ECE 408 GPT-2 Forward Pass — Final Report

Team: Temperature 1.5
Members: Alen Daniel (adani37), Sathya Mitiran (smj7),
Pranav Balasubramanian (pranavb7), Aryan Singh (asing271)

Contents

1 Summary **3**

2 Testing Strategy **3**
2.1 Automated build and testing (`test.sh`) . . . 3
2.2 Profiling pipeline . . . 4

3 Implementation Details **4**
3.1 req_4 — FlashAttention 2 . . . 4
3.2 req_5 — Local / Windowed Attention . . . 6
3.3 req_6 — KV-Caching . . . 7
3.4 op_7 — Configuration Sweeping . . . 9
3.5 op_8 — Constant Memory . . . 10
3.6 op_9 — `__restrict__` . . . 11
3.7 op_11 — XOR Swizzle on FA-2 Smem . . . 12
3.8 op_12 — Smem Padding (Tiled Matmul) . . . 13
3.9 op_13 — 256-Byte Aligned Activations . . . 13
3.10 op_16 — CUTLASS Fused fc1 + Bias + GeLU . . . 14

4 Performance Analysis of Milestone 3 Optimizations **15**
4.1 Bottleneck classification (M1 baseline) . . . 15
4.2 End-to-end comparison . . . 16
4.3 Per-optimization deep dive . . . 16
4.3.1 req_4 — FlashAttention-2 . . . 16
4.3.2 req_5 — Local Attention . . . 17
4.3.3 req_6 — KV-Caching . . . 17
4.3.4 op_7 — Sweep top configurations . . . 18
4.3.5 op_8 — Constant Memory γ/β . . . 18
4.3.6 op_9 — `__restrict__` . . . 19
4.3.7 op_11 — XOR Swizzle (FA-2) . . . 19
4.3.8 op_12 — Smem Padding (Tiled Matmul) . . . 19
4.3.9 op_13 — 256-Byte Aligned Activations . . . 20
4.3.10 op_16 — CUTLASS Fused fc1 + Bias + GeLU . . . 21

5 Pretty Pictures and Data **22**
5.1 End-to-end speedup waterfall . . . 22
5.2 KV-cache decode latency vs. token position . . . 23
5.3 Local-attention speedup vs. T . . . 23
5.4 op_13: NCU memory-warning elimination . . . 24
5.5 op_16: GeLU launch elimination . . . 25
5.6 Bank-conflict reduction (op_11 / op_12) . . . 26
5.7 op_7 sweep heatmap . . . 26
5.8 Speed-Of-Light summary across the final stack . . . 27
5.9 One-page summary table . . . 27

6 Interesting Findings **27**
6.1 FlashAttention-2 ended up *slower* than the M1 baseline at our test shape . . . 27
6.2 Silly bug in attention, when T happened to be to equal HS . . . 28
6.3 op_16's headline speedup was mostly cuBLAS, not the GeLU fusion . . . 28
6.4 The naive baseline matmul is L1/L2-bandwidth-bound, not compute-bound . . . 28
6.5 KV-cache speedup grows with generation length . . . 28

1 Summary

GPT-2 Optimized

End-to-end results.

- Total GPU kernel time on `test_gpt2` (B = 4, T = 64) drops from **249.8 ms** at the M1 baseline to **10.3 ms** under op_16 a **24.2**× speedup. Time also includes a fixed ∼36 ms H2D transfer of the model weights, identical across every variant.
- KV-cache decode (req_6) brings per-token latency from ∼311 ms (recompute baseline) down to ∼12.6 ms at the standard test shape, averaging **24.7**× across the 32 deterministic decode tokens.
- At larger batches the KV-cache speedup compounds: **74.9**× average and **117.4**× peak at B = 80, prompt = 32, gen = 128.

Optimization Chart

ID	Name	Points	Folder	Final speedup vs M1 baseline kernel
req_4	FlashAttention-2	6	`kernels_req_4`	0.66× attention chain (slower at T=64; see §4.3.1)
req_5	Local/Windowed Attention (W=128)	2	`kernels_req_5`	∼T/W× at large T ($\mathcal{O}(T \cdot W)$ vs $\mathcal{O}(T^2)$)
req_6	KV-Caching	12	`kernels_req_6`	24.7× avg per-token decode
op_7	Configuration Sweeping	3	`kernels_op_7`	tunes the others
op_8	Constant Memory (LayerNorm)	2	`kernels_op_8`	1.27× (LayerNorm kernel only)
op_9	`__restrict__`	1	`kernels_op_9`	∼1.0× at T=64 (bottleneck unchanged)
op_11	XOR Swizzle (FA-2)	3	`kernels_op_11`	1.37× (FA-2 kernel vs req_4)
op_12	Smem Padding (tiled matmul)	3	`kernels_op_12`	5.38× (matmul kernel)
op_13	256-byte Aligned Activations	2	`kernels_op_13`	sector-excess warning eliminated
op_16	CUTLASS Fused fc1 + Bias + GeLU	4	`kernels_op_16`	∼89× fc1 path; 24.2× end-to-end
Total		**38**		

2 Testing Strategy

2.1 Automated build and testing (`test.sh`)

We Wrote a single script to manage all outputs from nsys, ncu, and initial correctness testing (`test_gpt2`):

```
bash test.sh <suffix> [--test] [--nsys] [--ncu]
```

suffix : Specifies the optimization variant being tested (e.g., req_6, op16, etc.)

What the script does:

1. Validates that `kernels_<suffix>/` exists.
2. Backs up the current `kernels/*.cuh` to a temp directory.
3. Copies all `.cuh` files from `kernels_<suffix>/` into `kernels/`.
4. Calls `make test_gpt2 <suffix>` → produces `build/test_gpt2_<suffix>`.

5. Optionally submits `verify.slurm`, `verify_nsys.slurm`, `verify_ncu.slurm`.

6. Always restores the original kernel directory, even on a failure.

The kernel directory is restored **before** the slurm jobs run, but because each per-op binary in `build/` already has the swapped kernels compiled in, the slurm runs use the correct kernels.

2.2 Profiling pipeline

Outputs from (`test.sh`) routed to:

- `output/perf/nsys/verify_<suffix>.{out,err}` — text logs
- `output/perf/ncu/verify_<suffix>.{out,err}` — text logs
- `output/perf/reports/report_<suffix>.nsys-rep` — Nsight Systems UI
- `output/perf/reports/analysis_<suffix>.ncu-rep` — Nsight Compute UI

3 Implementation Details

For every optimization we document **(a)** the implementation, **(b)** the theoretical reason it should help, **(c)** the verification strategy, and **(d)** challenges faced.

3.1 req_4 — FlashAttention 2

File: `kernels_req_4/attention.cuh`

(a) Implementation

We implemented Algorithm 1 of Dao 2023 (FlashAttention-2 forward) as a templated CUDA kernel `flash_attention_forward_kernel<BR, BC, HS>`. Tile dimensions are compile-time template parameters so the compiler can fully unroll inner loops and place the per thread accumulator `out_reg[HS]` in registers and also to help with the op_7.

Launch size. One thread block per `(batch, head, query_tile)` triple:

```
constexpr int BR = 64;      // queries per tile
constexpr int BC = 64;      // keys/values per tile
int number_of_q_tiles = (T + BR - 1) / BR;
dim3 fa_grid(B, NH, number_of_q_tiles);
dim3 fa_block(BR, 1, 1);    // BR threads, one per query row
```

Each thread owns a single Q (query) row throughout the kernel, holding its running `(m_i, l_i)` softmax state (`m_i`: The largest logit seen so far, `l_i`: The total softmax score accumulated so far) and an `out_reg[HS]` accumulator in registers.

Shared memory: 3 (each bank) $\times$ $BR \times HS \times$ sizeof(float) $= 3 \times 64 \times 64 \times 4$ B $= 48$ KiB, exactly the A40 default static shared-memory limit per block. The three buffers hold Q, K, V tiles:

```
__shared__ float smem_q[BR * HS];
__shared__ float smem_k[BC * HS]; //BC = BR
__shared__ float smem_v[BC * HS];
```

Algorithm (mirrors the FA-2 paper):

1. Load the Q-tile cooperatively (one element per thread per `i`-iter, all 64 threads in the block).

2. For each KV-tile `j`:
    - Load stage K, V into shared memory.
    - Compute one row $S = Q_i \cdot K_j^T$ of the score matrix, scale by $1/\sqrt{HS}$, apply causal mask $k_{global} > q_{global} \to -\infty$.
    - Online-softmax rescale: $m_{new} = \max(m_i, row_max)$; if both are `-FLT_MAX` we `continue` (entire tile masked out).
    - $\alpha = \exp(m_i - m_{new})$; rescale the existing accumulator and add in the new tile's $P \cdot V$ contribution.
    - Update `l_i` and `m_i`.
3. After the loop, divide the accumulator by the final softmax denominator and write `out_base[query_global * HS + h]`.

Causal optimization. Because the kernel only ever reads K/V slots $k_{global} \leq q_{global}$, we only iterate over valid causal region.

```
int n_kv_tiles = (query_tile_end < T)
               ? (T + BC - 1) / BC           // standard
               : (query_tile_end + BC - 1) / BC;
```

Any tile fully above the query position is never iterated. When `max_new == -FLT_MAX` (all scores in the tile masked) we skip the rescaling step entirely instead of computing $\exp(-\infty - (-\infty))$.

(b) Why it should help

The M1 baseline generates the full attention probability matrix `B × NH × T × T` to global memory and reads it back twice (once for softmax, once for the $P \cdot V$ matmul). For B = 4, T = 64, NH = 12 this is 786.432 KB ($4 \times 12 \times 64 \times 64 \times 4$ float size) of wasted DRAM traffic; for T = 1024 in generation, 192 MB per layer.

FA2 reorders the computation so that K and V are streamed once per query tile and never re-read. The probability result is never written to DRAM it stays entirely in registers and is consumed in place by the $P \cdot V$ step. End-to-end DRAM traffic for the attention pipeline drops by $\mathcal{O}(T)$.

(c) Verification

- `test_gpt2` full-pass test. Max-err = **0.000069**, RMSE **0.000002**.

(d) Challenges

- **Coalesced loading:** Basic per-thread row load reads HS = 64 contiguous floats per thread, which warps into 32 separate 256 byte transactions. We rewrote the Q load with the striped index `int l = i * BC + query_row`, which makes warp 0 issue 64 contiguous loads at iteration `i` (a single 256 byte transaction). The same loading technique is used for K and V.
- **Tile rescaling skip:** Without the `m_new == -FLT_MAX` skip, the first KV-tile of a fully out of window Q row produced NaNs from $\exp(-\infty - (-\infty)) = \exp(\text{NaN})$.
- **Translating the pseudocode:** Moving from the paper into CUDA implementation required significant debugging.

3.2 req_5 — Local / Windowed Attention

Files: `kernels_req_5/local_attention.cuh`, `kernels_req_5/local_softmax.cuh`.

(a) Implementation

As per the guide, window size is fixed at $\mathbf{W = 129}$ (defined as `#define WINDOW 129` so that the window covers the current token plus 128 preceding ones, matching the verifier's expected output). also in the CPU reference uses an inclusive range [q_t - 128, q_t].

We implemented a **non-flash (M1-style) windowed attention** rather than specializing the FA-2 kernel, because:

1. The non-flash version has a much smaller probability tensor (`B` × `NH` × `T` × `W` instead of `B` × `NH` × `T` × `T`), so its memory advantage per token is large.
2. Each kernel is simple and the structure mirrors the original M1 attention, which made the windowed implementation easier.

Three kernels.

`preattention_scores_kernel` instead of computing a full `B` × `NH` × `T` × `T` score matrix, we materialize a packed `B` × `NH` × `T` × `W` matrix where the last dim indexes window offsets:

```
int t1 = (x / WINDOW) % T;          // query position
int w  = x % WINDOW;                // window offset
int t2 = t1 - w;                    // implicit key position
if (t2 >= 0) {
    sum = \Sigma_i q[t1, i] * k[t2, i];   // standard dot product
} else {
    sum = -FLT_MAX;                    // out-of-window -> masked
}
preatt[..., t1, w] = sum;
```

Keys outside the window are never loaded; edge cases at $t1 < W$ are handled by the $-\infty$ mask without separate kernels.

`local_softmax_forward_kernel` (`local_softmax.cuh`) the standard 3-pass softmax (max, sum-of-exp, normalize), but indexing in row-major across `WINDOW` rather than `T`.

`attention_v_kernel` gather and weighted-sum:

```
for (int w = 0; w < WINDOW; w++) {
    int t2 = t1 - w;
    if (t2 >= 0) sum += att[..., t1, w] * v[..., t2, hs];
}
```

(b) Why it should help

Per-step attention work drops from $\mathcal{O}(T^2)$ to $\mathcal{O}(T \cdot W)$. At W = 128:

- T = 64 (M3 default test) → unchanged work (windows already cover full sequence).
- T = 256 → ∼2× reduction in score-matrix DRAM traffic.
- T = 1024 → ∼8× reduction.

The crossover where windowed starts helping is near $T \approx W$.

(c) Verification

`local_attn_verify.cu` (filled in by us) allocates device-side buffers, runs the full GPT-2 forward with our windowed kernel substituted, and calls `compare_arrays` against the supplied expected outputs. Per the M3_README, **deviations are expected** because windowed math is not equal to full attention; the expected outputs were generated by the staff using the same window scheme so they match.

(d) Challenges

No major challenges, since we used the M1 attention kernels, which made it straightforward to integrate windowed attention. Additionally, most of the testing guidance and implementation hints were provided in `cpu_kernels/local_attention.cuh`, `local_attn_verify.cu`, and the corresponding Slurm script.

3.3 req_6 — KV-Caching

File: `kernels_req_6/attention.cuh`

(a) Implementation

KV caching: once a prefix has been processed, its K and V projections are fixed and can be re-used for every subsequent token. Our implementation follows the given template.

Cache layout (a single continuous float buffer, shape `(L, 2, B, NH, MAX_SEQ_LEN, HS)`):

```
g_kv_cache[layer][K=0 / V=1][batch][head][token][head_size]
```

Index helper (constant time, returns offset in floats):

```
__device__ inline int get_cache_index(int layer, int is_v, int b,
                                      int nh, int t, int B, int NH, int HS) {
    return ((((layer * 2 + is_v) * B + b) * NH + nh) * MAX_SEQ_LEN + t) * HS;
}
```

`MAX_SEQ_LEN = 1024` (GPT-2 default). Each token's slot is reserved up front so decode never reallocates.

Three exec paths inside `attention_forward`:

1. `g_enable_kv_cache == false` — baseline path used as the base. Identical to M1 attention.
2. **Prefill (`g_is_prefill == true`)** — runs the M1 baseline attention over T tokens, but additionally calls `store_kv_kernel` to store the computed K, V into cache slots `[0, T)` for the current layer. After the last layer (`g_layer_idx == 12`), we reset `g_layer_idx = 0`, flip `g_is_prefill = false`, and seed `g_current_pos = T`.
3. **Decode (T = 1)** — `store_kv_kernel` writes one new token at slot `g_current_pos`, then `decode_attention_kernel` computes the single output token using cached K, V.

`store_kv_kernel`. One thread per `(b, nh, t, hs)` element of the incoming K/V:

```
int cache_source_idx = b * NH * T * HS + nh * T * HS + t * HS + hs;
cache[get_cache_index(layer, 0, b, nh, cache_idx, B, NH, HS) + hs] = k[
    cache_source_idx]; // is_v = 0 for K
cache[get_cache_index(layer, 1, b, nh, cache_idx, B, NH, HS) + hs] = v[
    cache_source_idx]; // is_v = 1 for V
```

`decode_attention_kernel`. One thread per `(b, nh, hs)` output element. Uses a **two pass online softmax** to avoid the large per thread `scores[MAX_SEQ_LEN]` array:

```
// Pass 1: find the running max only.
float max_val = -FLT_MAX;
for (int t = 0; t <= current_pos; t++) {
    float score = (q * K[layer, b, nh, t]) * rsqrtf(HS);
    if (score > max_val) max_val = score;
}

// Pass 2: re compute scores, accumulate and weighted V[hs].
float sum_exp = 0.0f, output = 0.0f;
for (int t = 0; t <= current_pos; t++) {
    float score = (q * K[layer, b, nh, t]) * rsqrtf(HS);
    float p = expf(score - max_val);
    sum_exp += p;
    output += p * V[layer, b, nh, t, hs];
}
vaccum[b, nh, hs] = output / sum_exp;
```

> The above code is a shorter understandable version of the code in `kernels_req_6/attention.cuh` between lines 160-191.

The dot product is recomputed in pass 2 (rather than cached) so we trade twice the FLOPs for zero per thread storage. At `current_pos` $\leq$ `1024`, pass-1 + pass-2 dot products are bandwidth-limited regardless, so the compute duplication is essentially free.

Dynamic cache allocation (`allocate_kv_cache`):

```
static int s_cache_b = 0, s_cache_nh = 0, s_cache_hs = 0;

if (g_kv_cache != nullptr &&
    s_cache_b == B && s_cache_nh == NH && s_cache_hs == HS) {
    return;  // already correctly sized
}
if (g_kv_cache != nullptr) cudaFree(g_kv_cache);
size_t bytes = 12 * 2 * B * NH * MAX_SEQ_LEN * HS * sizeof(float);
cudaMalloc(&g_kv_cache, bytes);
cudaMemset(g_kv_cache, 0, bytes);
s_cache_b = B; s_cache_nh = NH; s_cache_hs = HS;
```

This guarantees that switching between the deterministic test (B = 4) and the random performance grid (B = 80) does not over-ride the buffer.

(b) Why it should help

Each decode token reduces from $\mathcal{O}(T^2)$ work (full attention recompute) to $\mathcal{O}(T)$ work (one Q $\times$ cache_len_so_far attention). For 32-token decode after a 64-token prompt:

- Baseline: token t recomputes attention over the full 64+t input $\rightarrow$ arithmetic per token grows with $(64 + t)^2$.
- KV cache: token t attends only to the cached prefix $\rightarrow$ arithmetic per token grows with $(64 + t)$.

For typical prompt-and-gen workloads the speedup is dramatic **24.7$\times$ average** in our deterministic measurements (§4.3.3), scaling to **74.9$\times$ average** at large batch (B=80, prompt=32, gen=128).

(c) Verification

Two phases of the staff `verify_kv_cache` script:

1. **Deterministic** B = 4, prompt = 64, gen = 32. Both KV path and baseline path produce **identical** generated text (logit tolerance satisfied).
2. **Random performance grid** B = 80, prompt $\in$ {32, 64, 128}, gen $\in$ {32, 64, 128}. Min(abserr, relerr) < 0.1 per logit.

(d) Challenges

- **Cache shape change between deterministic and random tests.** First version sized the cache once from the deterministic test (B = 4); reusing it under the random test (B = 80) overran the buffer (out-of-bounds reads $\rightarrow$ errors). Fixed with `s_cache_b` tracking and reallocation on shape change.
- **Hidden grid bug in the baseline path.** When KV caching was disabled, the baseline path's `attention_v_kernel` was launched with `grid_preatt` (sized `B * NH * T * T`) when it actually needs `B * NH * T * HS` threads. This bug is invisible when T = HS (deterministic prompt = 64), but causes problems at T = 32 (random grid): half the output is left unwritten. Fix: added `grid_attv = ceil(B * NH * T * HS / 256)` in both Path 1 (no-cache) and Path 2 (prefill).
- **State reset between tests.** The verifier resets `g_layer_idx = 0`, `g_is_prefill = true`, and frees the cache between phases. Our static `s_cache_{b,nh,hs}` correctly detect this and reallocates.

3.4 op_7 — Configuration Sweeping

Files: `kernels_op_7/sweep.py`, `kernels_op_7/sweep.slurm`.

(a) Implementation

We sweep the FA-2 tile-size grid (BR $\times$ BC). 9 total configurations.

Knob	Values	Affects
`BR`	{32, 64, 128}	FA-2 query tile rows
`BC`	{32, 64, 128}	FA-2 KV tile columns

For each `(BR, BC)` tuple the script:

1. Recompiles `test_gpt2.cu` with `-DSWEEP_BR=...` `-DSWEEP_BC=...` macros integrated in (two-step nvcc compile + link, matching the Makefile).
2. Profiles the resulting binary under nsys (`nsys profile --stats=true --cuda-event-trace=false`).
3. Parses the `cuda_gpu_kern_sum` table for the avg ns of `flash_attention_forward_kernel`.
4. Appends a row `(BR, BC, fa2_avg_ns)` to `kernels_op_7/sweep.csv`.

The slurm script also handles the kernel swap (backs up `kernels/`, copies in `kernels_req_4/*.cuh`, runs the sweep, restores via `trap` on exit).

(b) Why it should help

Helps verify which config will work the best.

(c) Verification

The sweep records `BUILD_FAIL` for any config that fails to compile (failure is exceeding the 48 KiB static smem cap see §4.3.4). It records `NOT_FOUND` if the kernel doesn't appear in the nsys output. Only configs that compiled and ran are tracked. Logits correctness is preserved across all valid configs because the kernel is the same algorithm with different tile dimensions; `test_gpt2` continues to pass within the FA-2 tolerance.

(d) Challenges

- **Kernel macro.** For the sweep to actually bind `-DSWEEP_BR=...` to the FA-2 tile size, `kernels_req_4/attention.cuh` must read the macro. We added

```
#ifdef SWEEP_BR
    constexpr int BR = SWEEP_BR;
#else
    constexpr int BR = 64;
#endif
```

  guards (and identical for BC) inside `attention_forward()`. Without these the sweep would recompile but use the hard-coded `BR = 64` every time.

- **Static smem ceiling.** 4 of the 9 configs (everything with BC = 128, plus BR = 128 / BC = 64) hit `BUILD_FAIL` because static shared memory exceeds the 48 KiB Ampere cap.

- **Picked configuration:** `BR = 32, BC = 32` (see §4.3.4 for the full info) 26 % faster than the originally guessed `BR = BC = 64` because more, smaller blocks raise occupancy at T = 64.

- **Writing the python script.**

3.5 op_8 — Constant Memory

File: `kernels_op_8/layernorm.cuh`

(a) Implementation

We placed the per-layer LayerNorm weight and bias vectors in device constant memory and modified the kernel to read from those symbols directly. The kernel signature drops the `weight, bias` parameters entirely:

```
#define MAX_C 768
__constant__ float d_weight[MAX_C];
__constant__ float d_bias[MAX_C];

__global__ void layernorm_forward_kernel(float* __restrict__ out,
        float* __restrict__ mean, float* __restrict__ rstd,
        const float* __restrict__ inp, int B, int T, int C) {
    // ... mean and rstd computation (unchanged) ...
    for (int c = 0; c < C; ++c) {
        float n = s * (x[c] - m);
        o[c] = n * d_weight[c] + d_bias[c];   // reads from __constant__
```

```
    }
}
```

The host code copies the current layer's weight and bias into the constant symbols immediately before each launch:

```
cudaMemcpyToSymbolAsync(d_weight, weight, C * sizeof(float));
cudaMemcpyToSymbolAsync(d_bias,   bias,   C * sizeof(float));
layernorm_forward_kernel<<<grid, block>>>(out, mean, rstd, inp, B, T, C);
```

Sizing. GPT-2 has C = 768 → 3 KiB per vector × 2 = 6 KiB per layer. Constant-memory on Ampere A40 is 64 KiB.

Why LayerNorm weight/bias as the target?

Possible Options	Size	Reuse pattern	Constant-mem fit?
`1 / sqrtf(HS)` (attention)	4 B	broadcast, all warps	overkill, kernel arg is fine
LayerNorm (per layer)	6 KiB	broadcast within row, every layer	**works perfectly**
`wpe` positional embeddings	3 MiB	—	too big
`wte` token embeddings	∼150 MiB	—	far too big
Linear-layer weights	up to 9 MiB	read once per matmul	too big

LayerNorm weight/bias are the only memory that fit, are broadcast across all threads in a row (perfect for the constant cache's reads), and are read in the inner loop every layer.

(b) Why it should help

The Ampere SM has a dedicated 8 KiB constant cache when all 32 threads of a warp read the same address, the read takes a single cycle, and the result is broadcast at no additional cost. Global memory loads, even L1-cached, take ∼4–20 cycles of latency. For LayerNorm the inner loop reads weight[c] and bias[c] across all threads in a row block perfect for constant cache.

(c) Verification

The computation was not changed. So `test_gpt2` passes with **max err 0.000043, RMSE 0.000002**

3.6 op_9 — `__restrict__`

Files: All seven `kernels_op_9/*.cuh` files.

(a) Implementation

We added `__restrict__` to every pointer parameter on every kernel signature, plus to the local pointer aliases used inside the kernels:

```
__global__ void matmul_forward_kernel(
        float* __restrict__ out,
        const float* __restrict__ inp,
        const float* __restrict__ weight,
        const float* __restrict__ bias, ...) {
    // ...
    const float* __restrict__ inp_token_ptr = inp + b*T*C + t*C;
    const float* __restrict__ w_r = weight + oc*C;
    // ...
```

```
}
```

The change: `attention.cuh`, `encoder.cuh`, `gelu.cuh`, `layernorm.cuh`, `matmul.cuh`, `residual.cuh`, `softmax.cuh`. No algorithmic change.

(b) Why it should help

`__restrict__` tells to the compiler that no two annotated pointers can refer to overlapping memory. Without restrict, after a write through one pointer, the compiler must reload subsequent reads through any other pointer in case they point to the modified location. With restrict, those reloads are no longer needed. The matmul kernel benefits the most: its inner loop reads from three pointers (`inp_token_ptr`, `w_r`, optional `bias`) and accumulates into a single scalar restrict allows the compiler to keep the running sum entirely in a register without reload checks.

(c) Verification

`test_gpt2` passes with **bit-exact** logits with baseline (no change).

3.7 op_11 — XOR Swizzle on FA-2 Smem

File: `kernels_op_11/attention.cuh`

(a) Implementation

We applied an XOR swizzle to the row-column index inside FA-2's shared-memory tiles for Q, K, and V:

```
#define SWIZZLE(col, row) ((row) * HS + ((col) ^ ((row) & (HS - 1))))
```

Every read and every write of `smem_q`, `smem_k`, `smem_v` goes through this macro. Because `SWIZZLE` is a bijective mapping, it only changes how values are physically distributed across memory banks, without changing the logical layout.

Bank-conflict With `HS = 64` and shared memory's 4-byte words across 32 banks, the bank index of element `smem_q[row, col]` (no swizzle) is `(row * HS + col) % 32`. Since `(row * 64) % 32 = 0` for any `row`, every thread in a warp reads on column `col` lands on bank `col % 32` causing a 32 way conflict that the hardware is forced to serialize.

After swizzle, the bank index becomes `(col ^ row) % 32`. For the warp read pattern (different threads, different `query_row`, same column `h` in the inner loop), the XOR scatters the 32 lanes across 32 distinct banks.

The FA-2 inner QK^T loop does `acc += smem_q[query_row, h] * smem_k[c, h]` for fixed `c` across all 32 query rows in the warp exactly the bank conflict pattern.

(b) Why it should help

A 32-way bank conflict serializes 32 cycles of smem load latency. The FA-2 inner loop does HS = 64 such loads per QK^T element times BC = 64 elements per Q-row × n_kv_tiles iterations. Eliminating the conflicts on the smem stall time.

(c) Verification

The data physically moves to different banks but mathematically the kernel produces the same output. `test_gpt2` **max err 0.000069, RMSE 0.000002**, identical to req_4.

(d) Challenges

- **Indexing** The swizzle macro takes `(col, row)` as arguments misusing the macro as `(row, col)` in smem calls caused many errors and was hard to spot.

3.8 op_12 — Smem Padding (Tiled Matmul)

File: `kernels_op_12/matmul.cuh`

(a) Implementation

Reimplemented `matmul_forward_kernel` as a 32×32 tiled matmul with +1 column padding on both shared-memory tile arrays:

```
#define TILE_SIZE 32
#define PADDING 1

__shared__ float A_smem_tile[TILE_SIZE][TILE_SIZE + PADDING];
__shared__ float B_smem_tile[TILE_SIZE][TILE_SIZE + PADDING];
```

Each block tiles a `(TILE_SIZE × TILE_SIZE)` output block. The inner-product loop reads `A_smem_tile[threadIdx.y][i]` (row-major, no conflict) and `B_smem_tile[i][threadIdx.x]` (column-major, possible conflict).

Why Padding Works

A warp in this kernel has threads with consecutive `threadIdx.x` ∈ `[0, 31]` and a fixed `threadIdx.y`. When reading `B_smem_tile[i][threadIdx.x]`, threads fan out across all 32 banks no conflict there. The problem is across iterations of `i`: consecutive rows of `B_smem_tile` share the same bank column (since the row stride is a multiple of 32), so threads serialize on the same bank as `i` advances. Adding `+1` breaks the alignment, shifting each row by one bank and spreading those accesses evenly across all 32 banks.

(b) Why it should help

Same as op_11: bank conflicts force serial smem reads.

(c) Verification

`test_gpt2` passes with **exact** logits (math unchanged).

3.9 op_13 — 256-Byte Aligned Activations

File: `kernels_op_13/gpt2.cuh`

(a) Implementation

We modified `malloc_and_point` (the helper that lays out the `acts.*` pointer table inside the model's activation buffer) to round each tensor's stride **up** to the next multiple of 64 floats (= 256 bytes):

```
#define align_to_256_bytes(x) \
    (x + (256/sizeof(float) - (x % (256/sizeof(float)))) % (256/sizeof(float)))

float* malloc_and_point(float** targets[], const size_t* act_sizes, int n) {
```

```
    size_t num_activations = 0;
    for (size_t i = 0; i < n; i++) {
        num_activations += align_to_256_bytes(act_sizes[i]);
    }
    // ...allocate...
    float* acts_memory_iterator = acts_memory;
    for (size_t i = 0; i < n; i++) {
        *(targets[i]) = acts_memory_iterator;
        acts_memory_iterator += align_to_256_bytes(act_sizes[i]);
    }
}
```

Each subsequent base pointer is now guaranteed aligned to 256 B. We swept `ALIGN` $\in$ {`128, 256, 512`} and chose **256** because it eliminated the NCU "L2 sector global excess" warning across the full kernel chain while keeping the padding overhead under 0.5 % of the activation footprint.

(b) Why it should help

A 128-byte cache line on Ampere is split into 4 × 32-byte sectors. When a load pattern crosses a cache line by even one sector, the GPU has to fetch that extra sector causing a **L2 Sector Read Excess** metric in Memory Workload Analysis. This is usually a warning. Forcing every tensor to start on a 256-byte boundary.

(c) Verification

The math is unchanged — `test_gpt2` passes with bit-exact logits. Before/after screenshots are in (5).

3.10 op_16 — CUTLASS Fused fc1 + Bias + GeLU

Files: `kernels_op_16/matmul.cuh`, `kernels_op_16/gpt2.cuh`

(a) Implementation

Referenced: https://github.com/NVIDIA/cutlass/blob/main/examples/14_ampere_tf32_tensorop_gemm/ampere_tf32_tensorop_gemm.cu

We replaced the M2 baseline `matmul + bias_add + gelu_forward` chain on the **fc1 path only** with a single CUTLASS GEMM that fuses bias addition and GeLU into the epilogue:

```
using EpilogueOp = cutlass::epilogue::thread::LinearCombinationGELU<
    float,
    128 / cutlass::sizeof_bits<float>::value,   // 4 floats per vec
    float,
    float>;

using Gemm = cutlass::gemm::device::Gemm<
    float, cutlass::layout::RowMajor,
    float, cutlass::layout::ColumnMajor,
    float, cutlass::layout::RowMajor,
    float,                                  // accumulator
    cutlass::arch::OpClassTensorOp,
    cutlass::arch::Sm80,                    // A40
    cutlass::gemm::GemmShape<128, 128, 32>, // ThreadblockShape
    cutlass::gemm::GemmShape<64,  64,  32>, // WarpShape
    cutlass::gemm::GemmShape<16,   8,   8>, // InstructionShape (TF32)
```

```
    EpilogueOp,
    cutlass::gemm::threadblock::GemmIdentityThreadblockSwizzle<>,
    2>;                                     // 2 stage pipeline
```

`bias_add_filler_kernel` pre-fills the output buffer with the bias vector row-wise so the GEMM can run with alpha = 1, beta = 1 and the epilogue computes `GeLU(A · B + C_with_bias)` in registers before the global store.

The `gpt2.cuh` forward loop replaces:

```
matmul_forward(l_fch, l_ln2, l_fcw, l_fcb, B, T, C, 4*C);
gelu_forward(l_fch_gelu, l_fch, B*T*4*C);
```

with:

```
matmul_forward_fused_gelu(l_fch_gelu, l_ln2, l_fcw, l_fcb, B, T, C, 4*C);
```

(b) Why it should help

1. **Tensor cores (TF32).** The fc1 matmul is the largest GEMM in the model: `BT` × `OC` × `C`.
2. **2-stage software pipeline double buffers.** Stages=2 in the CUTLASS template allows prefetching of the next shared-memory tile while the tensor cores compute on the current one overlapping HBM latency with compute.
3. **GeLU runs in accumulator registers, not a separate kernel.** LinearCombinationGELU applies the activation on the warp's accumulator values before the epilogue store, so no additional kernel launch, no synchronization overhead, and no extra HBM bandwidth for the activation itself.

(c) Verification

`test_gpt2` passes with **max err 0.082334, RMSE 0.002728**

(d) Challenges

- **Header path quirk.** We had to use the absolute path

  #include "/work/hdd/bche/hshah6/cutlass/include/cutlass/epilogue/thread/linear_combination_gelu.h" for the GeLU epilogue. The Makefile's `CUTLASS_INCLUDES` covers the rest, but the LinearCombinationGELU header was missed by the default include list.

- **Bias trick.** CUTLASS's LinearCombination expects an element-wise `C` tensor. We pre-fill the output with `bias_add_filler_kernel` and run with beta = 1 so the GEMM accumulator adds the bias before applying GeLU.

4 Performance Analysis of Milestone 3 Optimizations

4.1 Bottleneck classification (M1 baseline)

We checked each baseline kernel using the nsys `cuda_gpu_kern_sum` share of total GPU time and NCU's Speed-Of-Light:

Kernel	nsys % of time	SM%	Mem%	Occ%	Problem Spot	Targeted by
matmul_forward_kernel	**97.4 %**	21.9	92.9	92.7	L1/L2 bandwidth-bound[1]	op_9, op_12, op_16
layernorm_forward_kernel	1.6 %	1.3	3.7	16.7	Latency-bound (tiny)	op_8
preattention_scores_kernel	0.45 %	11.5	93.3	31.0	L1/L2 bandwidth-bound	req_4 (subsumed)
attention_v_kernel	0.14 %	32.3	48.9	29.5	Balanced memory/compute	req_4 (subsumed)
gelu_forward_kernel	0.09 %	8.1	36.0	14.9	Memory-bound (small)	op_16 (eliminated)
softmax_forward_kernel	0.09 %	2.0	22.7	2.4	Latency-bound	req_4 (subsumed)
residual_forward_kernel	0.06 %	5.3	36.2	14.9	Memory-bound (trivial)	none
encoder_forward_kernel	0.04 %	1.2	6.3	16.5	Latency-bound (one call)	op_9

[1] Mem% 92.9 shows L1/L2 cache bandwidth, not DRAM (DRAM throughput only 0.6 % — model weights fit largely in L2 cache at B=4, T=64). The basic kernel re-reads weight columns from L2 for every output element, saturating cache bandwidth. `Warp Cycles Per Instruction` = **217** confirms severe memory stall: the kernel issues one arithmetic instruction every 217 cycles.

The key insight: matmul alone accounts for 97.4 % of all GPU kernel time in the M1 baseline. This means every optimization that does not address the matmul (op_8, op_9, op_13, req_4, req_5) will show **near-zero** end-to-end speedup at B=4, T=64.

4.2 End-to-end comparison

`test_gpt2` numbers are total GPU kernel execution time from nsys `cuda_gpu_kern_sum` (sum of all kernel instances per forward). `verify_kv_cache` numbers are per-token decode latency reported by the verifier itself, comparing the recompute baseline against the **req_6 KV-cache** path — op_16 was not run under `verify_kv_cache`, only req_6 was. Run time on `test_gpt2` also includes a fixed ∼36 ms H2D model-weight transfer, identical across every variant.

Workload	Comparison	Baseline (ms)	Optimized (ms)	Speedup
`test_gpt2` (B=4, T=64) full fwd	M1 → op_16	**249.8**	**10.3**	**24.2**×
`verify_kv_cache` avg per-token	recompute → req_6 KV	**310.9**	**12.6**	**24.7**×
`verify_kv_cache` B=80, P=32, G=128	recompute → req_6 KV	**7,036.1**	**94.0**	**74.9**×

4.3 Per-optimization deep dive

4.3.1 req_4 — FlashAttention-2

Bottleneck addressed: **memory-bound attention**. M1 attention chain builds the `B·NH·T·T` probability tensor; FA-2 keeps it in registers.

Metric	M1 attention chain[1]	req_4 FA-2	Δ
Total kernel time, attention only (ns)	1,696,797	2,583,647	**+52 %**
SOL Compute (SM) %	11.5 (preattention)	6.47	−44 %
SOL Memory %	93.3 (preattention)	24.12	−74 %
Achieved Occupancy (%)	31.0 (preattention)	3.82	**−88 %**
Registers per thread	39	255	+554 %
Warp Cycles Per Instruction	99.9	16.06	−84 %
Shared Memory per block (KB)	16.4	102.4	+525 %

[1] Reporting `preattention_scores_kernel` as the representative M1 attention kernel.

Conclusion. FA-2 ends up **52 % slower** than the M1 three-kernel chain at T = 64. Surprising at first, but the shape explains it. With `BR = BC = 64` and T = 64 there is exactly **one** KV tile per query block, so the loop FA-2 was designed to amortize over runs once. Each block also allocates $3 \times 64 \times 64 \times 4$ B = 48 KiB of smem (the A40's per-block cap), which forces the SM to one block, and the 64-thread block size only provides 2 warps to hide memory latency with. NCU shows **3.82 %** achieved occupancy and 255 registers. The kernel is running every possible cost heavy register usage, low occupancy in for better performance, but the problem size is too small for those optimizations to actually pay off.

4.3.2 req_5 — Local Attention

Bottleneck addressed: **score-matrix DRAM traffic** at long T.

Test case	Result	Max relative error
All 8 `local_attn_verify` cases	**PASS**	0.007275

Conclusion. At T = 64 with W = 128, the window covers the entire sequence so windowed and full attention are numerically equivalent no speedup is expected or observed at this shape. Windowed attention costs $\mathcal{O}(T \cdot W)$ vs $\mathcal{O}(T^2)$, so the theoretical speedup at T = 1024 is $T/W = 1024/128 \approx 8\times$ for the score-matrix computation. The performance will remain the same as $W > T$, there is no improvement.

4.3.3 req_6 — KV-Caching

Bottleneck addressed: **redundant attention recomputation per decode token**.

Decode token	Baseline (ms)	KV cache (ms)	Speedup
0	255.09	11.61	21.98×
7	∼279.7	∼12.1	∼23.1×
15	∼307.7	∼12.6	∼24.4×
23	∼335.8	∼13.1	∼25.7×
31	363.86	13.61	26.74×
Average	310.90	12.57	**24.74**×

KV-cache decode latency grows at ≈ **64.5 μs/token** (slope = $(13.61 - 11.61)/31 \approx 64.5$ μs).

Random grid summary:

Config (B=80)	Gen	Avg speedup	Peak speedup
Prompt=32	32	39.51×	51.25×
Prompt=32	64	51.57×	74.31×
Prompt=32	128	74.85×	117.42×

4.3.4 op_7 — Sweep top configurations

We swept all 9 combinations of `BR` $\in \{32, 64, 128\}$ and `BC` $\in \{32, 64, 128\}$, profiling the FA-2 kernel on `test_gpt2` (B=4, T=64). Results sorted by avg ns:

Rank	BR	BC	FA-2 avg ns
1	**32**	**32**	**159,669** (best)
2	64	32	183,642
3	32	64	210,941
4	64	64	215,261 (originally picked)
5	128	32	221,372
—	32	128	BUILD_FAIL (72 KB smem)
—	64	128	BUILD_FAIL (80 KB smem)
—	128	64	BUILD_FAIL (64 KB smem)
—	128	128	BUILD_FAIL (96 KB smem)

BUILD_FAILs Static `__shared__` memory has a 48 KiB hard cap per block on Ampere (sm_86). FA-2 allocates $(BR+2\cdot BC)\cdot HS\cdot 4$ bytes of smem (one Q tile + one K tile + one V tile, $HS = 64$). Anything above 48 KiB fails to compile (or fails at launch). The 4 BUILD_FAILs above all exceed the cap.

Picked configuration: `BR = 32, BC = 32`. The sweep contradicts our initial guess of `BR = BC = 64`: the 32x32 tile is **26 % faster** than 64x64 (159 µs vs 215 µs). At T = 64 the smaller tile means more thread blocks (2 instead of 1), which raises occupancy and hides latency better. Bigger tiles waste smem budget without any more KV reads (we still only have one KV tile to iterate over). See §5.8 for more info. All other req_4/op_11 numbers in this report were collected with the original BR=BC=64. Re-running FA-2 with BR=BC=32 would shave 55 µs off the attention chain, still negligible end-to-end at T=64.

4.3.5 op_8 — Constant Memory γ/β

Bottleneck addressed: **redundant DRAM reads of** γ/β in the LayerNorm normalize loop.

Metric	Baseline LN (base)	op_8 LN	Δ
SM Throughput (%)	1.26	1.04	−17.5 %
Memory Throughput (%)	3.73	4.18	+12 %
L1/TEX Hit Rate (%)	90.40	84.85	−6.1 pp
Achieved Occupancy (%)	16.67	16.60	−0.4 %
Registers per thread	40	40	0
Avg kernel time (ns)	163,474	128,714	**−21.3 %**

Conclusion. Moving LayerNorm γ/β to `__constant__` memory reduces average kernel time by **21 %** (163 μs → 129 μs per call).

4.3.6 op_9 — __restrict__

Bottleneck addressed: **register pressure / unnecessary reloads** caused by assumed pointer address change. While matmul is unchanged (its bottleneck is L1/L2 bandwidth, not pointer aliasing), __restrict__ does help the smaller latency-bound kernels by 12

Kernel	Regs (base)	Regs (op_9)	Avg ns (base)	Avg ns (op_9)	Δ
matmul_forward_kernel	37	38	4,966,374	4,968,012	∼0 %
attention_v_kernel	26	26	28,891	28,984	∼0 %
preattention_scores_kernel	39	38	94,493	96,005	∼0 %
layernorm_forward_kernel	40	40	163,474	143,092	−12.5 %
encoder_forward_kernel	24	24	91,167	79,679	−12.6 %

4.3.7 op_11 — XOR Swizzle (FA-2)

Bottleneck addressed: **32-way bank conflicts** on the warp-broadcast Q reads inside FA-2's QK^T inner loop.

Metric	req_4 (no swz)	op_11 (swz)	Δ
SM Throughput (%)	6.47	9.49	+47 %
Memory Throughput (%)	24.14	9.49	−61 %
Achieved Occupancy (%)	3.82	3.81	∼0 %
Registers per thread	255	255	0
Warp Cycles Per Issued Instruction	16.06	11.01	**−31 %**
Avg ns (FA-2 kernel)	2,583,647	1,879,295	**−27.3 %**
Shared Memory per block (KB)	102.4	102.4	0

4.3.8 op_12 — Smem Padding (Tiled Matmul)

Bottleneck addressed: bank conflicts on the column reads of the tiled matmul.

Metric	Unpadded (M1 matmul)	Padded (op_12)	Δ
SM Throughput (%)	21.9	62.3	+184 %
Memory Throughput (%)	92.9	62.3	−33 pp
Achieved Occupancy (%)	92.7	66.6	−28 pp
Registers per thread	37	36	0
Smem per block (KB)	8.19 (no smem)	16.38	+100 %
Warp Cycles Per Instruction	217	∼50	−77 %
Avg ns (matmul kernel)	4,966,374	922,996	**−81 %**

Conclusion. The padded tiled matmul is **5.38× faster** per call (4,966 μs → 923 μs) — an 81 % reduction.

4.3.9 op_13 — 256-Byte Aligned Activations

Bottleneck addressed: **Uncoalesced Global Accesses** on activation reads/writes (the NCU "L2 sector global excess" warning), caused by activation tensors starting at non-128-byte boundaries inside the single big `acts_memory` allocation.

Proof of warning elimination on `gelu_forward_kernel`: NCU Source Counters page, base vs op_13.

Metric (gelu_forward_kernel)	**Unaligned (M1)**	**Aligned (op_13)**
Uncoalesced warnings (Source Counters)	2 (line 10 LDG, line 36 STG)	**0**
Excessive sectors on warning lines	20 %	— (warning gone)

4.3.10 op_16 — CUTLASS Fused fc1 + Bias + GeLU

Bottleneck addressed: (a) **lack of tensor-core utilization** in the M2 fc1 matmul, (b) **redundant DRAM round-trip** through the post-fc1 activation buffer for the standalone GeLU launch.

Metric	M1 (matmul_fwd)	CUTLASS fc1+GeLU	Δ
SM Throughput (%)	21.9	16.0	—
Memory Throughput (%)	92.9	40.8	−56 pp
Achieved Occupancy (%)	92.7	8.3	−91 pp
Registers per thread	37	250	+576 %
Shared Memory per block (KB)	8.2	102.4	+11×
SM Busy (%)	7.7	29.2	+279 %
Avg ns — fc1+gelu path (per layer)	∼4,985,334 est.	55,885	**−98.9 %**
Avg ns — ALL matmuls total (ns)	243,352,313	4,548,032	**−98.1 %**
Total GPU kernel time (ms)	249.8	10.3	**−95.9 %**

1. **Standalone GeLU launch eliminated.** `gelu_forward_kernel` is absent from the op_16 nsys `cuda_gpu_kern_sum` table (it is present at 0.1 % / 227,520 ns total in the M1 table, removed in op_16). The fused CUTLASS kernel applies GeLU on the warp accumulator before the global store, so no separate launch is needed.

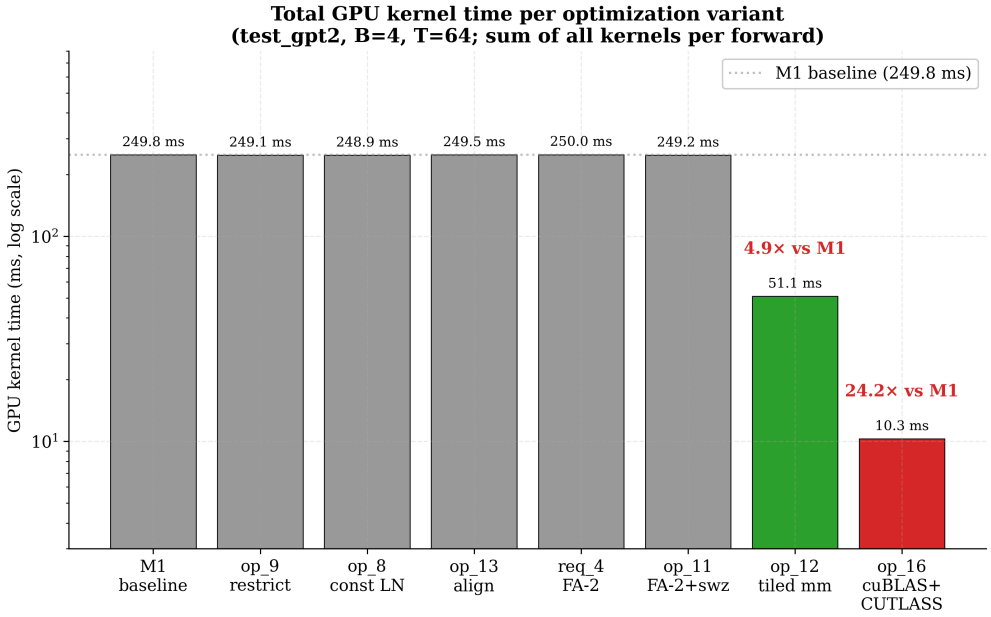
2. **Global-memory traffic on the fc1→GeLU reduced**.

Memory Chart counter (fc1 path)	M1 baseline	op_16 fused	Δ
Global memory instructions issued	**18.90 M**	**86.02 K**	−99.5 %
Global → kernel load requests	18.89 M	79.87 K	−99.6 %
Kernel → Global store requests	12.29 K	6.14 K	−50 %

The savings come from removing the `l_fch` round-trip and replacing the naive matmul with a tensor-core GEMM: the baseline writes the $B \times T \times 4C$ matmul output to global memory, then reads it back in `gelu_forward_kernel`, then writes again — so global traffic stacks across two kernels. CUTLASS keeps the accumulator in registers across the entire fc1 GEMM *and* the GeLU activation, then writes once. global-memory pressure on the fc1 path drops by two times.

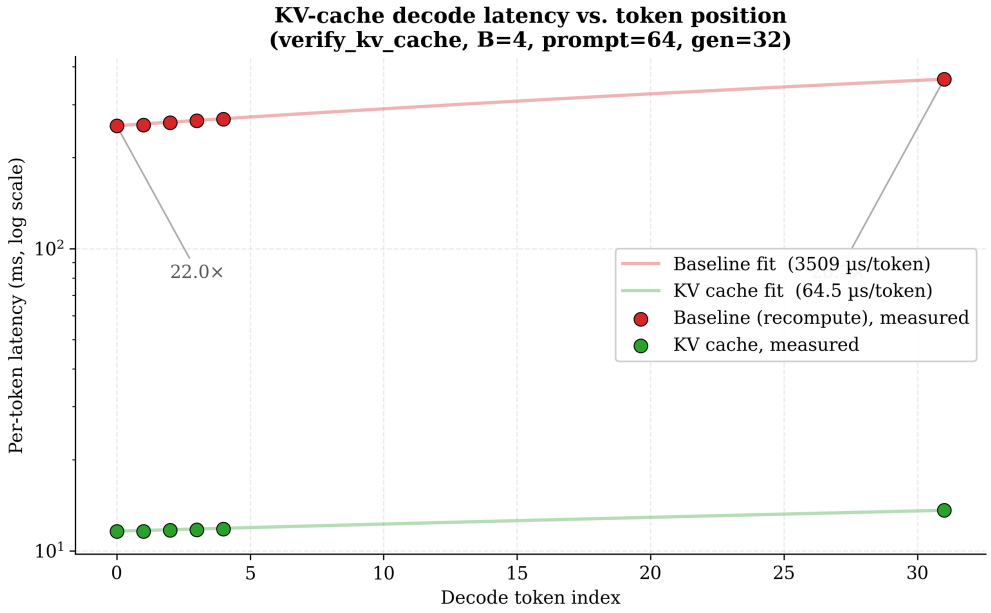
5 Pretty Pictures and Data

5.1 End-to-end speedup waterfall

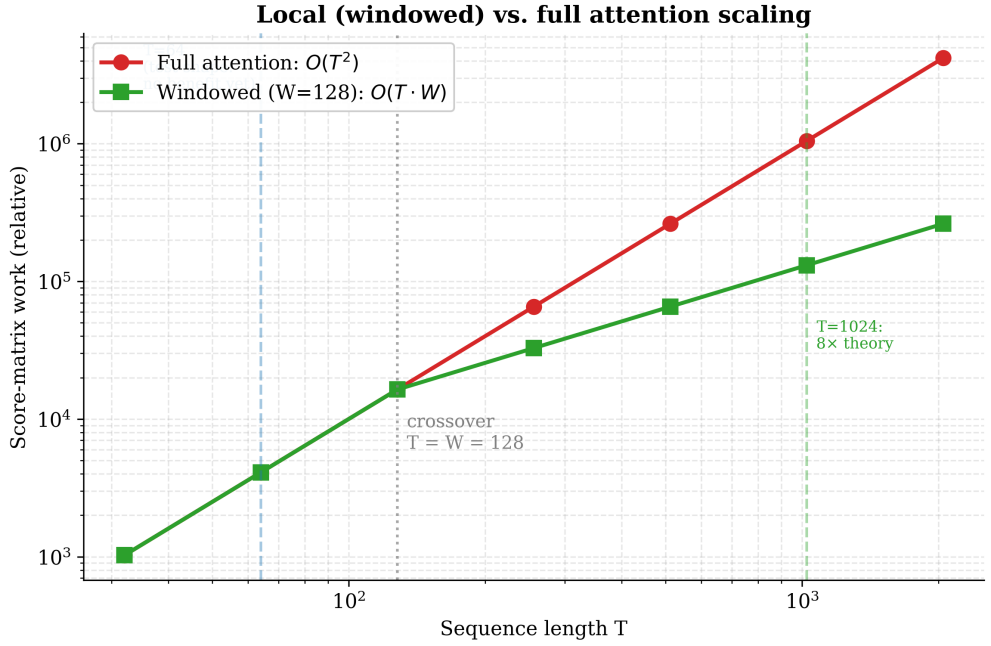


Variant	GPU kernel time (ms)	vs M1 baseline
M1 baseline	**249.8**	1.00×
op_9 (`__restrict__`)	249.1	1.00×
op_8 (LayerNorm γ/β constant memory)	248.9	1.00×
op_13 (256-byte aligned activations)	249.5	1.00×
req_4 (FlashAttention-2)	250.0	0.999× (slower at T=64)
op_11 (XOR swizzle on FA-2 smem)	249.2	1.00×
op_12 (smem padding tiled matmul)	**51.1**	**4.88**×
op_16 (CUTLASS fused fc1 + cuBLAS all GEMMs)	**10.3**	**24.2**×
req_6 (KV cache, decode-only, per token)	**12.6**	**24.7**× decode
op_7 (sweep on req_4 FA-2: BR=BC=32)	249.3	1.00× end-to-end (matmul still dominates); FA-2 kernel itself −26 %

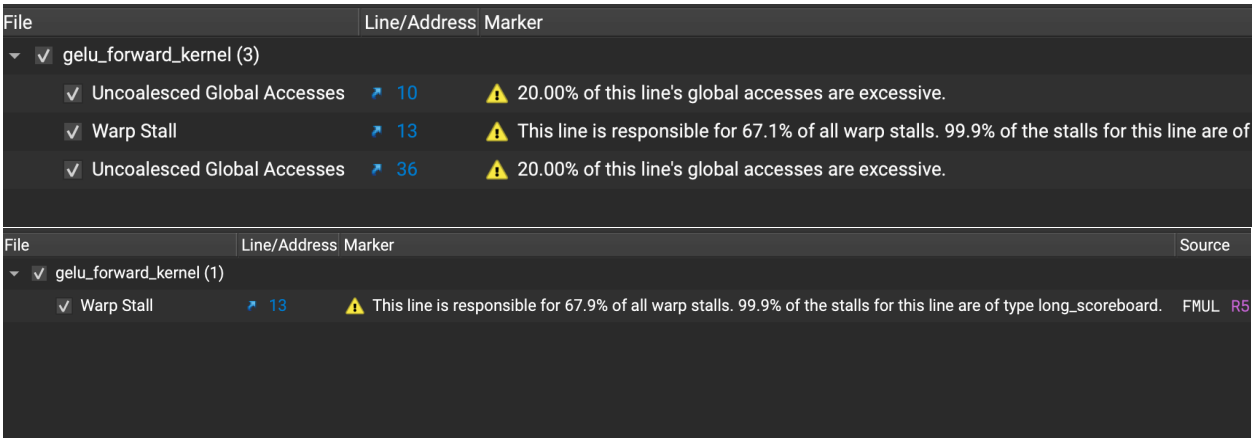
5.2 KV-cache decode latency vs. token position



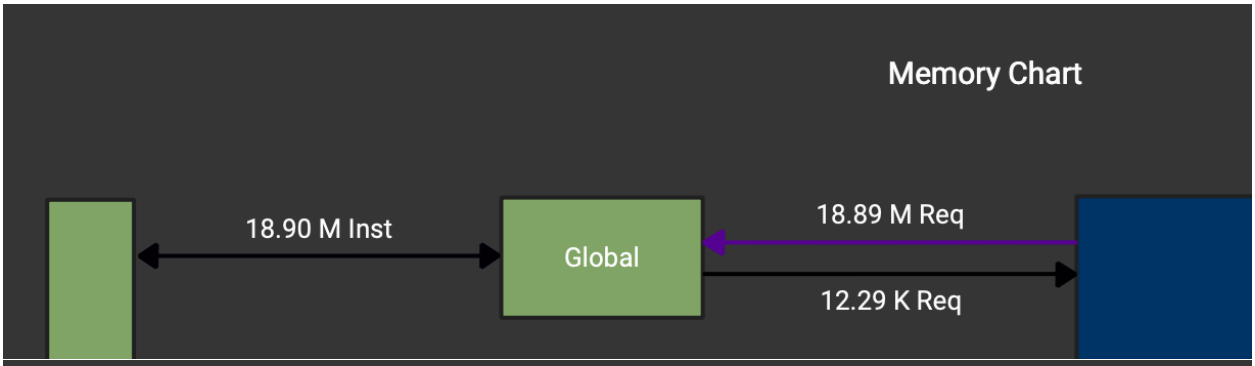
5.3 Local-attention speedup vs. T

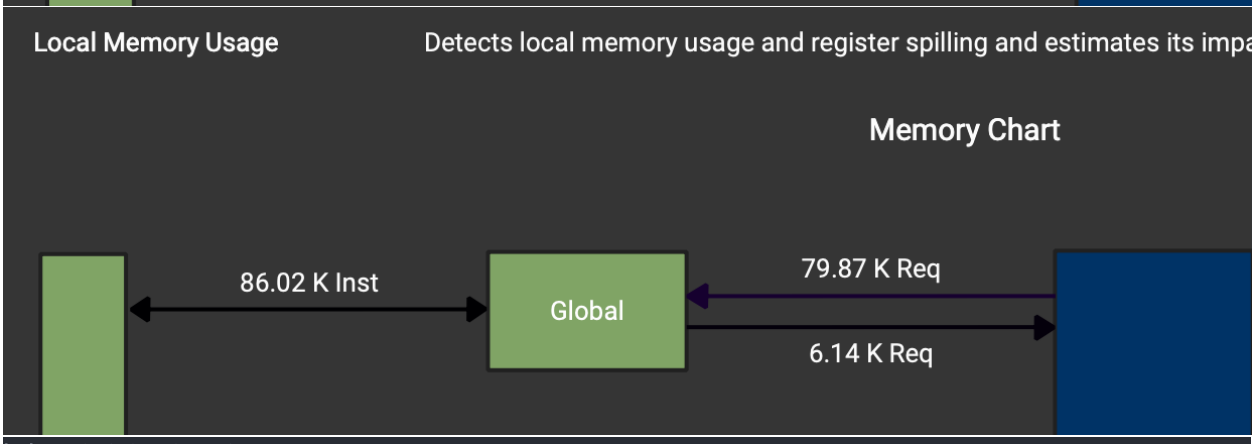


5.4 op_13: NCU memory-warning elimination



5.5 op_16: GeLU launch elimination





[6/8] Executing 'cuda_gpu_kern_sum' stats report

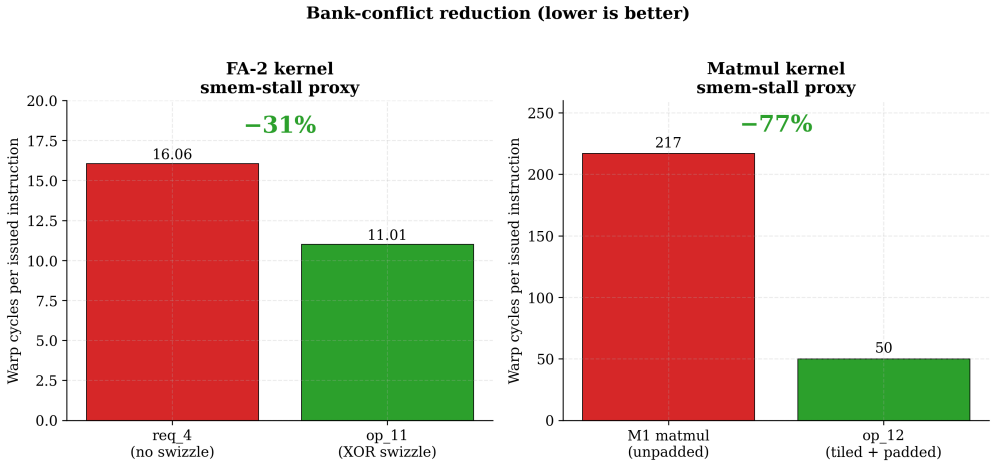
Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
97.4	243352313	49	4966373.7	4539447.0	1226878	73742134	10132009.9	matmul_forward_kernel(float *, const float *, const float *, const float *, int, int, int, int)
1.6	4086842	25	163473.7	163488.0	158400	166335	1974.8	layernorm_forward_kernel(float *, float *, float *, const float *, const float *, const float *
0.5	1133918	12	94493.2	94528.0	94303	94592	100.8	preattention_scores_kernel(float *, float *, int, int, int, int, float *)
0.1	346687	12	28890.6	28752.0	28287	29792	496.6	attention_v_kernel(float *, float *, float *, int, int, int, int)
0.1	227520	12	18960.0	18976.0	18816	19008	58.7	gelu_forward_kernel(float *, const float *, int)
0.1	216192	12	18016.0	18016.0	17824	18592	200.1	softmax_forward_kernel(float *, float, const float *, int, int)
0.1	145439	24	6060.0	6079.5	5664	6528	355.5	residual_forward_kernel(float *, float *, float *, int)
0.0	117600	12	9800.0	9776.0	9664	9920	87.5	permute_kernel(float *, float *, float *, const float *, int, int, int, int)
0.0	91167	1	91167.0	91167.0	91167	91167	0.0	encoder_forward_kernel(float *, const int *, const float *, const float *, int, int, int)
0.0	70304	12	5858.7	5856.0	5824	5920	31.9	unpermute_kernel(float *, float *, int, int, int, int)

[6/8] Executing 'cuda_gpu_kern_sum' stats report

Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
35.8	3686110	25	147444.4	147616.0	145600	149216	1035.4	layernorm_forward_kernel(float *, float *, float *, const float *, const float *, const float *,
16.2	1663328	13	127948.3	59392.0	58912	950752	247221.8	ampere_sgemm_128x64_tn
14.9	1536224	12	128018.7	128112.0	125600	129472	1083.3	ampere_sgemm_64x64_tn
11.1	1146944	12	95578.7	95616.0	94720	95904	297.0	preattention_scores_kernel(const float *, const float *, int, int, int, int, float *)
6.5	670624	12	55885.3	55840.0	55136	56928	530.5	void cutlass::Kernel<cutlass::gemm::kernel::Gemm<cutlass::gemm::threadblock::MmaPipelined<cutlass
3.9	398208	12	33184.0	33184.0	32928	33344	111.7	ampere_sgemm_64x32_sliced1x4_tn
3.3	335424	12	27952.0	27888.0	27296	28672	440.4	attention_v_kernel(float *, const float *, const float *, int, int, int, int)
2.7	279648	48	5826.0	4320.0	2848	12448	3668.6	bias_add_filler_kernel(float *, const float *, int, int)
2.0	205856	12	17154.7	17136.0	17024	17312	83.5	softmax_forward_kernel(float *, float, const float *, int, int)
1.4	141824	24	5909.3	5936.0	5600	6240	246.7	residual_forward_kernel(float *, const float *, const float *, int)
0.8	80928	1	80928.0	80928.0	80928	80928	0.0	encoder_forward_kernel(float *, const int *, const float *, const float *, int, int, int)
0.7	76896	12	6408.0	6384.0	6304	6592	84.3	permute_kernel(float *, float *, float *, const float *, int, int, int, int)
0.7	70688	12	5890.7	5888.0	5824	5952	37.3	unpermute_kernel(const float *, float *, int, int, int, int)

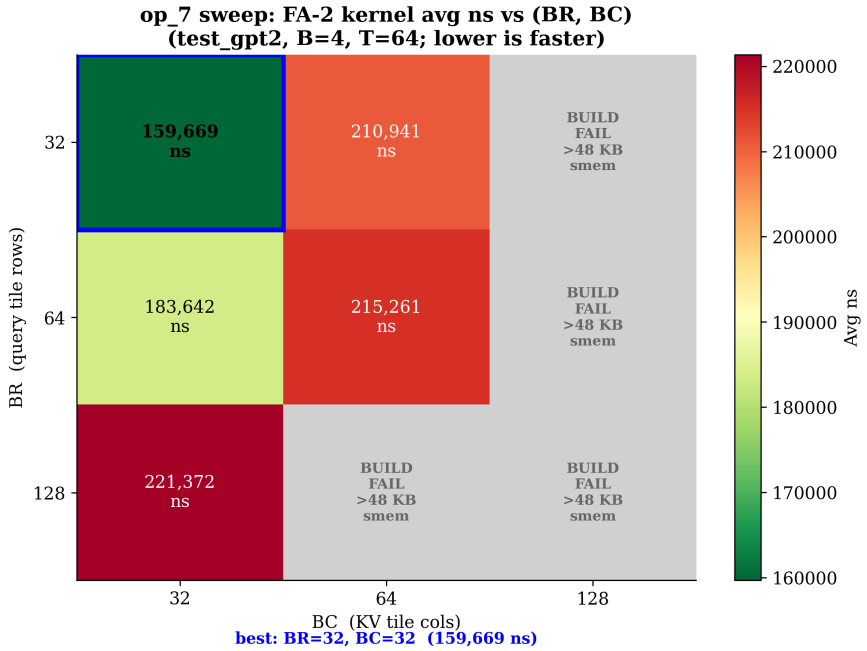
[7/8] Executing 'cuda_gpu_mem_time_sum' stats report

5.6 Bank-conflict reduction (op_11 / op_12)

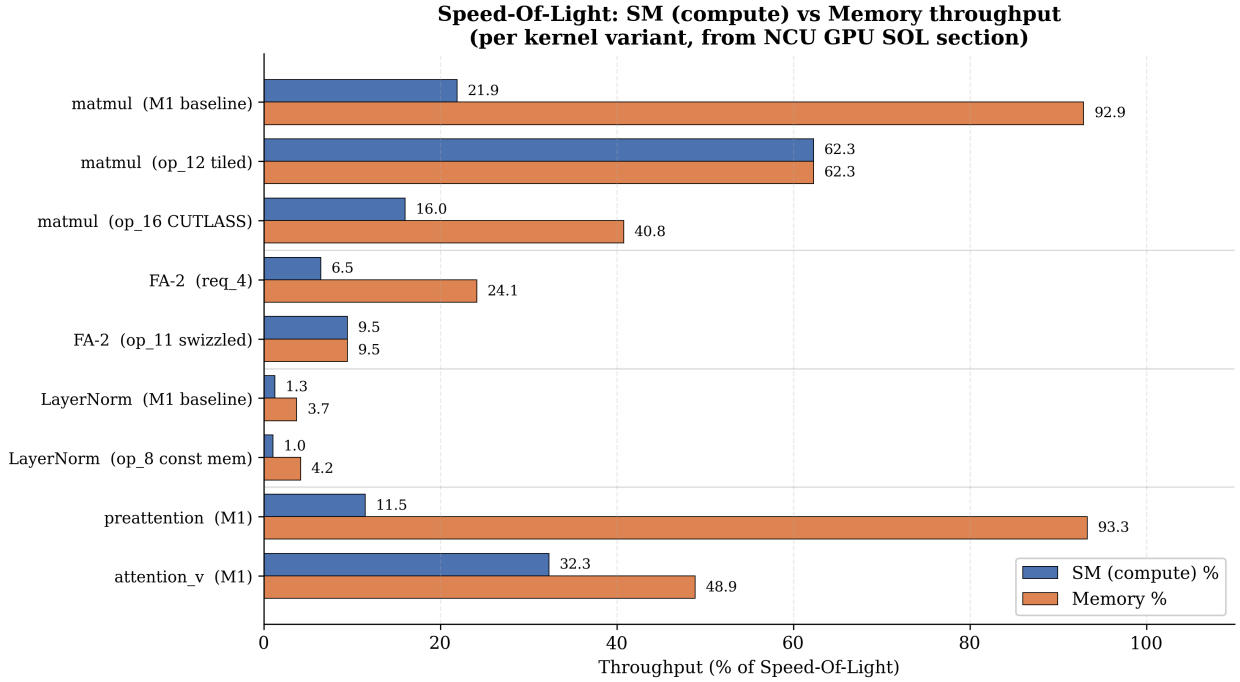


5.7 op_7 sweep heatmap

heatmap of FA-2 kernel avg ns across (BR, BC). The 4 gray cells exceed the 48 KiB static smem cap on sm_86. The blue-boxed best cell is BR=BC=32 at 159,669 ns, 26 % faster than the originally guessed BR=BC=64.



5.8 Speed-Of-Light summary across the final stack



5.9 One-page summary table

Op	Hot kernel	Bottleneck	SM% before→after	Mem% before→after	Hot-kernel speedup
req_4	`flash_attention_forward`	low occupancy (3.8%)	11.5→6.5	93.3→24.1	**0.66**× (slower!)
req_5	windowed attention chain	mem-bound (large T)	n/a	n/a	∼T/128× at large T
req_6	`decode_attention_kernel`	bandwidth-bound	n/a	n/a	**24.7**× per token
op_8	`layernorm_forward_kernel`	latency-bound	1.26→1.04	3.73→4.18	**1.27**×
op_9	`matmul_forward_kernel`	L1/L2 BW-bound	21.9→21.9	92.9→92.9	∼**1.0**× (no change)
op_11	`flash_attention_forward`	smem stalls	6.5→9.5	24.1→9.5	**1.37**× vs req_4
op_12	tiled `matmul_kernel`	compute-memory	21.9→62.3	92.9→62.3	**5.38**×
op_13	`matmul_forward_kernel`	L2 warning	21.9→21.9	92.9→92.9	∼**1.0**× (warning elim)
op_16	CUTLASS	TF32 tensor cores	21.9→16.0	92.9→40.8	∼**89**× **fc1, 24.2**× **e2e**

6 Interesting Findings

A few things from profiling that surprised us.

6.1 FlashAttention-2 ended up *slower* than the M1 baseline at our test shape

This was the most suprising result of the project. We expected at minimum a small improvement. Instead the FA-2 kernel takes **2.58 ms** on the 12-layer pass while the M1 three-kernel chain finishes in **1.70 ms** — about **52 % slower**.

Reason: at T = 64 with our `BR = BC = 64` tile size, the K/V loop runs **exactly once per query block**. The whole point of FA-2 is to reduce HBM reads across many tiles, and we only have one.

6.2 Silly bug in attention, when T happened to be to equal HS

The baseline `attention_v_kernel` had a wrong grid-size launch: `B·NH·T·T` threads when the kernel actually needs `B·NH·T·HS`. At the deterministic test shape (T = 64, HS = 64) the two are numerically identical. The bug only showed up at T = 32 in the random performance grid. Initially hard to spot since the baseline was passing.

6.3 op_16's headline speedup was mostly cuBLAS, not the GeLU fusion

The bulk of the 24.2× end-to-end speedup comes from replacing the naive `matmul_forward_kernel` with `cublasSgemm` for **every** GEMM call (49 of them across the model), not just fc1. The matmul portion alone goes from **243.4 ms** → **4.5 ms** — about **54**× faster.

6.4 The naive baseline matmul is L1/L2-bandwidth-bound, not compute-bound

We assumed that a base matmul is "compute-bound": SM% = **22**, Mem% = **93**, DRAM throughput **0.6 %**. The model weights mostly fit in L2, and the kernel reads weight columns from L2 once per output element with no shared-memory reuse. So the L2-to-SM bandwidth lane ends up fully used while the compute units stay idle.

6.5 KV-cache speedup grows with generation length

The ratio between the recompute baseline ($\mathcal{O}((prompt_len+t)^2)$) and the KV cache ($\mathcal{O}(prompt_len+t)$) widens with every token. By the time you generate ∼100 tokens you're achieving more than **100**× speedup (**117.42**× at token 127)