472
40.1 Why static batching wastes what it wastes . . . 472
40.2 Iteration-level scheduling . . . 473
40.3 The replay . . . 474
40.4 Prefill in the same batch . . . 476
40.5 The ceiling . . . 477
40.6 Selective batching . . . 479
40.7 The policy questions the loop does not answer . . . 480
40.8 What continuous batching makes worse . . . 481
40.9 Measuring a scheduler honestly . . . 481

41 Chunked Prefill, Preemption, and Fairness 485
41.1 The stall . . . 485
41.2 Chunked prefill . . . 486
41.3 What chunk size costs . . . 488
41.4 The scheduler's other job: deciding what a step is worth . . . 489
41.5 Preemption . . . 490
41.6 Fairness . . . 492

42 Runtime Architectures Compared: vLLM, SGLang, TensorRT LLM, FlashInfer 496
42.1 The four design centres . . . 497
42.2 What they all have in common . . . 498
42.3 Where they actually differ . . . 499
42.4 Reading a system's source . . . 500
42.5 What to choose, and how to decide . . . 501
42.6 What you would build . . . 502
42.7 Version risk . . . 503

43 Serving, SLOs, and Benchmarking You Can Defend 506
43.1 Open loop and closed loop . . . 506
43.2 Where it actually breaks . . . 508
43.3 Goodput . . . 509
43.4 The workload is a parameter, and most benchmarks hide it . . . 511
43.5 Reproducibility, stated as a checklist . . . 512
43.6 The catalogue of ways benchmarks mislead . . . 512
43.7 A specification you can adopt . . . 513

Part IX Distributed Inference 516

44 Collectives, NCCL, and Interconnects 517
44.1 The five collectives . . . 517
44.2 Ring all-reduce and its cost . . . 518
44.3 The physical layer . . . 520
44.4 What tensor parallelism actually sends . . . 521

44.5 Making a collective overlap with something . . . 522
44.6 When the collective is the thing that is broken . . . 523
44.7 What NCCL does . . . 524

45 Tensor Parallelism **527**
45.1 Column parallel, then row parallel . . . 527
45.2 Heads, and what GQA does to them . . . 530
45.3 The central number . . . 531
45.4 The other things that get sharded, and one that must not . . . 533
45.5 Prefill scales, decode does not . . . 534
45.6 Making it cheaper . . . 534

46 Pipeline Parallelism and Prefill/Decode Disaggregation **538**
46.1 Splitting by layers . . . 538
46.2 Where the schedules differ . . . 541
46.3 Disaggregation . . . 541
46.4 Why disaggregation is a scheduling result rather than a systems one . . . 543
46.5 Composing the axes . . . 544

47 Sequence Parallelism, Expert Parallelism, and MoE Inference **548**
47.1 Splitting by tokens . . . 548
47.2 Ring attention . . . 549
47.3 Mixture of experts . . . 550
47.4 Routing is a memory-system problem before it is a modelling one . . . 551
47.5 Expert parallelism and all-to-all . . . 552
47.6 Load imbalance and capacity . . . 553
47.7 Multi-head latent attention, briefly . . . 554

Part X End-to-End Performance Engineering **558**

48 The Diagnostic Loop and Full-Stack Timeline Analysis **559**
48.1 Why This Matters . . . 559
48.2 The loop . . . 560
48.3 The layers, and what can see each one . . . 561
48.4 The waterfall, and why the residual comes first . . . 562
48.5 A step model with four terms and four axes . . . 563
48.6 Three regressions that look identical . . . 564
48.7 Localising, and admitting when you cannot . . . 566
48.8 Symptom to layer . . . 567
48.9 Attributing on real hardware . . . 569
48.10 The attribution, once more, with the moral . . . 570

49 When a Faster Kernel Makes a Slower System **573**
49.1 Case 1: fusion, benchmarked at the wrong shape . . . 574
49.2 Case 2: quantization exposes the control plane . . . 575
49.3 Case 3: batch raised throughput and cut goodput . . . 576
49.4 Case 4: prefix caching lowers the effective batch . . . 577
49.5 Case 5: one more GPU makes it slower . . . 578
49.6 Case 6: a faster kernel that buys nothing . . . 578
49.7 The general principle, stated properly . . . 579
49.8 A pre-mortem, in six questions . . . 580
49.9 When the local and global numbers legitimately disagree . . . 581

50 Capacity Planning, Cost Models, and Regression Engineering **584**
50.1 Four constraints, one of which binds . . . 584
50.2 Which constraint binds, and when . . . 586
50.3 From capacity to cost . . . 586
50.4 What an objective costs . . . 587
50.5 Does another GPU pay . . . 588
50.6 Choosing hardware with the same arithmetic . . . 589
50.7 Keeping the gains . . . 590
50.8 Interleave, or measure the weather . . . 591
50.9 What to assert on . . . 591

Part XI Building InferEngine **595**

51 InferEngine: Architecture and Codebase Review **596**
51.1 The map . . . 596
51.2 The file map . . . 597
51.3 What each module guarantees . . . 599
51.4 How tightly coupled is it . . . 600
51.5 One request, all the way through . . . 601
51.6 The deliberate gaps . . . 602
51.7 The test strategy . . . 603
51.8 Reading a performance number . . . 604

52 The Executor, the Kernel Layer, and the Paged KV Cache Manager **607**
52.1 Choosing a kernel . . . 607
52.2 One step's plan, traced . . . 609
52.3 The block manager, out of memory . . . 610
52.3.1 Eviction is part of admission . . . 611
52.4 Capturing the decode step . . . 612
52.4.1 Bucketing, and why it is not optional . . . 614

53 The Scheduler and the Continuous Batching Loop **618**
53.1 One iteration, five phases . . . 619
53.2 Two budgets, one admission test . . . 620
53.3 The knobs . . . 621
53.4 Priority, aging, and a bound you can state . . . 621
53.5 Preemption, assembled . . . 623
53.6 The growth reserve, and an optimum in the interior . . . 625
53.7 The prefix cache is a scheduling resource . . . 625
53.8 Watching it run . . . 627
53.9 The sweep . . . 627
53.10 What this chapter did not fix . . . 631

54 Quantization, Speculation, Multi-GPU, and the Final Optimization Pass **634**
54.1 The composed model . . . 635
54.2 Wiring in quantization . . . 637
54.3 Wiring in speculation . . . 637
54.4 Wiring in multi-GPU . . . 638
54.5 The final pass . . . 639
54.6 The journey . . . 641
54.7 What v1.0 is not . . . 643
54.8 Reading the production engines . . . 643

54.9 The answer to the question . . . 644

A Reference Hardware Tables **647**
A.1 The reader's card: RTX 3060 12GB . . . 647
A.2 The measurement device: RTX 3060 Laptop 6GB . . . 648
A.2.1 What it achieves . . . 648
A.2.2 The same card, measured four times . . . 649
A.2.3 Carrying these numbers to another card . . . 650
A.3 The datacenter baseline: A100 40GB SXM . . . 651
A.4 The current generation: H100 SXM5 . . . 651
A.5 The derived quantities in use . . . 652
A.6 Generational comparison . . . 652
A.7 Transferring the concepts to AMD . . . 653

B Notation and Symbols **654**
B.1 Model shape . . . 654
B.2 Workload and runtime . . . 654
B.3 Hardware . . . 655
B.4 Units, and the GB/GiB rule . . . 655
B.5 Tensor layout and launch geometry . . . 655
B.6 How figures are sourced . . . 656

C Environment Setup **657**
C.1 What you need, by part . . . 657
C.2 Driver and toolkit . . . 657
C.3 PyTorch, Triton and the version matrix . . . 658
C.4 Profilers . . . 658
C.5 Which exercises need which hardware . . . 658
C.6 Renting . . . 658

D The InferEngine Codebase Map **659**
D.1 The module ledger . . . 659
D.2 Interfaces, in one paragraph each . . . 661
D.3 Running it . . . 661

E Glossary **662**

F Annotated Bibliography **667**
F.1 Transformer architecture . . . 667
F.2 Attention kernels . . . 667
F.3 KV cache and serving runtimes . . . 668
F.4 Quantization . . . 668
F.5 Speculative decoding . . . 668
F.6 Distributed inference . . . 668
F.7 GPU architecture and performance modelling . . . 669
F.8 How to read a systems paper, briefly . . . 669

G Selected Exercise Solutions and Hints **670**
G.1 Level 1: full solutions . . . 670
G.2 Levels 2 and 3: hints and checkpoints . . . 671
G.3 Level 4: discussion notes . . . 672

References **673**
Model configurations . . . 675

Preface

This book is about the two halves of one question: what actually happens between a prompt arriving at a server and tokens streaming back, and how an engineer makes that whole path fast. The first half is large language model inference, the systems discipline: KV caches, batching, scheduling, quantization, speculation, serving. The second half is GPU kernel engineering: memory hierarchies, warps, tiling, Tensor Cores, CUDA and Triton, profiling. Most treatments pick one half. This book's argument is that they are one subject, because every serving decision is ultimately priced in kernel terms and every kernel matters only through the system it lands in. The two meet in the middle of the book, at attention, and stay welded for the rest of it.

This is a book built on arithmetic rather than folklore. Two model shapes, Ref-1B and Ref-8B, are defined precisely in Chapter 2 and reused in every calculation through Chapter 54, so that by the end the same numbers have been seen from twenty angles and can be produced from memory. A single cost model, introduced in Chapter 3 and refined through Chapter 41, prices every optimization discussed, and its final composed form reproduces every earlier chapter's result as a self-test. When the model says a fix is worth 1.72x, the derivation is on the page.

That emphasis shapes the form. Performance numbers age badly: they belong to a card, a driver, a library version and a workload, and a figure quoted without those is worth very little a year later. So the aim throughout is to teach performance to be predicted from first principles and then measured on the hardware in front of you, which is the skill that transfers, rather than to supply a table of someone else's results to be memorised.

A note on the numbers

Hardware specifications and published results are cited to the documents current at the time of writing. Vendor figures, driver behaviour, library versions and software interfaces change between releases, and any figure quoted here should be checked against the documentation for the version in use before it is relied upon.

Performance figures depend on hardware, driver, library version, thermal and power conditions, and workload shape. Figures given for one configuration do not transfer to another, and the arithmetic throughout is provided so that results can be recalculated for the configuration actually in use.

Product names and trademarks mentioned are the property of their respective owners and are used for identification only.

How to read this book

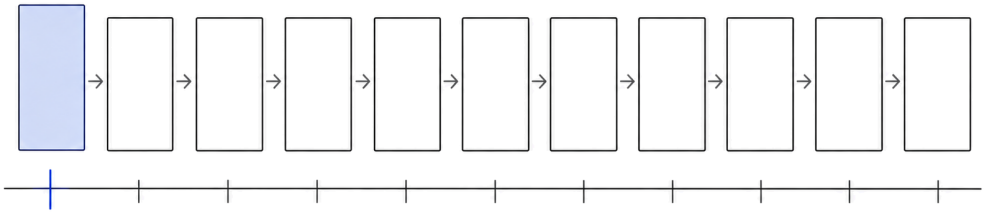
The parts are sequential and the dependency graph is real: Part I's inference arithmetic and Part II's hardware model are assumed everywhere; the kernel parts (III through V) build on each other; attention (VI) welds the halves; and everything after consumes everything before. A reader who already writes CUDA can skim III and IV but should not skip Chapter 9's roofline discipline or Chapter 15's occupancy model, which the rest of the book leans on constantly. A serving engineer who will never write a kernel can read I, II, VI, VII, VIII, X and XI and take the kernel chapters on faith, though Chapter 22's profiling and Chapter 26's what-the-GPU-actually-runs will repay even a systems-only reader.

The continuous project, InferEngine, grows in `code/inferengine/` from Chapter 2's reference decoder to Chapter 54's assembled engine: twenty-four modules, each carrying one invariant and one self-test, all runnable without a GPU. Appendix D is the map. The exercises are levelled 1 through 4, from comprehension to open-ended engineering; Appendix G has selected solutions, hints and checkpoints. The fenced boxes the book uses throughout, calc for worked arithmetic, mistake for instructive failures, experiment for things to run yourself, career for where each skill is used professionally, are all skimmable independently: reading only the calc boxes is a legitimate fast pass through the book, and reading only the mistake boxes is a legitimate second one.

Machine setup, including what runs without a GPU (most of it), is Appendix C. The notation is Appendix B, the glossary Appendix E, and the annotated bibliography, with a reading order for the primary sources, Appendix F.

PART I

Understanding LLM Inference



1

What Happens Between a Prompt and a GPU

Someone types a question into a chat box. Around a third of a second later the first word appears, and the rest arrive at a steady few dozen per second. Between those two moments a request crossed about a dozen distinct pieces of software, was turned into roughly forty five thousand individual GPU kernel launches, and pulled about half a terabyte of data out of memory. All of that to produce two hundred words.

Almost none of that is visible from either end. The person sees a text box. The model weights sit on disk as a few gigabytes of numbers. Everything interesting happens in between, and everything slow happens in between too.

This chapter walks that middle. It is deliberately shallow: every stage gets a paragraph or two, enough to name it, say what it does, and say what unit its performance is measured in. The chapters that follow are the deep version. What matters here is that by the end you have a map, and that every later chapter has somewhere to attach.

1.1 The stack

Here is the whole thing at once.

Read it top to bottom as a chain of translations. Text becomes tokens. Tokens become a scheduling decision. The scheduling decision becomes a batch of work for the runtime. The runtime walks a computation graph. Something turns that graph into kernels. Kernels become warps executing on streaming multiprocessors. Warps issue loads and stores against a memory hierarchy. When one GPU is not enough, some of those loads and stores become messages on an interconnect. What comes out the bottom is a number: tokens per second, and the latency distribution behind it.

Notice the right-hand column. Every layer has its own unit, and those units do not convert into each other cleanly. Doubling the kernels-emitted count does not halve tokens per second. Halving bytes per second at the memory system might do nothing at all if the bottleneck is somewhere else. Most confused performance work comes from reasoning in one layer's units about a problem that lives in another's.

> **ENGINEERING NOTE**
>
> The single most useful habit in performance work is asking, before anything else, which layer a problem lives in. An engineer who says "attention is slow" has not said anything actionable yet. An engineer who says "the decode attention kernel is reading the KV cache at 40 percent of achievable bandwidth" has named a layer, a unit and a target.

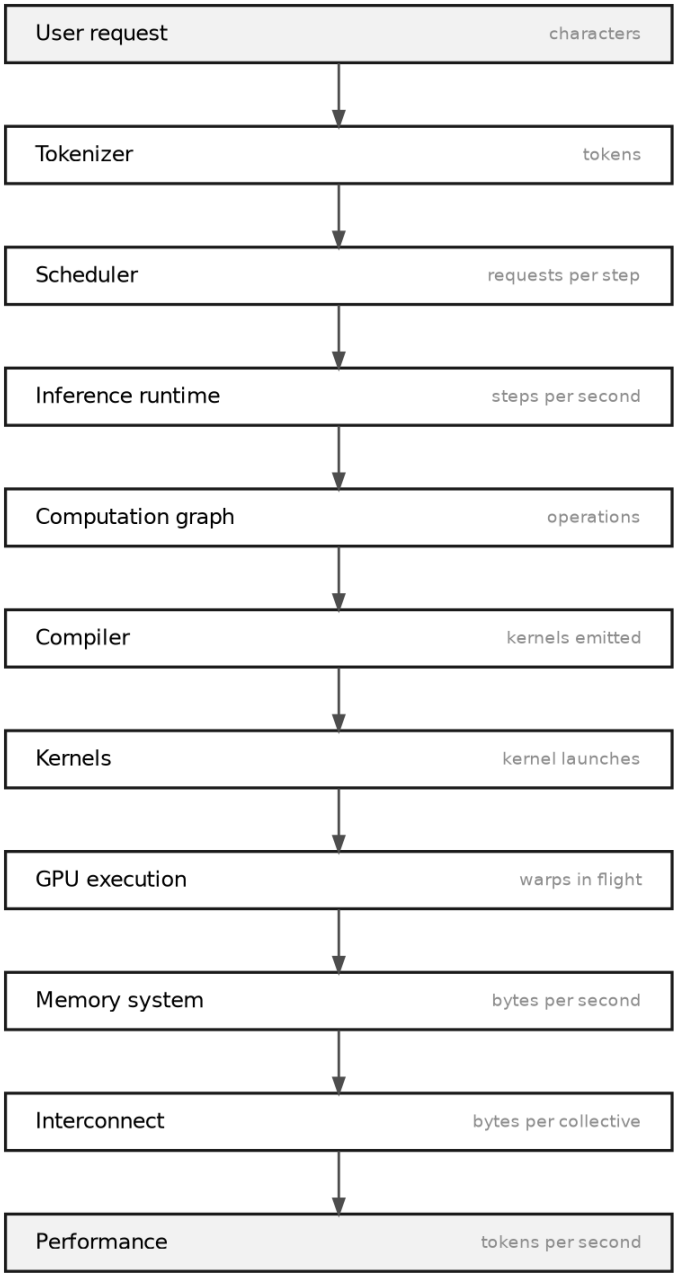


Figure 1.1. The path from a user request to a number that describes performance. Each layer transforms the request into a different representation, and each layer is measured in a different unit. Every part opens by returning to this column.

To make the units concrete, here is the one request this chapter follows, expressed in every layer's own currency:

Layer	This request, in that layer's unit
user	1 question, about 150 words back
tokenizer	40 prompt tokens in, 200 generated tokens out
scheduler	1 admission, 201 step-membership decisions
runtime	1 prefill pass + 200 decode passes
kernels	about 46,000 launches
GPU execution	about 1.4 seconds of streaming-multiprocessor time
memory system	about 500 GB read, almost all of it the same 2.5 GB of weights
interconnect	0 bytes, single GPU
performance	240 tokens, one latency distribution

Table 1.1 **DERIVED** from the arithmetic later in this chapter and in Chapter 2; the kernel count, the byte count and the GPU time are computed in this chapter's calc boxes. One request, nine descriptions, and none of the rows converts into its neighbours by simple division. Keeping this table straight in your head is most of what "thinking across the stack" means.

1.2 The request to follow

Concrete beats abstract, so fix a specific request and keep it for the whole chapter.

A user sends a prompt of 40 tokens and the model produces 200 tokens in reply. The model is **Ref-1B**: a 16-layer decoder-only transformer with a hidden size of 2048, 32 query heads, 8 key/value heads, a head dimension of 64, a feed-forward intermediate size of 8192, and a vocabulary of 128256. That shape matches a Llama-3.2-1B-class model, and it is deliberately small enough that its FP16 weights and a useful cache fit on a 12 GB consumer card.

Those seven numbers will follow you through every later chapter. Chapter 2 explains where each one comes from and what it costs. For now, treat them as the dimensions of the machine.

1.3 Text becomes tokens

The model has never seen a character. It operates on integers drawn from a fixed vocabulary, and the tokenizer is the piece that converts between the two.

Modern LLM tokenizers are subword: common words get one token, rare words get split into pieces, and arbitrary byte sequences can always be represented because the vocabulary bottoms out at individual bytes. This is why token counts and word counts never quite agree, and why the same sentence in English and in Telugu can cost very different numbers of tokens.

Tokenization is CPU work, it is fast, and it is almost never your bottleneck. It earns its place in this chapter for two reasons. First, it sets the value of `S_p`, the prompt length, which drives everything downstream: prefill cost, cache size, attention cost. Second, when tokenization does become a problem, it is usually in a way that looks like something else, such as a request whose latency is dominated by a 200 KB system prompt being re-tokenized on every turn.

Out of this stage comes an array of 40 integers.

1.4 The request joins a queue

Between the tokenizer and the GPU sits a scheduler, and it is more important than its position in the diagram suggests.

A serving system is almost never running one request. It is running dozens, at different stages, with different remaining lengths, competing for the same GPU and the same pool of memory. The scheduler decides, at every step, which requests run and which wait. That decision is made not once per request but repeatedly, between individual token steps, which is the idea that made modern LLM serving practical and which Chapter 40 covers in full.

For this request, the scheduler contributes something simple: a queue wait. The GPU was busy with other work, and the request sat for a few milliseconds before being admitted. Behind that small wait sits a surprising amount of policy: what to do when memory is short, which request to evict when it runs out, how to stop a long prompt from starving behind a stream of short ones. Chapter 53 assembles all of it; for now it is enough that admission is a decision, not a formality.

> **COMMON MISTAKE**
>
> It is tempting to treat queue wait as somebody else's problem, an artifact of load rather than of engineering. It is not. Queue wait is where a scheduling policy becomes visible to the user, and at high load it can exceed all of the GPU work combined. A system whose kernels are individually excellent and whose scheduler admits work badly is a slow system.

1.5 Prefill: the whole prompt at once

Now the GPU does something.

The model must build up internal state for the 40 prompt tokens before it can predict token 41. Because all 40 tokens are already known, they can be processed together, in parallel, in a single pass through all 16 layers. This is **prefill**.

Prefill is the phase that looks like classic deep learning compute. The dominant operations are large matrix multiplications: a `[40, 2048]` activation matrix multiplied against `[2048, N]` weight matrices, layer after layer. There is a lot of arithmetic and, relative to that arithmetic, not very much data movement, because each weight matrix is loaded once and used against all 40 token positions.

At the end of prefill, two things exist. The first is a probability distribution over the vocabulary for the next token. The second, and more consequential, is the **KV cache**: for every layer and every position, the key and value vectors that attention will need again on every subsequent step. Chapter 4 derives its size exactly. For now, note only that it exists, that it is large, and that it grows.

1.6 Decode: one token at a time, two hundred times

Prefill is one pass. Decode is 200 passes, and they cannot be done in parallel, because token 42 depends on token 41 having been chosen.

Each decode step feeds exactly one token into the model, walks all 16 layers, and produces one distribution over the vocabulary. Structurally it is the same computation as prefill. Numerically it is a completely different workload, and the difference is the most important idea in Part I.

Consider what a decode step actually has to do. To compute a single output vector, it must read every weight in the model. Ref-1B in FP16, with tied input and output embeddings, is roughly 2.5 GB of weights. Reading 2.5 GB once, on a card that sustains a few hundred gigabytes per second, sets a hard floor on how fast one decode step can be, and that floor has nothing to do with how fast the card can multiply. The arithmetic performed against those weights is tiny by comparison: one token's worth.

Prefill amortises that weight read across 40 positions. Decode amortises it across one. Same weights, same kernels, entirely different bottleneck.

> **MENTAL MODEL**
>
> Prefill is a delivery truck arriving fully loaded. Decode is the same truck making the same journey to deliver a single envelope. The journey costs what it costs; what changes is how much useful work you got out of it. Most of the techniques that follow, batching, quantization, speculative decoding, are ways of putting more into the truck.
> The analogy runs out quickly, and Chapter 3 replaces it with arithmetic intensity, which is the precise version of the same statement.

1.7 Sampling, and text again

Each step's output is a vector of 128256 logits, one per vocabulary entry. Something has to turn that into a single token.

The simplest policy is greedy: take the largest. In practice most systems apply a temperature, then restrict to the top-k entries or to the smallest set whose probabilities sum to top-p, then sample. These are cheap operations by GPU standards, but they are not free, and at batch size one they are one more small kernel in a chain of small kernels, which Chapter 16 shows can matter more than its FLOP count suggests.

The chosen token is appended to the sequence, its key and value vectors are written into the cache, and the loop runs again.

When generation stops, either at an end-of-sequence token or a length limit, the token IDs are converted back to text. In a streaming system this happens incrementally, token by token, which is why the user sees words appear rather than waiting for the whole reply.

Figure 1.2 One request, end to end. ILLUSTRATIVE, not measured.

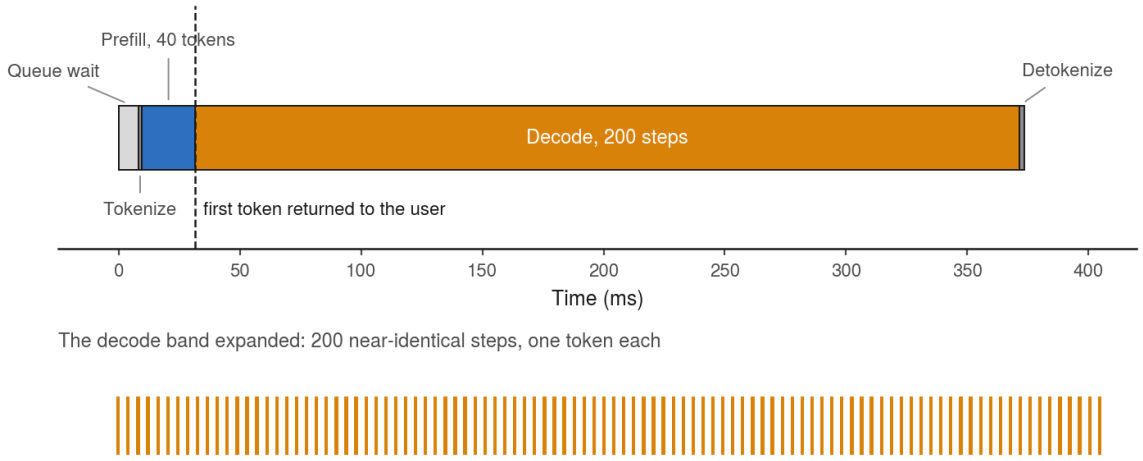


Figure 1.2. The same request drawn to scale. Prefill does far more arithmetic than any single decode step, and takes a small fraction of the total time, because there are two hundred decode steps and only one prefill. The lower panel expands the decode band into its individual steps.

1.8 Where the time actually went

Now put the stages on a common axis.

The shape of that chart is the shape of essentially every LLM request, and it explains why the field talks about two different latency metrics rather than one.

Time to first token covers everything up to the dashed line: queue, tokenize, prefill. It is what the user experiences as responsiveness. It scales with prompt length, because prefill does.

Time per output token is the width of one stripe in the lower panel. It is what the user experiences as reading speed. It barely depends on prompt length and depends heavily on how many other requests share the batch.

An optimization can improve one and worsen the other. Chapter 5 gives these metrics their proper definitions and Chapter 41 shows a scheduler knob that trades them against each other directly.

WORKED CALCULATION

The floor under each phase, before any measurement. DERIVED

The timeline's proportions are not free choices; they follow from four numbers you can compute before running anything. Ref-1B has 1.236 billion parameters, 2.47 GB in FP16, and its linear layers cost about 1.95 GFLOP per prefill token, charging the vocabulary head only for the final position, which is the convention Chapter 3 sets out (Chapter 2 derives both).

Prefill: 40 tokens costs `40 x 1.95` = 78 GFLOP. Take 8 TFLOP/s as a round achieved rate on the reference card (Chapter 10 derives the peak, Chapter 22 measures what fraction of it real kernels reach): about 10 ms.

Decode: every step must read the 2.47 GB of weights, and the RTX 3060 moves 360 GB/s (SPECIFICATION). The floor is `2.47 / 360` = 6.9 ms per token, and 200 tokens cost at least 1.37 s. That floor comes from bandwidth alone; the arithmetic in a decode step could be done thirty times over in the time the weights take to arrive.

So the request spends roughly 10 ms in prefill and at least 1,370 ms in decode: a hundred-to-one split, from four numbers and two divisions. This is the calculation hiding inside Figure 1.2, and learning to run it in your head for any model and card pair is worth more than most profiling

sessions. It is also the whole reason Part VII exists: the only way under a bandwidth floor is to shrink the bytes.

WORKED CALCULATION

How many kernels is that? DERIVED

Take one decode step in eager PyTorch, with no fusion and no graph capture. Per layer: input RMSNorm, three projections for Q, K and V, rotary embedding applied to Q and K, attention, output projection, a residual add, a post-attention RMSNorm, gate and up projections, the SwiGLU activation, the down projection, and a second residual add. That is roughly 14 kernel launches. Across 16 layers: `16 x 14 = 224`. Add a final norm, the vocabulary projection, and the sampling chain, and one decode step costs on the order of 230 kernel launches.

Over 200 tokens: about 46000 launches for a single short request.

And since every one of those 200 steps must read all 2.5 GB of weights, the request pulls roughly 500 GB out of memory in total, to emit 200 tokens.

Two consequences follow, and both get chapters of their own. Each launch has a fixed cost that is paid whether the kernel does much work or not, which is Chapter 16. And each of those 14 per-layer kernels reads its input from memory and writes its output back, which is the argument for fusion in Chapter 21.

1.9 The five places a millisecond can hide

Given a request that is too slow, there are exactly five broad answers, and what follows is a systematic treatment of each.

It is waiting for work to be admitted. The GPU is busy, the queue is deep, or the scheduler is making poor admission decisions. Lives in Parts VIII and X.

It is waiting for data. The kernels are running but starved, reading from memory at a fraction of what the hardware can deliver, or moving far more bytes than the algorithm requires. Lives in Parts II, IV and VI.

It is doing arithmetic inefficiently. The right data is present but the kernel uses the wrong units, the wrong tile shape, or leaves the Tensor Cores idle. Lives in Parts III and IV.

It is not on the GPU at all. Python overhead, kernel launch gaps, synchronization points, tokenizer work, framework dispatch. Lives in Parts V, VIII and X.

It is talking to another GPU. A collective sits on the critical path of every step and costs more than the compute it enabled. Lives in Part IX.

Diagnosing which of the five you are looking at, with evidence rather than intuition, is the skill everything that follows exists to teach. Figure 1.3 shows where each part does its work.

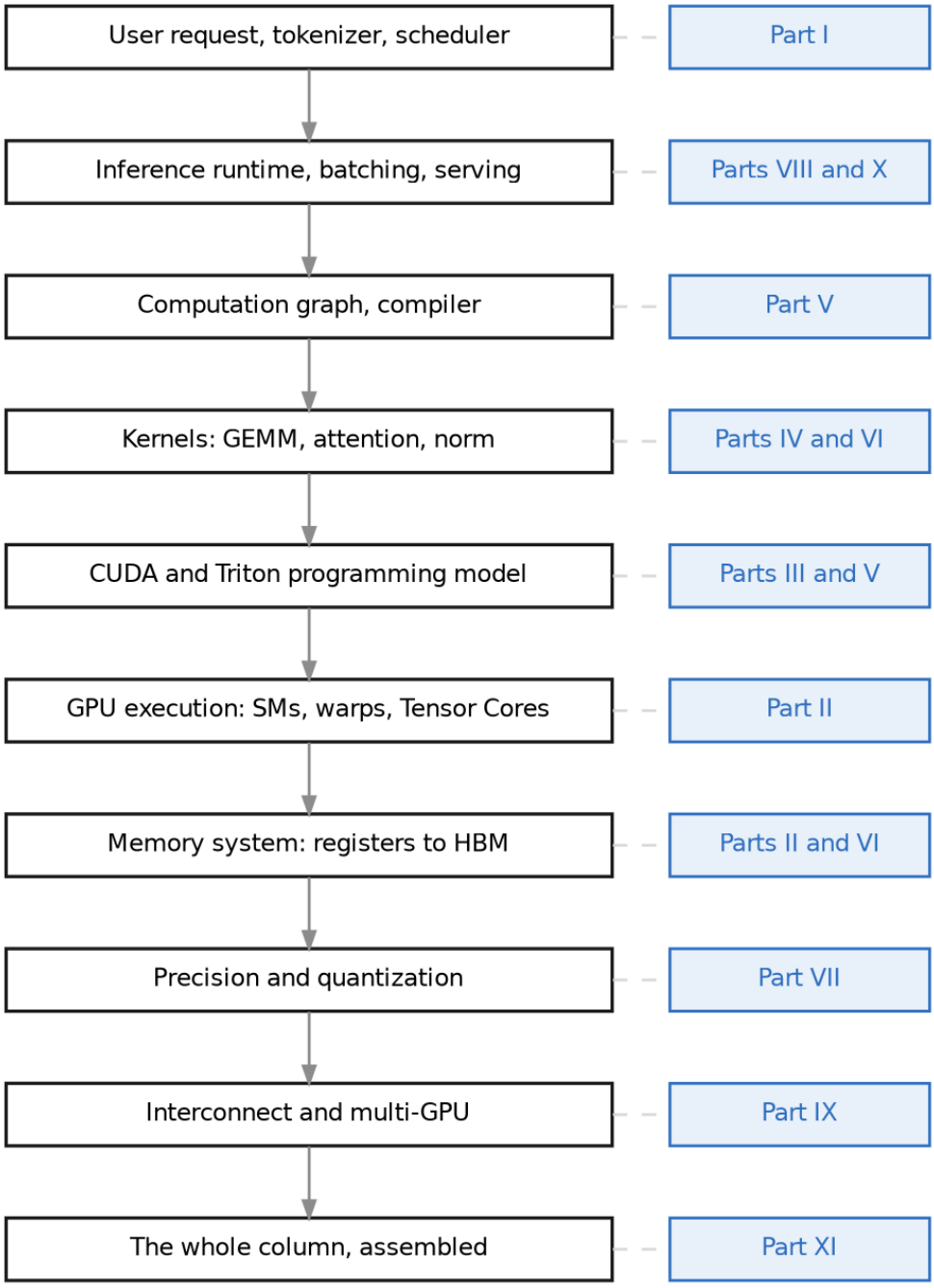


Figure 1.3. The parts mapped onto the stack. Reading in order walks the column from the top down, then reassembles it in Part XI.

1.10 How to read the numbers

One convention matters enough to state before any technical content arrives.

Performance numbers in technical writing are usually presented without provenance, and readers cannot tell a measured result from a vendor claim from a plausible-sounding estimate. The convention here is different: every quantitative claim carries one of five badges, and you will see them in table captions, chart titles and body text.

SPECIFICATION is printed in a vendor whitepaper or datasheet. The document and its version are cited. Example: the RTX 3060 provides 360 GB/s of memory bandwidth.

DERIVED is computed from specification values, with the arithmetic shown, so you can check it. Example: 3584 CUDA cores at 1.78 GHz performing two floating point operations per fused multiply-add gives 12.74 TFLOP/s of FP32 throughput.

PUBLISHED is a measurement reported in a cited paper or vendor engineering post, quoted together with the hardware, software version and workload it was measured on, so you know how far it generalises.

MEASURED is a number this book produced by running the code on hardware. The caption names the device, the date, the iteration count and the number of discarded warmups, because a measurement without those is not reproducible and therefore not a measurement. Every such table in this book comes from one machine, described in Appendix A, and the point of quoting it is never the absolute rate. It is the method and the fraction of peak.

ILLUSTRATIVE marks a figure that shows the shape of a result rather than a reading from a particular machine, and the caption says so. Figure 1.2 is one: the proportions follow from the arithmetic, while the absolute milliseconds depend on the card, the driver and the workload, and should be established on the hardware in question.

> **COMMON MISTAKE**
>
> The habit this convention is fighting is the reflex to quote a speedup without its configuration. “INT4 gives a 4x speedup” is not a fact, it is a fact-shaped object. Four times faster than what baseline, at what batch size, on what hardware, measured over which phase? Chapter 37 shows a case where the same quantization scheme gives 1.9x and 0.8x depending only on batch size. Both numbers are real. Neither is the answer.

1.11 Running it yourself

Everything above is observable in about sixty lines. This script loads a small model, runs prefill and decode as separate explicit calls rather than hiding them inside `generate`, and timestamps each stage.

```
# code/ch01/hello_inference.py
# Written against PyTorch 2.13 and transformers 4.x. Requires a CUDA GPU.
import os, time, torch
from transformers import AutoModelForCausalLM, AutoTokenizer

MODEL = os.environ.get("MODEL_ID", "TinyLlama/TinyLlama-1.1B-Chat-v1.0")
PROMPT = "Explain what a GPU streaming multiprocessor does, in two sentences."
MAX_NEW = 200

def stamp(marks, name):
```

```
    torch.cuda.synchronize()          # GPU work is async; without this you time
nothing
    marks.append((name, time.perf_counter()))

marks = []
stamp(marks, "start")

tok = AutoTokenizer.from_pretrained(MODEL)
model = AutoModelForCausalLM.from_pretrained(
    MODEL, torch_dtype=torch.float16).to("cuda").eval()
stamp(marks, "model loaded")

ids = tok(PROMPT, return_tensors="pt").input_ids.to("cuda")
stamp(marks, "tokenized")

with torch.inference_mode():
    # Prefill: all prompt tokens in one pass. Returns the KV cache.
    out = model(input_ids=ids, use_cache=True)
    next_id = out.logits[:, -1, :].argmax(-1, keepdim=True)
    cache = out.past_key_values
    stamp(marks, "prefill done")

    # Decode: one token per iteration, feeding the cache forward.
    generated = [next_id]
    for _ in range(MAX_NEW - 1):
        out = model(input_ids=next_id, past_key_values=cache, use_cache=True)
        next_id = out.logits[:, -1, :].argmax(-1, keepdim=True)
        cache = out.past_key_values
        generated.append(next_id)
    stamp(marks, "decode done")

text = tok.decode(torch.cat(generated, dim=-1)[0], skip_special_tokens=True)
stamp(marks, "detokenized")

print(text, "\n")
t0 = marks[0][1]
prev = t0
for name, t in marks[1:]:
    print(f"{name:<16} {1000*(t-prev):8.1f} ms   (cumulative {1000*(t-t0):8.1f}
    ms)")
    prev = t

prompt_len, gen_len = ids.shape[1], len(generated)
prefill_ms = 1000 * (marks[3][1] - marks[2][1])
decode_ms  = 1000 * (marks[4][1] - marks[3][1])
print(f"\nprompt {prompt_len} tokens, generated {gen_len} tokens")
print(f"time to first token   {prefill_ms:.1f} ms")
print(f"time per output token {decode_ms/max(gen_len-1,1):.2f} ms")
```

Two details in that listing are worth more than they look.

The `torch.cuda.synchronize()` call inside `stamp` is not defensive clutter. CUDA kernel launches are asynchronous: the Python call returns as soon as the work is queued, not when it finishes. Without the synchronize, every timestamp records how long it took to submit work, which is a real and interesting number but not the one you meant to measure. Chapter 5 makes this into a proper harness and Chapter 16 explains the asynchrony.

Passing `past_key_values` forward explicitly is what makes the decode loop cheap. Without it, every step would reprocess the entire sequence from the beginning, turning 200 cheap steps into 200 increasingly expensive ones. That is the KV cache, and Chapter 4 derives what it costs in bytes.

> **EXPERIMENT**
>
> **Change the prompt length, keep the output length fixed.**
> Run it with `MAX_NEW = 200` and a prompt of roughly 16 tokens. Record time to first token and time per output token. Now use a prompt of roughly 512 tokens, without changing `MAX_NEW`, and record both again.
> *Predict before you run it.* Which of the two metrics moves, and by roughly what factor?
> *Then explain the result.* One of the two should scale close to linearly with prompt length. The other should barely move at first and then begin to creep up as the prompt gets longer. The creep is not noise. Chapter 3 explains the first behaviour and Chapter 28 explains the second, and it is worth writing down your own explanation now so you can check it later.

1.12 Where this request ends up

One promise, so the length of the road ahead reads as a route rather than a wall. The naive loop in this chapter's script, run without its KV cache, would decode Ref-8B-class models at a few tokens per second, because every step would redo all the work before it. By the end, the same request runs through an engine you will have built yourself, and Chapter 54 prices the whole journey on one axis: the reference implementation to the finished engine is a factor of about 1,100 in decode throughput on one fixed workload, and every jump in between is a chapter with the arithmetic shown. About a factor of twenty of that comes from caching work instead of redoing it, another twenty from keeping the hardware full instead of idle, and the rest from spending precision to buy bandwidth. None of it requires anything exotic; all of it requires knowing, at each layer of Figure 1.1, what the machine is actually doing. That is the project.

> **WHERE THIS SKILL IS USED**
>
> Time-to-first-token and time-per-output-token regressions are triaged by inference engineers and AI performance engineers. Whoever can localise a regression to a layer of Figure 1.1 within an hour is the person the on-call rotation is glad to have.

1.13 Exercises

> **EXERCISES**
>
> **Level 1, understanding**
> **1.1** For each symptom, name the layer of Figure 1.1 you would investigate first, and say what unit you would measure it in. (a) Time to first token doubles when the prompt doubles. (b) Time per output token doubles when a second user connects. (c) Time per output token is

unchanged by batch size, at any batch size. (d) Total latency is fine at low load and terrible above 30 concurrent requests.

1.2 A colleague says "we made attention 2x faster and end-to-end latency dropped by 3 percent". Without knowing anything else about their system, give two explanations that are both consistent with those numbers.

1.2b Redo the calc box's two divisions for a model ten times larger (24.7 GB of FP16 weights) on a card with 2,000 GB/s of bandwidth and 200 TFLOP/s of achieved compute. Which phase's floor moved by more, and what happened to the hundred-to-one split?

Level 2, implementation

1.3 Extend `hello_inference.py` to report, per stage, both the wall-clock time and the number of tokens processed in that stage. Print tokens per second for prefill and for decode separately. They should differ by more than an order of magnitude; state which is larger and why before you look.

1.4 Add a flag that disables the KV cache, by dropping `past_key_values` and instead feeding the full sequence so far on every step. Measure time per output token with and without. Plot decode step time against step index for both. Describe the shape of each curve.

Level 3, optimization

1.5 Using only the script, find the batch size at which your card's time per output token first begins to rise noticeably. Batch identical prompts to keep it simple. Report the batch size and the step time at 1 and at that batch size. You now have your card's rough decode-batching sweet spot; keep the number, Chapter 3 explains it.

1.6 Reduce the total wall-clock time of a 200-token generation by at least 15 percent without changing the model, the precision, or the generated text. State exactly what you changed and why it worked. There are at least three valid answers available at this point.

Level 4, engineering challenge

1.7 You are handed a production complaint: "the model got slower this week". You have access to the serving logs, which record per-request prompt length, output length, queue wait, time to first token and end-to-end latency, and you know that no model weights, no kernels and no hardware changed.

Write down the three hypotheses you would rank highest, and for each one, the single query against those logs that would confirm or eliminate it. Then say which hypothesis you would test first and why. There is no single right answer, and the ordering of your tests is the part that is being assessed.

KEY TAKEAWAYS

An LLM request crosses about a dozen layers, and each layer measures itself in a different unit. Reasoning in the wrong layer's units is the most common source of wasted performance work.

Prefill processes the whole prompt in one parallel pass and is dominated by arithmetic. Decode processes one token per pass, two hundred times, and is dominated by reading the model's weights out of memory. They run the same kernels and have completely different bottlenecks.

Because there are many more decode steps than prefill passes, decode dominates total latency for typical request shapes, even though prefill performs far more arithmetic.

A single short request costs on the order of forty six thousand kernel launches and half a terabyte of memory traffic in an unoptimised eager implementation, which is why launch overhead, fusion and bandwidth are subjects rather than footnotes.

Every number is labelled specification, derived, published, measured or illustrative. Quoting a speedup without its configuration is not a fact.

2

The Transformer as an Inference Workload

Almost everything written about the transformer explains it as a modelling idea: attention lets every position look at every other position, and that turned out to work. That explanation is correct and it is useless for engineering purposes, because it says nothing about what the machine has to do.

This chapter rebuilds the same architecture from the other end. Not "what does this layer represent" but "how many bytes does it move, how many multiply-accumulates does it perform, and what shape are the matrices". By the end you will be able to take any decoder-only model configuration and compute, on paper, its parameter count, its arithmetic per token, its memory footprint, and which of its sublayers deserves your attention. That skill is used constantly and it takes about fifteen minutes to learn.

Everything here is arithmetic, derived from the model configuration alone.

2.1 Notation

Seven symbols describe the shape of a decoder-only transformer. They are fixed from here on and never redefined.

Symbol	Meaning	Ref-1B	Ref-8B
`L`	number of layers	16	32
`D`	hidden size, the residual stream width	2048	4096
`H`	number of query heads	32	32
`H_kv`	number of key/value heads	8	8
`d_h`	head dimension	64	128
`D_ff`	feed-forward intermediate size	8192	14336
`V`	vocabulary size	128256	128256

Table 2.1 SPECIFICATION The two reference model shapes used throughout. Ref-1B matches a Llama-3.2-1B-class configuration and is small enough to run on a 12 GB card. Ref-8B matches a Llama-3-8B-class configuration and is used for worked examples where larger numbers are more instructive.

Three relationships hold in both, and in most modern models:

`H * d_h = D`. The concatenated query heads are exactly as wide as the residual stream, so `W_Q` and `W_O` are square.

`H_kv < H`. This is grouped-query attention (AINSLIE23), and Chapter 4 shows that it is almost entirely a memory decision rather than a modelling one.

`D_ff` is roughly `3.5 * D`. The exact ratio varies, but the feed-forward block is always several times wider than the residual stream, which is why it dominates the parameter count.

Tensor shapes are written in storage order with the fastest-varying dimension last. A prefill activation is `[B, S, D]`.

2.2 One layer, drawn honestly

Read it as twelve operations. Normalize. Project to Q, K and V. Apply rotary position embedding to Q and K. Attend. Project back. Add the residual. Normalize again. Project up through two parallel matrices. Apply the gated activation. Project back down. Add the residual again.

Four of those twelve are matrix multiplications against large weight matrices, and they account for essentially all of the arithmetic. The other eight are elementwise or reduction operations that carry almost no arithmetic at all. Chapter 20 shows that those eight can still consume a third of a decode step, which is one of the more surprising results in Part IV, and the reason is visible already in this diagram: every one of them reads the entire `[B, S, 2048]` residual stream out of memory and writes it back.

> ENGINEERING NOTE
>
> The residual stream is the spine. Everything else is a detour that reads from it, computes something, and adds back into it. Once you see the architecture this way, a lot of design choices stop being mysterious. Normalization sits at the entrance to each detour so the detour sees a well-scaled input. The residual add sits at the exit so gradients and activations both stay bounded. And the width `D` appears in every single weight matrix, which is why changing it is the most expensive decision in a model configuration.

2.3 Counting parameters

Work through one layer, matrix by matrix.

The attention block holds four weight matrices. `W_Q` maps `D` to `H * d_h`. `W_K` and `W_V` each map `D` to `H_kv * d_h`, which is smaller by a factor of `H / H_kv`. `W_O` maps `H * d_h` back to `D`.

```
params_attn = D*H*d_h + 2*D*H_kv*d_h + H*d_h*D
```

The feed-forward block holds three. SwiGLU needs a gate projection and an up projection, both `D` by `D_ff`, and a down projection `D_ff` by `D`. This is worth pausing on: the gated activation costs three matrices where a classic two-layer MLP costs two, so a SwiGLU block with intermediate size `D_ff` has 1.5 times the parameters of a ReLU block with the same `D_ff`.

```
params_mlp = 3 * D * D_ff
```

Two RMSNorm scale vectors add `2*D`, which is negligible and which is carried anyway because it matters in Chapter 20 for a different reason.

Outside the layers there is the embedding table, `V * D`, a final norm, and the language modelling head, another `V * D` unless it shares weights with the embedding table.

> WORKED CALCULATION
>
> **Ref-8B, parameter by parameter.** DERIVED
> Attention, per layer:
> `4096*4096 + 2*4096*1024 + 4096*4096 = 16777216 + 8388608 + 16777216 = 41943040`
> MLP, per layer:
> `3 * 4096 * 14336 = 176160768`

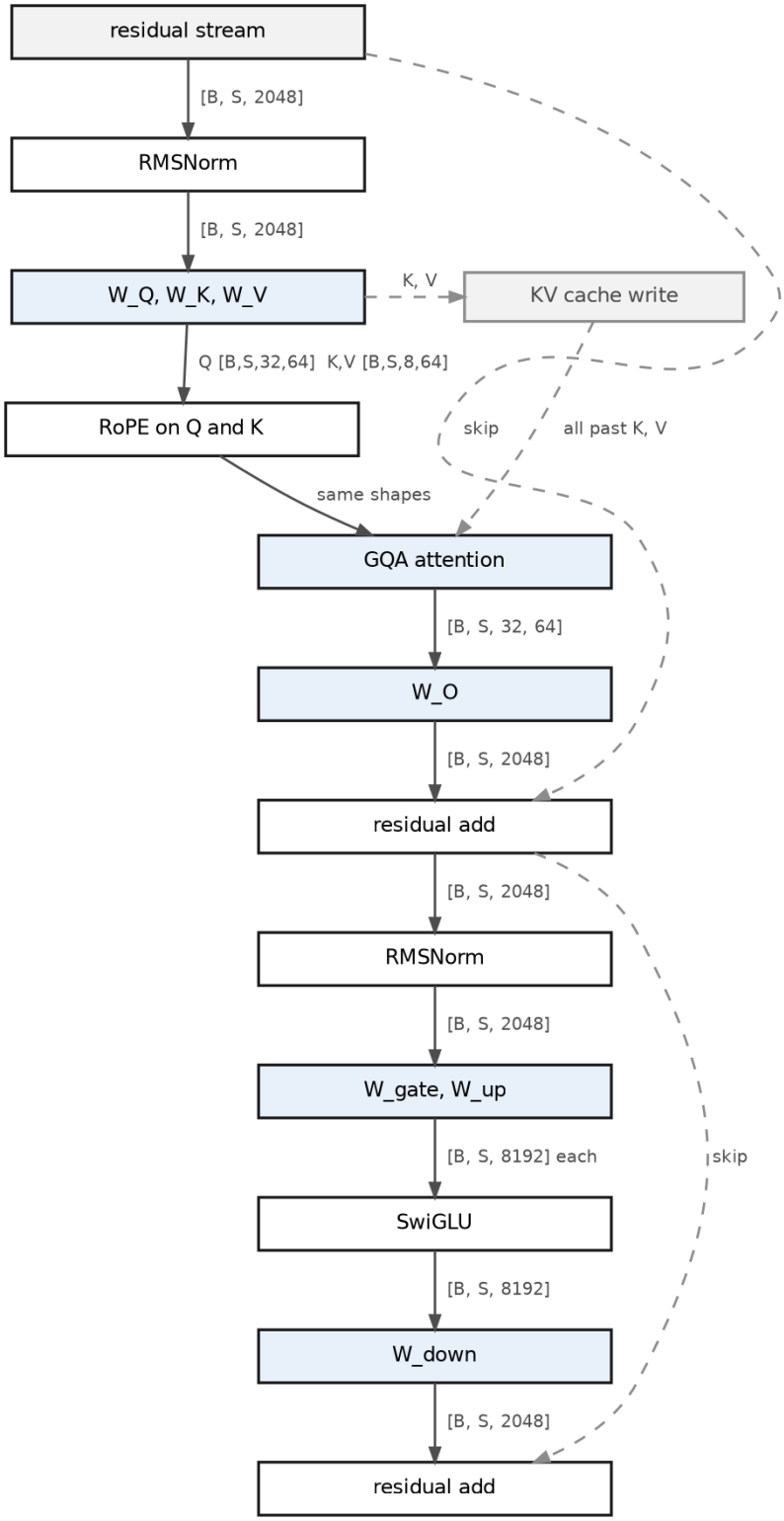


Figure 2.1. Every tensor in one Ref-1B decoder layer, with its shape. Solid boxes hold weights and do arithmetic. Dashed edges are the residual skips and the KV cache traffic. Note that the residual stream stays `[B, S, 2048]` from top to bottom: the layer is a function that maps the stream to itself.