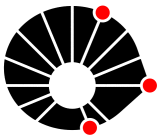

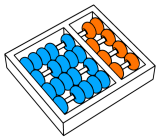

Exploração do Modelo de Programação CUDA Tile para GPUs NVIDIA

G. Braga *H. Hyviquel*

Relatório Técnico - IC-PFG-26-32
Projeto Final de Graduação
2026 - Julho

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Exploração do Modelo de Programação CUDA Tile para GPUs NVIDIA

Guilherme Braga* Hervé Yviquel*

Resumo

A busca por maior produtividade na programação de GPUs tem motivado o surgimento de modelos que elevam o nível de abstração sem sacrificar desempenho. Nesse contexto, a NVIDIA introduziu o *CUDA Tile*, que adota o *tile* como unidade lógica de execução, comunicação e sincronização entre *threads*, encapsulando padrões de cooperação tradicionalmente expressos de forma manual em CUDA. Este projeto investiga empiricamente esse modelo, comparando-o com o CUDA tradicional e com o Triton, um compilador JIT com *auto-tuning* que parte do mesmo diagnóstico, mas adota filosofia distinta. São descritos os três modelos, a definição da metodologia de comparação (princípio de equidade, protocolo de medição, tratamento estatístico e o modelo *Roofline* como ferramenta unificadora), a infraestrutura experimental desenvolvida (*bench harness*) e os resultados da comparação para diferentes padrões de kernels.

Palavras-chave: GPU, CUDA, CUDA Tile, Triton, Tensor Cores, programação paralela, modelo Roofline, FlashAttention.

Sumário

*Instituto de Computação, Universidade Estadual de Campinas, 13081-970 Campinas, SP.

1 Introdução

A computação de alto desempenho consolidou as Unidades de Processamento Gráfico (GPUs) como plataformas centrais para processamento paralelo massivo, com aplicações que vão de simulações científicas a aprendizado de máquina. A plataforma CUDA (*Compute Unified Device Architecture*) da NVIDIA [1] tornou essa capacidade acessível por meio de uma interface próxima de linguagens de alto nível, mas o modelo de programação tradicional ainda expõe o desenvolvedor a detalhes de baixo nível: sincronizações explícitas via `__syncthreads()`, gerenciamento manual de memória compartilhada e coordenação fina de acessos.

Para reduzir esse custo de produtividade sem abrir mão de desempenho, surgiram abstrações de mais alto nível. O **CUDA Tile** [2, 17] é a proposta mais recente da NVIDIA nessa direção: introduz a noção de *tile* como unidade lógica de execução, comunicação e sincronização entre *threads*, encapsulando padrões comuns de cooperação de forma declarativa. Em paralelo, fora do ecossistema NVIDIA, o **Triton** popularizou-se como um compilador JIT com *auto-tuning* nativo e uma DSL em Python voltada a engenheiros de *machine learning*, partindo do mesmo diagnóstico — "o CUDA explícito é trabalhoso demais" — mas chegando a uma solução arquiteturalmente diferente.

O problema de produtividade que motiva essas abstrações não é exclusivo da NVIDIA. No ecossistema AMD, a pilha ROCm oferece o **HIP** (*Heterogeneous-Compute Interface for Portability*), uma API em C++ deliberadamente próxima do CUDA e projetada para portar código entre as duas plataformas, ocupando, portanto, papel análogo ao do CUDA tradicional. Sobre essa base surgiram abstrações de mais alto nível, como o **FlyDSL** (*Flexible Layout Python DSL*), uma DSL em Python baseada em MLIR desenvolvida pela própria AMD que, ao contrário do Triton, expõe controle fino de baixo nível a usuários experientes, e o **TileLang**, um esforço independente de fabricante que adota a mesma ideia de *tile* como abstração central e já foi validado em GPUs AMD (por exemplo, reduzindo em uma ordem de grandeza o número de linhas de um *kernel* de *Flash Attention* sem perda de desempenho). Vale ressaltar, ainda, que o próprio Triton, por ser de código aberto e multiplataforma, executa tanto em GPUs NVIDIA quanto AMD, o que o distingue das soluções proprietárias comparadas neste trabalho.

1.1 Motivação e abordagem

Toda abstração de alto nível introduz potenciais *trade-offs* entre produtividade e controle fino sobre o *hardware*. Avaliar empiricamente esses *trade-offs* é o objetivo deste trabalho. Em vez de uma comparação binária (CUDA Tile *versus* CUDA tradicional), adota-se uma **comparação triangular**: CUDA tradicional, CUDA Tile e Triton, porque ela expõe o espaço de *trade-offs* de forma mais clara: CUDA Tile e Triton resolvem o mesmo problema (abstrair *tiling*) com filosofias distintas (extensão de C++/Python familiar ao ecossistema CUDA *versus* DSL Python com JIT e *auto-tuning*). Essa assimetria torna as conclusões mais generalizáveis: onde o CUDA Tile for mais expressivo que o baseline porém menos produtivo que o Triton, tem-se um argumento sobre completude de abstração; onde igualar o Triton em desempenho com código portável de ecossistema, tem-se um argumento sobre custo-benefício.

2 Fundamentação Teórica

2.1 Arquitetura de GPUs e o modelo CUDA tradicional

Uma GPU NVIDIA organiza a execução em *threads* agrupadas em blocos, que por sua vez executam sobre *Streaming Multiprocessors* (SMs) [6]. Dentro de um bloco, as *threads* cooperam por meio de memória compartilhada (rápida, on-chip) e de barreiras de sincronização. No modelo tradicional, essa cooperação é expressa explicitamente pelo programador: o *tiling* (particionamento dos dados em blocos que cabem na memória compartilhada) é manual [7, 8], as barreiras `__syncthreads()` são inseridas à mão e a redução intra-*warp* frequentemente recorre a primitivas como `__shfl_down_sync()` [1]. Esse controle fino é a fonte tanto do alto desempenho do CUDA quanto de sua baixa produtividade e propensão a erros de sincronização.

2.2 O modelo CUDA Tile (cuTile)

O CUDA Tile baseia-se na especificação e nas ferramentas da *Tile IR* (*Intermediate Representation*) [15], das quais o **cuTile** é a interface de programação. Um ponto essencial para o desenho experimental é que a implementação *user-facing* atualmente disponível é em **Python** (`pip install cuda-tile`); o suporte em C++ está previsto para o futuro. Trata-se de ferramenta recente e em evolução rápida: a versão 1.0.0 foi publicada em dezembro de 2025 e a 1.2.0 em março de 2026, o que recomenda revisitar a documentação oficial periodicamente ao longo do projeto.

O cuTile eleva o nível de abstração ao tratar operações coletivas como primitivas de primeira classe. Em particular, a redução está disponível diretamente como `ct.sum`, `ct.max`, `ct.min`, `ct.prod` e, para operadores customizados, `ct.reduce` [2], dispensando a árvore manual em memória compartilhada e as barreiras explícitas. O modelo mira especificamente os *Tensor Cores*, de modo que ganhos de desempenho podem se manifestar sobretudo em precisão reduzida (fp16/bf16).

Modelo de compilação e abstração de memória. A distinção arquitetural central em relação ao CUDA tradicional está na *preservação* da semântica de *tile* ao longo de toda a pilha de compilação. A *Tile IR* é uma ISA virtual orientada a *tiles* e tensores [15], e não a *threads* SIMT, de modo que os blocos lógicos permanecem entidades de primeira classe até próximo do *driver*, cabendo à *runtime*/compilador da CUDA mapeá-los aos *Tensor Cores* e à hierarquia de memória. Trata-se de um deslocamento do gerenciamento explícito de *threads* para abstrações baseadas em *tile*, no qual decisões de escalonamento e mapeamento ao *hardware* são delegadas ao compilador. No plano de programação, o cuTile adota um modelo *array-centric*: os acessos são expressos por índice e *shape* (por exemplo, `ct.load(a, index=(pid,), shape=(tile_size,))`), ocultando a aritmética de ponteiros. Como consequência dessa abstração, o código de *tile* é projetado para ser compatível com arquiteturas futuras (*forward-compatibility*).

Requisitos de hardware. O cuTile Python requer GPU com *compute capability* 8.x, 9.x, 10.x, 11.x ou 12.x e *driver* r580 ou superior, em Linux x86_64/aarch64 ou Windows x86_64, com Python 3.10–3.14. A inclusão da geração 8.x (Ada Lovelace/Ampere) na lista de requisitos é relevante: viabiliza a execução do cuTile em GPUs amplamente disponíveis, sem exigir necessariamente *hardware* de geração Blackwell.

2.3 Triton

O Triton é um compilador JIT de código aberto com *auto-tuning* nativo, exposto como uma DSL embarcada em Python. O programador descreve o *kernel* em termos de blocos (*tiles*) de dados, e o compilador encarrega-se de decisões de baixo nível como tamanho de bloco, número de *warps* e *pipelining*, frequentemente exploradas automaticamente pelo decorador `@triton.autotune`.

Pilha de compilação e portabilidade. Diferentemente do cuTile, o *frontend* do Triton é traduzido por um compilador próprio (Triton IR, baseado em MLIR $\rightarrow$ LLVM $\rightarrow$ PTX) [3], historicamente produzindo código SIMT em nível de *thread*: a semântica de *tile* existe na descrição do *kernel*, mas é "achatada" para o modelo de *threads* durante a geração de PTX. O modelo de memória é *pointer-centric*, ou seja, o programador constrói *block pointers* e manipula *offsets* explicitamente, em contraste com o modelo *array-centric* do cuTile. Por ser de código aberto e

multiplataforma, o Triton executa tanto em GPUs NVIDIA quanto AMD (via *backend* ROCm), característica ausente nas alternativas proprietárias aqui comparadas.

Convergência com a *Tile IR* e implicação para este estudo. Existe ainda um *backend* Triton-para-Tile-IR[18], que aproxima as duas pilhas ao permitir que o Triton emita CUDA Tile IR em vez de PTX, preservando a semântica de *tile* e o suporte nativo aos *Tensor Cores*. Esse caminho, entretanto, exige CUDA $\geq$ 13.1 e GPUs de geração Blackwell, não sendo aplicável ao *hardware* deste trabalho (Ada Lovelace, *compute capability* 8.9). Duas consequências decorrem daí. Primeiro, o *backend* nativo padrão do Triton (PTX/SIMT) é, na prática, a única via disponível na plataforma empregada, o que também é metodologicamente preferível, pois evita medir predominantemente o *frontend*. Segundo, e mais importante para a validade do estudo, o Triton avaliado aqui percorre seu caminho clássico (PTX/SIMT) enquanto o cuTile percorre a *Tile IR*: a comparação contrasta, portanto, *pilhas de compilação genuinamente distintas*, e não o mesmo *backend* acessado por dois *frontends*. Por serem fatores de confusão clássicos em *benchmarks*, a versão e o *backend* empregados são documentados explicitamente.

A Tabela 1 sintetiza os principais contrastes arquiteturais e funcionais entre o Triton e o cuTile discutidos nesta seção.

Tabela 1: Comparação arquitetural e funcional entre Triton e NVIDIA cuTile.

Critério	**Triton**	**cuTile**
Origem e governança	Código aberto (OpenAI, 2021); múltiplos fabricantes	Proprietário NVIDIA (CUDA 13.1, 2025); exclusivo NVIDIA
Modelo de programação	DSL em Python baseada em *tiles*; abstrai o gerenciamento de *threads*	DSL em Python baseada em *tiles*; abstrai o gerenciamento de *threads*
Abstração de memória	*Pointer-centric*: *block pointers* e aritmética de *offsets* explícita	*Array-centric*: `load`/`store` por índice e *shape*
IR e pilha de compilação	Triton IR (MLIR) $\rightarrow$ LLVM $\rightarrow$ PTX; *lowering* para SIMT em nível de *thread*	CUDA Tile IR (ISA virtual orientada a *tile*); semântica de *tile* preservada até a *runtime*
Mapeamento aos Tensor Cores	Compilador próprio gera PTX/SASS	*Runtime*/compilador CUDA mapeia *tiles* ao hardware
Estratégia de portabilidade	Entre fabricantes (NVIDIA, AMD)	Entre gerações NVIDIA (*forward-compatibility* para Blackwell)
Suporte de hardware	Amplo (via *backend* PTX)	Ampere, Ada e Blackwell (CC 8.x, 10.x, 11.x e 12.x)
Backend no hardware do estudo (RTX 4060, Ada, CC 8.9)	Caminho clássico PTX/SIMT	Caminho CUDA Tile IR
Maturidade e ecossistema	Madura; adoção ampla (ex.: PyTorch *Inductor*)	Recente (dez. de 2025)

Nota: o *backend Triton-to-TileIR* (2026) permite ao Triton emitir CUDA Tile IR em vez de PTX, aproximando as duas pilhas; contudo, exige CUDA $\geq$ 13.1 e GPU Blackwell, não se aplicando ao hardware deste estudo (Ada, CC 8.9).

Fonte: Elaborado pelo autor.

2.4 O modelo Roofline

O modelo *Roofline* [5] é a principal ferramenta de interpretação dos resultados. Ele posiciona cada *kernel* em um gráfico de desempenho (GFLOPS) *versus* intensidade aritmética (FLOP/byte), classificando-o como *memory-bound* ou *compute-bound* e revelando o quão perto cada implementação chega do teto atingível. Sua importância para este trabalho é metodológica: um *kernel* em CUDA Tile que atinge praticamente o mesmo ponto do *roofline* que o baseline, porém com metade das linhas de código e nenhuma sincronização explícita, constitui um argumento muito mais forte (“a

abstração não custou desempenho") do que a leitura isolada de uma leve diferença de tempo.

3 Metodologia de Comparação

3.1 Plataformas comparadas e seus papéis

A comparação envolve (a princípio e quando aplicáveis) quatro implementações de cada padrão, com papéis bem definidos (Tabela 2). Um ponto metodológico importante é a distinção entre *baseline* e *teto*: chamadas de bibliotecas *vendor-tuned* (cuBLAS, cuDNN) [13, 14] representam um teto de desempenho prático, e não um baseline escrito à mão. Rotulá-las como "CUDA tradicional" inflaria a régua e tornaria a comparação injusta.

Tabela 2: Plataformas avaliadas e seus papéis na comparação.

Plataforma	**Papel**	**Forma de implementação**
CUDA tradicional	*baseline*	*Kernel* CUDA C++ literal (`__global__`, `__syncthreads()`, memória compartilhada manual) escrito como *string* e compilado via NVRTC [4] (`cp.RawKernel`).
CUDA Tile (cuTile)	objeto de estudo	Primitivas de *tile* em Python, sem contorná-las.
Triton	comparador	DSL Python com `@triton.autotune` onde aplicável, *backend* nativo.
cuBLAS/cuDNN	teto *vendor*	Chamadas de alto nível via CuPy, usadas apenas como ponto de referência no *roofline*.

3.2 Princípio de equidade

O risco central de qualquer *benchmark* comparativo é a *assimetria de otimização* — comparar uma implementação polida de um modelo com uma implementação ingênua de outro. Para neutralizá-lo, cada padrão é implementado segundo a *melhor expressão natural* de cada modelo, e não a otimização absoluta: o baseline usa *tiling* manual correto com `__syncthreads()`; o cuTile usa as primitivas de *tile* sem contorná-las; o Triton usa `@triton.autotune` no *backend* nativo.

3.3 Camada comum de memória: CuPy

Adota-se o CuPy como camada comum de *arrays* e memória para os três modelos. Essa escolha não é improvisada: o próprio *quickstart* do cuTile usa CuPy como camada de *arrays* (os tensores de entrada/saída são *arrays* CuPy e o *launch* usa o *stream* corrente do CuPy). Padronizar tudo em CuPy torna o ambiente mais consistente e justo. É essencial, porém, distinguir seus dois modos de uso: (i) a API de alto nível (`cp.matmul`, `cp.sum`) despacha para bibliotecas *vendor* e portanto serve apenas como teto; (ii) o `cp.RawKernel`/`cp.RawModule` permite escrever o *kernel* CUDA C++ literal, gerando PTX/SASS idêntico ao do CUDA puro, o que é a via correta para o baseline "tradicional".

3.4 Protocolo de medição

Cronometra-se *somente* a execução do *kernel*, por meio de *CUDA events*, com a *closure* de *launch* fechada sobre *buffers* já alocados: nenhuma alocação ou cópia entra no laço cronometrado. Isso neutraliza o *overhead* de *launch* em Python, que para *kernels* não-triviais é desprezível e, de

todo modo, fica fora da janela de medição. Um *warmup* de pelo menos 10 iterações descartadas é duplamente necessário: estabiliza *clocks* e *caches* e dispara a compilação JIT do Triton e do cuTile, que de outro modo contaminaria a primeira medição.

3.5 Tratamento estatístico

Coletam-se no mínimo 50 repetições por configuração, reportando média, desvio padrão (com `ddof=1`), mínimo e mediana. Reporta-se a distribuição inteira, não apenas o mínimo: a variância de *launch* é uma característica do modelo, não um ruído a descartar. Cada padrão é avaliado em pelo menos três escalas de problema (*small*, *medium*, *large*) para capturar comportamento de *scaling* e amortização de *overhead*: *kernels* pequenos expõem custo de abstração; *kernels* grandes expõem eficiência de execução.

3.6 Validação numérica antes de cronometrar

Cada saída é conferida contra uma referência (NumPy/float64) *antes* de qualquer medição de tempo. Sem essa etapa, corre-se o risco de comparar a velocidade de *kernels* que produzem resultados diferentes. Tolerâncias relaxadas são corretas para fp16/bf16 e para a reordenação de somas inerente a reduções paralelas.

3.7 Métricas

As métricas dividem-se em três camadas, descritas na Tabela 3. As métricas *qualitativas* merecem destaque: a contagem de primitivas de sincronização explícita é, talvez, o indicador mais direto do nível de abstração de cada modelo. Para garantir objetividade, LOC e primitivas são contadas excluindo comentários e *includes*; o tempo de implementação é cronometrado do início da codificação até o primeiro teste que passe, registrado ao longo do projeto (não retrospectivamente); e a legibilidade segue uma rubrica de 1 a 5 com critérios definidos *antes* de implementar.

Tabela 3: Camadas de métricas coletadas pela infraestrutura experimental.

Camada	Métricas
Derivadas	GFLOPS efetivos, largura de banda efetiva e intensidade aritmética, a partir do tempo medido e do modelo de tráfego (FLOPs e *bytes* movidos).
Perfilagem (Nsight Compute)	Ocupação atingida, taxas de acerto L1/L2, pressão de registradores e *bytes* reais de DRAM (`dram__bytes.sum`).
Qualitativas	Linhas de código (LOC) e número de primitivas de sincronização explícita; tempo de implementação; rubrica de legibilidade.

4 Infraestrutura Experimental (*Bench Harness*)

A espinha dorsal da comparação é um *bench harness* reutilizável que implementa o protocolo da Seção 3. Cada padrão é descrito uma vez com três implementações (baseline, cuTile, Triton), a definição de FLOPs e *bytes* movidos, e a referência de validação. O restante do fluxo (*warmup*, *timing*, validação, métricas derivadas e classificação no *roofline*) aplica-se sem alterações.

Os componentes principais do *harness* são:

- `time_kernel`: cronometragem via *CUDA events* com *warmup* e múltiplas iterações, retornando média, desvio, mínimo e mediana.

- `attach_derived`: converte tempo em GFLOPS efetivos, largura de banda efetiva e intensidade aritmética.
- `attach_roofline`: classifica o *kernel* como *memory-* ou *compute-bound* e calcula a eficiência como percentual do teto atingível.
- `count_loc` e `count_sync_primitives`: medem objetivamente o tamanho do *kernel* e o número de primitivas de sincronização explícita.
- `measure_peak_bandwidth` e `measure_peak_gflops`: estabelecem os tetos medidos de banda e de fp32 usados no *roofline*.

Por padrão, o cálculo das métricas derivadas usa a **mediana** do tempo (e não a média), por ser robusta a *outliers* lentos causados por *throttling* térmico, particularmente relevante em GPUs de *laptop*. A média e o desvio são mantidos no relatório de saída como medida de variância.

5 Caso de Controle: SAXPY

5.1 Justificativa

Antes de avaliar padrões que diferenciam os modelos, é necessário validar o próprio instrumento. O SAXPY ($y \leftarrow \alpha x + y$) é o caso de controle ideal: é fortemente *memory-bound* (intensidade aritmética $\approx 0{,}17$ FLOP/byte) e não oferece margem para abstração alguma se destacar. Espera-se, portanto, que as três implementações caiam praticamente no mesmo ponto do *roofline*, perto do teto de banda. Qualquer desvio desse comportamento indica problema no instrumento, não diferença entre modelos.

5.2 Ambiente experimental

Os experimentos preliminares foram conduzidos na GPU descrita na Tabela 4. Vale notar que a confirmação empírica de que o cuTile instala e executa nessa placa (*compute capability* 8.9) resolveu uma dúvida inicial de viabilidade de *hardware*: a lista de requisitos do cuTile foi atualizada para incluir a geração 8.x, e o *quickstart* oficial executou com sucesso.

Tabela 4: Ambiente experimental utilizado nos resultados preliminares.

Item	**Valor**
GPU	NVIDIA GeForce RTX 4060 Laptop GPU (Ada Lovelace)
Compute capability	8.9
Memória global	8,6 GB
Cache L2	34 MB
Banda teórica (aprox.)	256 GB/s
Banda de pico *medida*	223 GB/s ($\approx$87% do teórico)
Pico fp32 *medido* (GEMM)	6,8 TFLOPS (limitado por potência/térmica)

5.3 A anomalia do regime *cache-resident*

A primeira versão do *harness* produziu eficiências superiores a 100 % do teto de banda, algo aparentemente impossível. A investigação mostrou que *não* se tratava de erro de medição de tempo, mas da colisão entre um modelo analítico de tráfego (que assume que todo o tráfego vai à DRAM) e a realidade da *cache* L2. Para um SAXPY com $N = 1{,}05 \times 10^6$ elementos, a pegada

residente (≈8 MB) cabe inteira na L2 de 34 MB; após o *warmup*, as leituras batem na *cache*, não na DRAM, de modo que a "banda" calculada é fictícia. O gradiente observado confirma o diagnóstico: à medida que a pegada ultrapassa a *cache*, a eficiência aparente converge para o valor real (Tabela 5).

Tabela 5: Eficiência aparente *versus* pegada de memória (SAXPY, versão 1). O efeito de *cache* desaparece à medida que a pegada excede a L2.

N	Pegada residente	Regime	Eficiência aparente
1×10^6	≈8 MB	cabe na L2	343 %
8×10^6	≈67 MB	≈2× a L2	125 %
64×10^6	≈512 MB	muito acima da L2	104 %

5.4 Ajustes metodológicos (versão 2)

A análise motivou quatro ajustes no *harness*, que tornam confiáveis todos os padrões subsequentes:

1. **Dimensionamento ciente de *cache*.** Os tamanhos deixam de ser fixos: a L2 real é lida do dispositivo e derivam-se os N que cobrem os três regimes (*cache-resident*, transicional, *dram-bound*), permitindo comparar os modelos no regime em que o *roofline* de DRAM é válido e ainda estudar deliberadamente o regime *cache-resident*.
2. **Pico de banda robusto.** Um microbenchmark de *streaming* (≈1 GB, bem acima de qualquer L2) com *best-of-N* substitui a medição anterior, que subestimava o pico (chegava a reportar um "pico" menor que a banda de um *kernel* real).
3. **Pico de fp32 medido.** Um GEMM grande de cuBLAS estabelece o teto real de fp32, substituindo o valor *placeholder*. Não afeta o SAXPY (*memory-bound*), mas é o que torna confiável o *roofline* dos padrões *compute-bound* futuros.

5.5 Resultados validados

Após os ajustes, o instrumento comporta-se como esperado (Tabela 6). O pico de banda medido (223 GB/s) é ≈87 % do teórico — valor saudável e nunca inferior à banda de um *kernel* real. Nos regimes transicional e *dram-bound*, a eficiência cai para a faixa realista de 75–95 % esperada de um SAXPY *memory-bound*. O regime *cache-resident* continua exibindo valores aparentes acima de 100 %, o que é esperado e não problemático: como a comparação entre modelos se dá no regime *dram-bound*, esse ponto não contamina as conclusões.

Tabela 6: SAXPY após os ajustes (versão 2). Eficiência calculada contra o pico de banda *medido* (223 GB/s).

Regime	**Eficiência (banda efetiva)**	**Observação**
cache-resident	>100 % (artefato)	dados servidos pela L2; banda fictícia
transicional	90,8 %	pegada ≈2× a L2
dram-bound	87,5 %	regime de referência, *roofline* válido

Em resumo, o SAXPY cumpriu o papel de caso de controle: expôs exatamente os efeitos que um *kernel memory-bound* trivial deveria expor (*cache* + *roofline* de DRAM + pico subestimado), e os ajustes 1–4 resolveram a anomalia de forma definitiva. O instrumento está validado.

6 Primeiro Padrão Diferenciador: Redução Paralela

6.1 Por que a redução

A redução ("vetor → escalar", e.g. soma de todos os elementos) é o primeiro padrão em que a diferença de *expressividade* entre os modelos deve aparecer de forma nítida. No baseline, a redução exige uma árvore em memória compartilhada, *shuffles* de *warp* (`__shfl_down_sync`) e barreiras `__syncthreads()`; no cuTile e no Triton, resolve-se com uma única chamada de primitiva (`ct.sum` no cuTile, `tl.sum` no Triton). A redução é *memory-bound* (intensidade aritmética $\approx 0{,}25$ FLOP/byte), logo, em desempenho, espera-se que os três modelos encostem no teto de banda, na mesma faixa do SAXPY no regime *dram-bound* (75–95 %).

6.2 Implementação e princípio de equidade

O modelo de custo da redução é simples e fixo: para N elementos em fp32, flops $= N$, os *bytes* do modelo somam $4N$ (uma leitura por elemento) e a pegada residente é $4N$, o que dá intensidade aritmética $\approx 0{,}25$ FLOP/byte: firmemente *memory-bound*. As três implementações compartilham a mesma estrutura lógica em dois níveis (vetor → parciais → escalar), respeitando o princípio de equidade descrito na Seção 3: a comparação isola a variável "abstração" porque todos os *backends* partilham a mesma camada de memória (CuPy) e o mesmo protocolo de medição.

O *baseline* é a expressão idiomática competente do padrão em CUDA C++ (`cp.RawKernel` via NVRTC). Ele faz a redução em árvore: acumulação em registrador durante a carga via *grid-stride*, redução intra-*warp* por `__shfl_down_sync`, publicação dos parciais de cada *warp* em memória compartilhada, uma barreira `__syncthreads()` e a redução final pelo primeiro *warp*. O número de parciais é limitado por um *grid-stride* com número de blocos fixo, e o segundo estágio reaproveita o mesmo *kernel* com um único bloco. O cuTile, por sua vez, expressa a mesma operação através da primitiva de redução `ct.sum`, sem árvore manual nem barreiras: o padrão multi-passo idiomático reduz por um fator de bloco a cada passo até restar um elemento.

O Triton segue a mesma estrutura multi-passo (cada passo reduz por um fator de bloco fixo até restar um elemento), usando a primitiva `tl.sum`. Há, porém, uma sutileza de interoperabilidade: a *build* de Triton utilizada rejeita *arrays* CuPy, de modo que os dados são entregues por uma ponte *zero-copy* via DLPack [12] para *torch* (`torch.from_dlpack`), construída uma *única* vez fora do laço cronometrado, para não medir o custo de CPU da ponte a cada iteração e, sobretudo, para preservar a camada de memória comum (os mesmos *buffers*) e, com ela, a equidade da comparação. O *timing* do Triton usa eventos *torch*, pois o *kernel* roda na *stream* do *torch*. Como o cuTile, o Triton não expõe nenhuma primitiva de sincronização.

A validação numérica precede toda medição: a referência é calculada em float64 e a tolerância relativa é relaxada (`rtol`$= 10^{-2}$), porque a soma em fp32 de milhões de termos acumula erro e cada *backend* reordena as somas de forma diferente. Exigir igualdade exata confundiria diferença numérica legítima com defeito de implementação.

Refinamento da métrica qualitativa. No padrão de redução, os *shuffles* de *warp* são cooperação explícita entre *threads* tanto quanto a barreira `__syncthreads()`. Por isso o contador de primitivas de sincronização do *baseline* foi estendido para incluir também `__shfl_down_sync` (e variantes `__shfl_xor_sync`/`__shfl_sync`). Essa correção reforça e torna honesto o contraste com cuTile e Triton, que não expõem nenhuma dessas primitivas.

6.3 Refinamentos de medição e a régua empírica

O desenvolvimento da redução expôs, além da anomalia de *cache* já discutida para o SAXPY (Seção 5), dois problemas adicionais na definição do teto de referência. Narrá-los é parte da metodologia, pois foram o que levou à régua adotada em todo o restante do estudo.

Régua casada ao padrão de acesso. Mesmo no regime *dram-bound*, a eficiência ainda ultrapassava 100 %. A causa não estava no tempo do *kernel*, mas no teto: o pico era medido com `cp.copyto` (leitura + escrita), ao passo que a redução é *read-only* e, no GDDR6, a leitura pura sustenta banda *maior* que a cópia. A régua ficava, portanto, abaixo do que o *kernel* real alcançava. A correção foi medir o pico com um padrão de acesso *casado*: um teto *read-only* para a redução (via `cp.sum`), mantendo o teto de cópia apenas onde ele é o correto (o SAXPY, que lê e escreve).

Régua empírica fixa (*roofline* empírico). Persistia ainda uma oscilação entre processos: execuções "boas" ($\approx$98 %) e "ruins" ($>$110 %). A investigação mostrou que a eficiência acompanhava o *pico medido*, não o tempo do *kernel*: a mediana do tempo era estável (coeficiente de variação de $\approx$0,2 % no *dram-bound*), mas o pico *read-only* variava de processo para processo, e como `cp.sum` opera na mesma faixa de banda do próprio *kernel*, a razão entre os dois ficava à mercê do ruído. A decisão adotada, prática comum na literatura, foi calibrar o pico *uma única vez*, fixá-lo, e usá-lo como *denominador comum* para todos os *backends* e execuções. Assim, a eficiência passa a significar "fração da melhor banda observada para este padrão de acesso neste *hardware*", é $\leq 100\,\%$ por construção e preserva a comparabilidade; a variação que resta reflete *throttling* real, não ruído da régua.

Dois utilitários materializam essas decisões e tornam o protocolo reprodutível. O `calibrate_peak.py` mede os picos uma vez e os grava (com *timestamp*) num arquivo de calibração lido pelos *benchmarks* — a régua *read-only* da redução ($\approx$248 GB/s), o pico fp32 de *CUDA cores* e o pico de *Tensor Core* em bf16 usado no GEMM; recalibra-se ao trocar GPU, *driver* ou condição térmica. O `run_repeats.py` formaliza um segundo nível de repetição: enquanto o laço interno de `time_kernel` ($\geq$50–100 iterações mais *warmup*) captura o *jitter* dentro de um processo, apenas relançar o programa como *processos novos* reseta a compilação JIT (Triton e cuTile compilam uma vez por processo), o estado do alocador/contexto CUDA e o estado térmico inicial. Um laço ingênuo dentro do `main` não bastaria, pois não reinicia nada disso. O utilitário relança o alvo como subprocessos independentes (cinco execuções, com *cooldown* entre elas), agrega as medianas entre processos e reporta o coeficiente de variação (CV) inter-processo por configuração.

6.4 Estimador central: média ou mediana?

A convenção de boa parte da literatura é reportar média $\pm$ desvio, e há bons motivos: a média usa todos os dados, é o estimador natural sob ruído gaussiano e o desvio comunica a variabilidade diretamente. O problema é que o tempo de *kernel* de GPU não tem ruído simétrico — ele é enviesado à direita. O *hardware* impõe um piso de tempo (não há como ir mais rápido que o limite físico), mas qualquer perturbação (*jitter* do SO, *hiccup* de *launch* e, sobretudo, *throttling* térmico) só empurra medições para mais lento. A cauda é toda de um lado. Nesse regime, a média é puxada para cima pelos *runs* lentos, enquanto a mediana representa melhor o desempenho típico atingível.

O efeito é concreto nos próprios dados: no cuTile da redução no maior tamanho, a mediana entre execuções é 2,170 ms e a média 2,294 ms, aproximadamente 5,7 % de diferença, com a média inflada justamente pelos *runs* sob *throttling*. A prática adotada atende às duas exigências: usa-se a *mediana* como estimador central nas figuras e no valor de destaque (robusta ao *throttling*

do *laptop*), reportando *média* $\pm$ desvio como medida de dispersão. Se os *clocks* puderem ser travados, a diferença entre os dois encolhe, o que por si só é um dado honesto a registrar.

6.5 Resultados da redução

Todos os números de produção a seguir provêm do ambiente unificado (WSL2, *filesystem* nativo, *toolchain* alinhada) descrito na Seção 8, onde os três *backends* foram medidos lado a lado. Com a régua fixa calibrada (pico *read-only* ≈248 GB/s), a Figura 1 posiciona os três *backends* no *roofline* do padrão. As figuras são geradas por *padrão* (e não combinadas), de modo que cada uma usa o teto de *compute* correto para o seu *dtype* (fp32 na redução) e conta a história daquele regime; isso também mantém as figuras legíveis à medida que novos padrões são adicionados. Os três pontos caem sobre a rampa de banda, na mesma intensidade aritmética (≈0,25 FLOP/byte), saturando a régua, sendo esse o comportamento esperado de um *kernel memory-bound*.

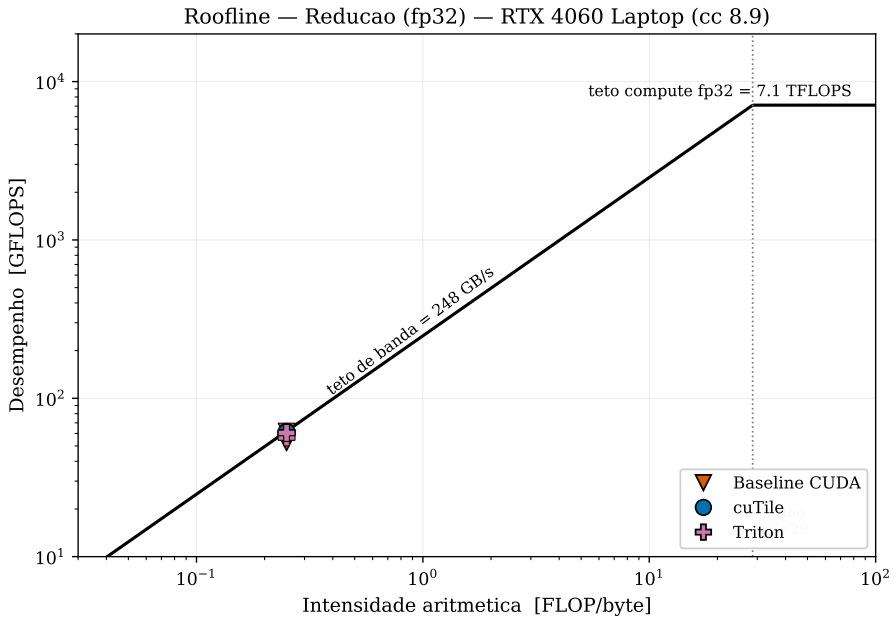

Figura 1: *Roofline* da redução (fp32) na RTX 4060 Laptop (cc 8.9). Os três modelos coincidem sobre a rampa de banda, na intensidade aritmética ≈0,25 FLOP/byte, muito distante do joelho, ou seja, saturando a banda, como esperado de um padrão *memory-bound*.

A Figura 2 mostra a largura de banda efetiva em função do tamanho do problema, com barras de erro de reprodutibilidade entre processos. Os três modelos aproximam-se do teto (≈248 GB/s) à medida que a pegada cresce; no menor tamanho (regime transicional) aparece uma leve hierarquia real (cuTile $>$ Triton $>$ *baseline*), que se dissolve no regime *dram-bound*, onde todos saturam a banda.

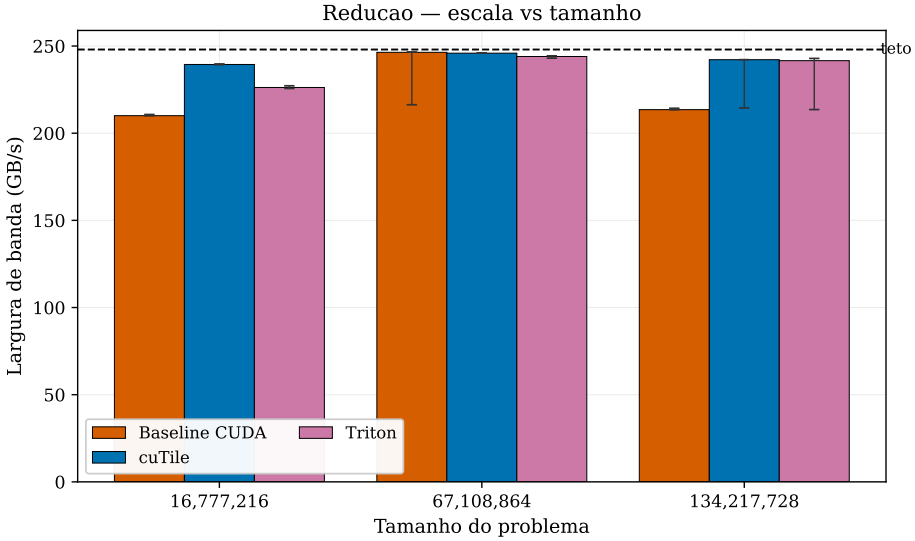

Figura 2: Escala da redução: largura de banda efetiva *versus* tamanho do problema para os três *backends*. Todos convergem para a régua de ≈248 GB/s; as barras de erro refletem a dispersão entre processos.

A Tabela 7 consolida desempenho e esforço. O contraste central do padrão, porém, aparece de forma mais imediata na Figura 3: o gráfico de *desempenho* × *esforço*, tomado no tamanho limpo de 67M (ponto *dram-bound* sem ambiguidade de performance), onde os três *backends* são praticamente idênticos em desempenho (≈99 % da régua). cuTile e Triton coincidem no canto de baixo esforço (6 linhas de código e *zero* primitivas de sincronização), enquanto o *baseline* entrega o mesmo desempenho ao custo de 22 linhas e 3 primitivas (uma `__syncthreads()` e dois `__shfl_down_sync`). Essa é a manchete do padrão: as duas abstrações não custaram banda e ainda cortaram o código por um fator de ≈3,7.

Tabela 7: Desempenho e esforco no padrao de reducao (fp32). Ambos os modelos saturam a banda de DRAM; o contraste esta no esforco.

Modelo	N	Tempo (ms)	Banda (GB/s)	Efic. (%)	LOC
Baseline CUDA	16,777,216	0.319	210.1	84.3	22
Baseline CUDA	67,108,864	1.089	246.4	98.9	22
Baseline CUDA	134,217,728	2.514	213.5	85.7	22
cuTile	16,777,216	0.280	239.4	96.1	6
cuTile	67,108,864	1.092	245.9	98.7	6
cuTile	134,217,728	2.217	242.2	97.2	6
Triton	16,777,216	0.297	226.2	90.8	6
Triton	67,108,864	1.100	244.0	98.0	6
Triton	134,217,728	2.222	241.6	97.0	6

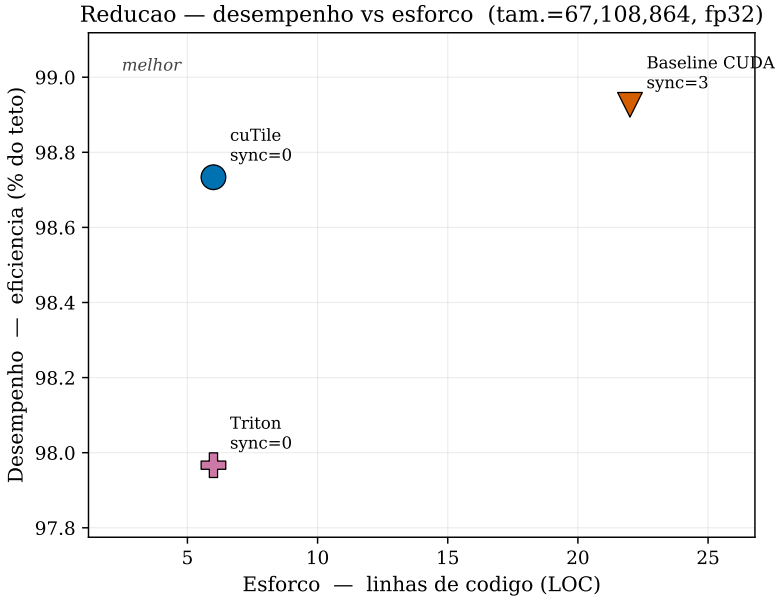

Figura 3: Redução — desempenho *versus* esforço em $N = 67{,}1 \times 10^6$ (fp32). cuTile e Triton ocupam o canto "alto desempenho, pouco código" (6 LOC, *sync* 0); o *baseline* entrega desempenho equivalente ao custo de código muito maior (22 LOC, *sync* 3).

Quatro conclusões resultam do padrão de redução. Primeiro, no regime que importa (o *dram-bound* de 67M), há *empate técnico* de desempenho: os três *backends* de alto nível e o *baseline* igualam-se no teto de banda (≈99 %), de modo que a abstração não custou largura de banda. Segundo, o resultado *qualitativo* é o contraste decisivo: cuTile e Triton usam 6 linhas e *zero* primitivas de cooperação, contra 22 linhas e 3 primitivas do *baseline*. Terceiro, no regime transicional (16,7M) há uma leve hierarquia *real* (cuTile ≈96 % > Triton ≈91 % > *baseline* ≈84 %), atribuível ao tratamento da fronteira de *cache*, o regime mais sensível, mas aqui com CV baixo e consistente entre *runs*. Quarto, e mais importante como ressalva: por ser *memory-bound*, este resultado era esperado e *não* estressa os compiladores do cuTile e do Triton. É precisamente por isso que o padrão seguinte, o GEMM, *compute-bound*, foi incluído.

Correção de interpretação (Triton e o *filesystem*). Uma aparente ineficiência do Triton na redução (≈73 %), observada em medições iniciais, era em boa parte contaminação de I/O do *filesystem* do Windows montado no WSL2 (`/mnt/c`), e não uma característica do `tl.sum` [19]; no *filesystem* nativo, o Triton cola no teto como os demais (Seção 8).

Uma ressalva sobre o ponto de 134M. Nesse tamanho, o comportamento é bimodal e bem definido: os três *backends* compartilham as *mesmas* duas populações de performance ("boa" ≈2,20 ms/≈97 %; "ruim" ≈2,51 ms/≈85 %), por *throttling* térmico do *laptop* sem *clocks* travados. Nesta bateria, o *baseline* caiu quase todo na população "ruim" enquanto cuTile e Triton ficaram majoritariamente na "boa"; a diferença entre cuTile e Triton nesse ponto é essencialmente *nula* (medianas de ≈2,22 ms). Trata-se, portanto, de artefato de amostragem das populações, não de desempenho de *backend*. A leitura correta é a do ponto limpo de 67M; onde o *throttling* introduz ambiguidade (134M), o correto é reportar as duas populações separadamente ou notar a equivalência.

7 Segundo Padrão Diferenciador: GEMM com *Tiling* (*Tensor Cores*)

Se a redução testa o eixo da *produtividade* em um cenário onde o desempenho já está saturado, o GEMM (multiplicação de matrizes densas) testa o eixo do *desempenho*. É um padrão *compute-bound* e é exatamente o cenário para o qual o cuTile foi desenhado, e onde a diferença entre os modelos deve aparecer também nos números, e não só no código.

7.1 Motivação e escolha de precisão

A decisão, alinhada ao objetivo do estudo, foi usar os *Tensor Cores*, com precisão principal em **bf16 com acumulação em fp32**, o padrão real do *machine learning* moderno. Descartaram-se o fp32 puro (que não aciona *Tensor Cores*) e o fp16 com acumulação em fp16 (numericamente instável). O bf16 foi preferido ao fp16 por ter o mesmo expoente do fp32, o que traz robustez numérica e uma validação mais tranquila; o fp16 fica como possível segundo ponto de precisão no futuro. Para uma matriz quadrada $S \times S$, tem-se flops $= 2S^3$ e bytes $= 6S^2$ (A, B e C, dois *bytes* cada), o que dá intensidade aritmética $S/3$ FLOP/byte, firmemente *compute-bound*. A validação usa a norma relativa de Frobenius contra uma referência em fp32, com tolerância relaxada (3 %), pois o bf16 introduz erro real e esperado.

7.2 Camada de memória: *torch* e a ponte para o *baseline*

Este padrão revê a decisão de usar CuPy como camada de memória. A verificação mostrou que o suporte a bf16 no CuPy é recente e frágil (dependente de `ml_dtypes`, exigindo versões muito novas de NumPy e com lacunas de cobertura), enquanto o *torch* tem bf16 nativo e robusto, e o próprio exemplo de referência da NVIDIA usa tensores *torch*. Adotou-se, portanto, o *torch* como camada de memória do padrão GEMM, com *todos* os *backends* operando sobre ele. A equidade é preservada *dentro* da comparação (todos partilham *torch*); a inconsistência entre padrões (redução em CuPy, GEMM em *torch*) é documentada e justificada pelo bf16. A escolha traz três vantagens: `torch.matmul` em bf16 é o cuBLAS [12, 13] com *Tensor Core* (o teto "de graça"), o Triton aceita tensores *torch* nativamente (sem a ponte necessária na redução), e o *timing* por `torch.cuda.Event` ocorre corretamente na *stream* do *torch*.

O *baseline* manual, contudo, é escrito em CUDA C++ (`cp.RawKernel`), enquanto os dados vivem em tensores *torch* bf16. A ponte é *zero-copy*: o tensor *torch* é visto como um *array* CuPy `uint16` sobre a *mesma* memória (o ponteiro é idêntico; o *kernel* apenas reinterpreta os *bits* como `__nv_bfloat16*`), e o *kernel* é lançado na *stream* do *torch* (via *external stream*) para que o *timing* por eventos o capture corretamente. Essa ponte é reutilizada pelos dois *baselines* descritos a seguir. Vale registrar um detalhe de ambiente que custou tempo: no Windows, o *torch* do PyPI vem *sem* CUDA por padrão (erro *"Torch not compiled with CUDA enabled"*); é preciso instalar a *wheel* com CUDA e confirmar a disponibilidade do dispositivo antes de medir.

7.3 As cinco implementações e as duas *baselines*

O GEMM emprega cinco *backends*, com uma decisão de projeto importante: *dois baselines* manuais, não um. Cada um responde a uma pergunta distinta e, juntos, tornam a comparação completa e defensável.

- **`baseline_simt`** — GEMM *tilado* 16×16 em memória compartilhada, usando *CUDA cores* (*sem Tensor Core*), com acumulação em fp32. Responde: "quanto custa *não* usar *Tensor Cores*?" ($\approx$25 LOC, 2 primitivas de sincronização).

- **`baseline_wmma`** — GEMM WMMA idiomático (a versão "livro-texto"): um *warp* por *tile* $16 \times 16 \times 16$, com fragmentos carregados direto da memória global a cada passo de K, acumulação em fp32 e escrita de volta com conversão. Sem *staging* em memória compartilhada, sem *double-buffering*, sem *register blocking*. É a expressão mínima correta do WMMA. Responde: "quanto custa usar *Tensor Cores à mão*, de forma idiomática?" (24 LOC, 6 primitivas. As operações `wmma::` são contadas como cooperação, pois são a maquinaria que a primitiva do cuTile esconde).
- **cuTile** — GEMM adaptado do exemplo *MatMul* da NVIDIA [20]: acumulador em fp32, laço sobre os *tiles* de K com carga em modo de *padding* zero, produto pela primitiva `ct.mma` e conversão final. *Tiles* de $128 \times 128 \times 64$ (16 LOC, *zero* primitivas de sincronização).
- **Triton** — *kernel matmul* idiomático com a primitiva `tl.dot` (o análogo direto ao `ct.mma` do cuTile), acumulação em fp32 e saída bf16. Usam-se blocos *fixos* ($128 \times 128 \times 32$, `GROUP_M`=8) em vez de `@triton.autotune`, para uma comparação determinística e coerente com os *tiles* fixos do cuTile (o *auto-tuning*, realce idiomático do Triton, fica como extensão opcional). 32 LOC, *zero* primitivas de sincronização.
- **cuBLAS** — `torch.matmul` em bf16 (*Tensor Cores*), o teto *vendor* de referência.

A razão de manter as duas *baselines* é decompor o ganho em dois eixos: o `baseline_simt` isola o ganho de *hardware* de simplesmente usar *Tensor Cores* (SIMT → WMMA), enquanto o `baseline_wmma` isola o valor da *abstração* sobre o código de *Tensor Core* manual (WMMA → cuTile/Triton).

Dois níveis de abstração. Embora cuTile e Triton compartilhem a ausência de cooperação explícita (*sync* 0), eles não estão no mesmo nível de abstração. O *kernel* Triton é mais verboso que o do cuTile (32 *versus* 16 LOC) porque expõe explicitamente a aritmética de ponteiros, os *strides* e as máscaras de contorno, enquanto o cuTile abstrai tudo atrás de índices de *tile*. Isso revela dois pontos distintos dentro do campo "alto nível": mais abstração (cuTile) *versus* mais controle (Triton): um *trade-off* que o padrão GEMM torna visível e que enriquece a tese para além de "abstração *versus* manual". Os Trechos 1 e 2 tornam concreto esse contraste. Nos dois, o laço percorre os *tiles* de K acumulando em fp32; a diferença está no que fica exposto. No cuTile, cada passo é `ct.load` → `ct.mma`, e o *kernel* lê-se como álgebra linear; no Triton, a mesma operação exige aritmética explícita de ponteiros, *strides* e máscaras. (A ordenação agrupada de blocos, para reúso de L2, aparece *inline* no Triton e fatorada num *helper* no cuTile; é parte do que separa 16 de 32 linhas. Mesmo se considerássemos todas as linhas do helper em LOC, o cuTile permaneceria sendo menos extenso, mantendo a análise válida nesse caso).

```
@ct.kernel
def _cutile_matmul_kernel(A, B, C, tm: ConstInt, tn: ConstInt, tk: ConstInt):
    M = A.shape[0]
    N = B.shape[1]
    bid = ct.bid(0)
    bidx, bidy = _swizzle_2d_from_bid(M, N, tm, tn, CUTILE_GROUP_SIZE_M, bid)
    num_tiles_k = ct.num_tiles(A, axis=1, shape=(tm, tk))
    acc = ct.full((tm, tn), 0, dtype=ct.float32)     # fp32 accumulation
    zero_pad = ct.PaddingMode.ZERO
    dtype = ct.tfloat32 if A.dtype == ct.float32 else A.dtype
    for k in range(num_tiles_k):
        a = ct.load(A, index=(bidx, k), shape=(tm, tk), padding_mode=zero_pad
            ).astype(dtype)
```

```
        b = ct.load(B, index=(k, bidy), shape=(tk, tn), padding_mode=zero_pad
            ).astype(dtype)
        acc = ct.mma(a, b, acc)
    acc = ct.astype(acc, C.dtype)
    ct.store(C, index=(bidx, bidy), tile=acc)
```

Trecho de código 1: cuTile: *kernel* `_cutile_matmul_kernel` (escopo de módulo). O laço de K resume-se a `ct.load` e `ct.mma`; acumulação em fp32 e *cast* final para bf16.

```
@triton.jit
def matmul_kernel(a_ptr, b_ptr, c_ptr, M, N, K,
                  stride_am, stride_ak, stride_bk, stride_bn, stride_cm,
                      stride_cn,
                  BLOCK_M: tl.constexpr, BLOCK_N: tl.constexpr,
                  BLOCK_K: tl.constexpr, GROUP_M: tl.constexpr):
    pid = tl.program_id(0)
    num_pid_m = tl.cdiv(M, BLOCK_M)
    num_pid_n = tl.cdiv(N, BLOCK_N)
    num_pid_in_group = GROUP_M * num_pid_n
    group_id = pid // num_pid_in_group
    first_pid_m = group_id * GROUP_M
    group_size_m = min(num_pid_m - first_pid_m, GROUP_M)
    pid_m = first_pid_m + (pid % group_size_m)
    pid_n = (pid % num_pid_in_group) // group_size_m
    offs_am = (pid_m * BLOCK_M + tl.arange(0, BLOCK_M)) % M
    offs_bn = (pid_n * BLOCK_N + tl.arange(0, BLOCK_N)) % N
    offs_k = tl.arange(0, BLOCK_K)
    a_ptrs = a_ptr + (offs_am[:, None] * stride_am + offs_k[None, :] *
        stride_ak)
    b_ptrs = b_ptr + (offs_k[:, None] * stride_bk + offs_bn[None, :] *
        stride_bn)
    acc = tl.zeros((BLOCK_M, BLOCK_N), dtype=tl.float32)   # fp32
        accumulation
    for k in range(0, tl.cdiv(K, BLOCK_K)):
        a = tl.load(a_ptrs, mask=offs_k[None, :] < K - k * BLOCK_K, other
            =0.0)
        b = tl.load(b_ptrs, mask=offs_k[:, None] < K - k * BLOCK_K, other
            =0.0)
        acc = tl.dot(a, b, acc)                            # Tensor Core
            primitive
        a_ptrs += BLOCK_K * stride_ak
        b_ptrs += BLOCK_K * stride_bk
    c = acc.to(tl.bfloat16)
    offs_cm = pid_m * BLOCK_M + tl.arange(0, BLOCK_M)
    offs_cn = pid_n * BLOCK_N + tl.arange(0, BLOCK_N)
    c_ptrs = c_ptr + stride_cm * offs_cm[:, None] + stride_cn * offs_cn[None,
         :]
    c_mask = (offs_cm[:, None] < M) & (offs_cn[None, :] < N)
    tl.store(c_ptrs, c, mask=c_mask)
```

Trecho de código 2: Triton: *kernel* `matmul_kernel` idiomático, com `tl.dot` e aritmética explícita de ponteiros, *strides* e máscaras.

Por que o WMMA idiomático fica atrás. É essencial enquadrar o `baseline_wmma` como o esforço *idiomático*: a versão que um programador competente escreve primeiro, e não como um espantalho. Ele fica aquém do cuBLAS, do cuTile e do Triton por omitir as otimizações que uma implementação "séria" (estilo CUTLASS [16], que é essencialmente o que o cuBLAS faz)

aplicaria: não há *staging* em memória compartilhada nem reúso de dados (cada *warp* recarrega da memória global suas faixas de A e B a cada passo de K, e o *kernel* passa a ser dominado por tráfego global redundante); há um único *warp* por bloco (blocos minúsculos, sem reúso entre *warps* e com ocupação pobre); não há *software pipelining*/*double-buffering* (os *Tensor Cores* ficam ociosos esperando a memória global); não há *register*/*warp blocking*; não se usa cópia assíncrona (`cp.async`, disponível em cc 8.x) [1]; e não há *swizzling* da memória compartilhada.

O caminho para uma versão séria, considerando pontos como *block tiling* em memória compartilhada (a mudança de maior impacto, que corta o tráfego global por um fator significativo), múltiplos *warps* por bloco com *warp*/*register tiling*, *double-buffering*/*pipelining* com `cp.async`, *swizzling* e otimização de epílogo, é um esforço à parte, e fecharia boa parte da distância de desempenho. Essa observação visa deixar claro que a comparação é "abstração *versus* esforço manual *idiomático*" e não "*versus* o melhor WMMA possível".

Esse enquadramento é justamente o que dá força ao resultado. Os compiladores do cuTile e do Triton aplicam *automaticamente*, por trás das primitivas `ct.mma` e `tl.dot`, várias dessas otimizações (*tiling*, *pipelining*, *swizzling*): a abstração não apenas remove a maquinaria manual (declaração de fragmentos, *layouts*, *staging*, escrita de volta), como *entrega* otimizações que o *kernel* idiomático escrito à mão tipicamente omite.

7.4 A régua de *Tensor Core*

O teto de *Tensor Core* em bf16 exigiu um ajuste análogo ao da régua da redução. Calibrado inicialmente em um único tamanho ($S = 8192$), deu 27,2 TFLOPS, mas o cuBLAS em $S = 2048/4096$ atingia 28–29 TFLOPS, o que produziria eficiência *vendor* acima de 100 %. A causa é que um único tamanho de calibração não maximiza o *throughput* ($S = 8192$ já esbarra em limite térmico/de potência por ser uma carga longa). A correção foi varrer 2048/4096/8192 e tomar o *máximo*; no ambiente de produção (WSL2), essa varredura fixou o teto de *Tensor Core* bf16 em ≈30 TFLOPS, com o que o cuBLAS volta a ficar $\leq 100\,\%$. O pico fp32 de *CUDA cores*, por sua vez, afeta apenas o joelho do *roofline* e não a leitura *compute-bound* do GEMM.

7.5 Resultados do GEMM

Com a régua fixa (≈30 TFLOPS de *Tensor Core* em bf16) e a *toolchain* de compilação alinhada (Seção 8), a Figura 4 posiciona os cinco *backends* no *roofline*. Como o padrão é *compute-bound*, todos os pontos caem na região plana (à direita do joelho), e a leitura relevante é a distância vertical ao teto: cuTile, Triton e cuBLAS colados no topo, o WMMA manual bem abaixo e o SIMT no piso. (A coluna genérica de "banda GB/s" não tem significado para um *kernel compute-bound*; o que importa é TFLOPS/eficiência.)

A Figura 5 mostra a eficiência (fração do teto bf16) em função do tamanho. Tanto o cuTile quanto o Triton amortizam o *overhead* com o tamanho e estabilizam próximos ao cuBLAS: o cuTile sobe de ≈65 % em $S = 1024$ para ≈84 %, enquanto o Triton, mais rápido, acompanha o cuBLAS de perto e o *sustenta* melhor nos maiores tamanhos. O WMMA manual e o SIMT permanecem em patamares muito inferiores.

A Tabela 8 consolida os cinco *backends*, e a Figura 6 traz o gráfico de *desempenho* × *esforço* em $S = 4096$, a figura-tese do padrão. cuTile e Triton ocupam a faixa de alto desempenho e nenhuma cooperação explícita, mas em pontos distintos de esforço de código: o cuTile no canto mais econômico (16 LOC, *sync* 0, ≈84 % do teto) e o Triton um pouco à direita e acima (32 LOC, *sync* 0, ≈94 %, junto ao cuBLAS). O WMMA manual e o SIMT ficam muito abaixo em desempenho.

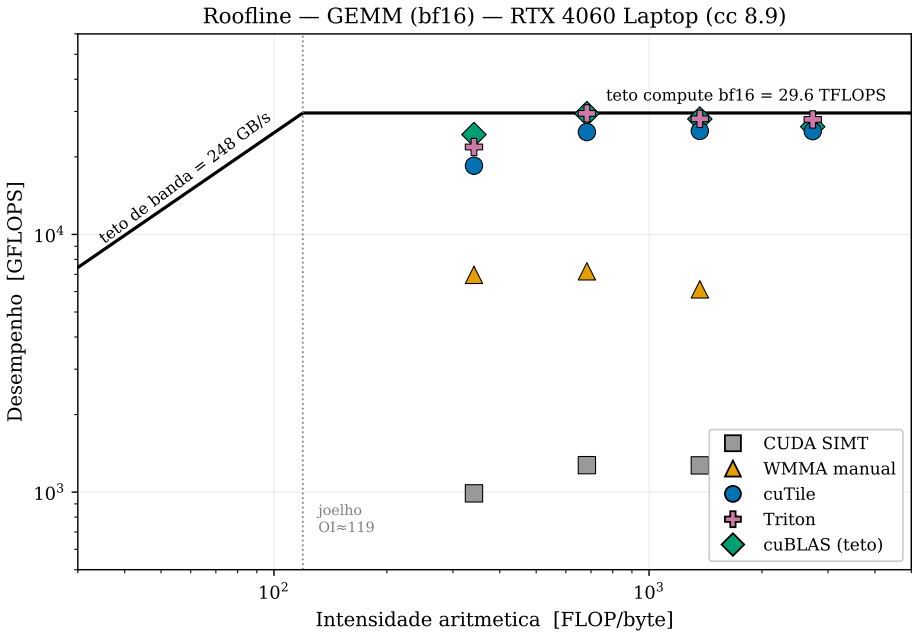

Figura 4: *Roofline* do GEMM (bf16) na RTX 4060 Laptop (cc 8.9). Na região *compute-bound*, cuTile, Triton e cuBLAS colam no teto de ≈30 TFLOPS, enquanto o WMMA manual e o SIMT ficam muito abaixo, a distância ao teto é a história do padrão.

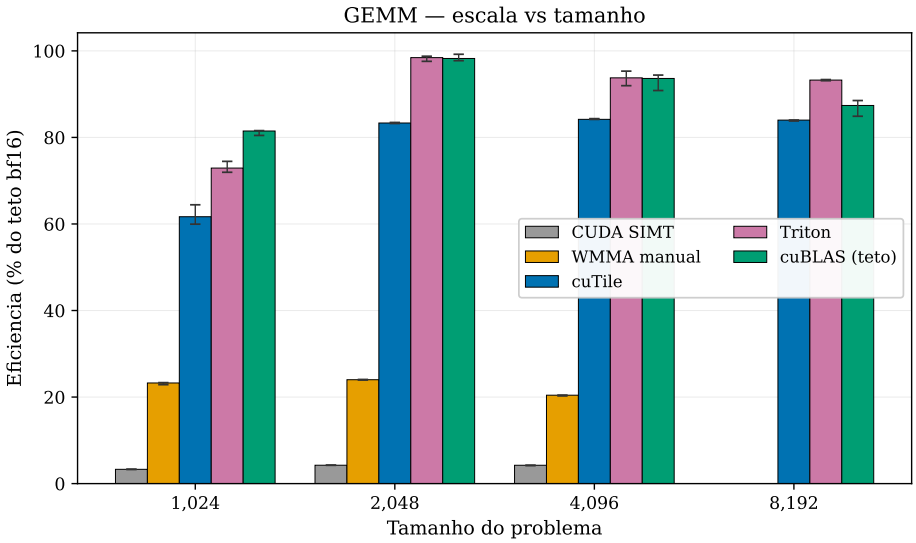

Figura 5: Escala do GEMM: eficiência (% do teto bf16) *versus* tamanho. cuTile e Triton crescem com o tamanho e estabilizam próximos ao cuBLAS (o Triton sustentando melhor a eficiência nos maiores tamanhos); o WMMA manual e o SIMT ficam bem atrás.

Tabela 8: Desempenho e esforco no padrao GEMM (bf16). Estimador central entre execucoes; LOC e primitivas de sincronizacao contadas objetivamente. cuBLAS entra como teto vendor.

Modelo	S	Tempo (ms)	TFLOPS	Efic. (%)	LOC	Sync
CUDA SIMT	1024	2.172	1.0	3.3	25	2
CUDA SIMT	2048	13.501	1.3	4.2	25	2
CUDA SIMT	4096	108.228	1.3	4.2	25	2
WMMA manual	1024	0.308	7.0	23.3	24	6
WMMA manual	2048	2.389	7.2	24.0	24	6
WMMA manual	4096	22.454	6.1	20.4	24	6
cuTile	1024	0.116	18.5	61.7	16	0
cuTile	2048	0.688	25.0	83.3	16	0
cuTile	4096	5.450	25.2	84.2	16	0
cuTile	8192	43.689	25.2	84.0	16	0
Triton	1024	0.098	21.8	72.9	32	0
Triton	2048	0.583	29.5	98.4	32	0
Triton	4096	4.893	28.1	93.8	32	0
Triton	8192	39.356	27.9	93.3	32	0
cuBLAS (teto)	1024	0.088	24.4	81.5	–	–
cuBLAS (teto)	2048	0.584	29.4	98.2	–	–
cuBLAS (teto)	4096	4.900	28.1	93.6	–	–
cuBLAS (teto)	8192	41.998	26.2	87.4	–	–

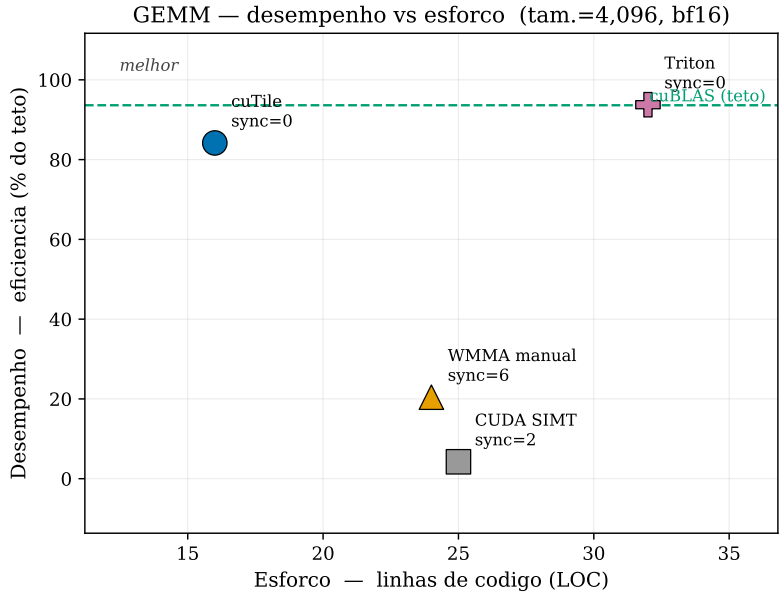

Figura 6: GEMM — desempenho *versus* esforço em $S = 4096$ (bf16). cuTile e Triton dominam a região de alto desempenho e zero primitivas de sincronização, com o cuTile mais conciso (16 LOC) e o Triton mais rápido (32 LOC); o WMMA manual e o SIMT ficam muito abaixo em desempenho.

A leitura dos resultados (Tabela 8) é clara e interessante. Em $S = 4096$, a ordenação é: Triton $\approx$ cuBLAS ($\approx$28 TFLOPS, $\approx$94 % do teto) $>$ cuTile ($\approx$25 TFLOPS, $\approx$84 %) $\gg$ WMMA manual ($\approx$6 TFLOPS) $>$ SIMT ($\approx$1,3 TFLOPS). Os dois eixos do estudo aparecem confirmados: o *Tensor Core* traz o salto de *hardware* (SIMT $\rightarrow$ WMMA, $\approx 5\times$), e as abstrações trazem produtividade *e* desempenho muito acima do esforço manual idiomático: tanto o cuTile quanto o Triton são $\approx 4\times$ mais rápidos que o WMMA manual e $\approx 20\times$ mais rápidos que o SIMT, com *zero* primitivas de sincronização. Aqui, portanto, a abstração *ganhou* desempenho, não apenas o economizou.

Cruzamento por tamanho e sustentação de eficiência. A hierarquia entre os modelos de alto nível depende do tamanho. Em matrizes pequenas ($S = 1024$), cuBLAS e Triton lideram e o cuTile fica atrás, pois seus *tiles* fixos subutilizam a carga; em matrizes grandes, o Triton assume a ponta. Em $S = 8192$, o Triton chega a ser $\approx$6 % mais rápido que o cuBLAS em tempo absoluto (com validação contra referência fp32 e CV baixo em cinco *runs*). O enquadramento correto disso não é "o Triton supera o teto", é importante esclarecer que nenhum *backend* passa de 100 % da régua, mas "o Triton *sustenta* melhor a eficiência em tamanhos grandes": nesta GPU *consumer*, o cuBLAS perde eficiência de $\approx$98 % em $S = 2048$ para $\approx$87 % em $S = 8192$ (suas heurísticas são calibradas sobretudo para GPUs de *datacenter*), enquanto o Triton a mantém na faixa de 93–98 %.

A métrica adequada a cada padrão. No GEMM, o número de primitivas de sincronização é a métrica que separa nitidamente abstração de esforço manual: WMMA e SIMT expõem a maquinaria de baixo nível (6 e 2 primitivas), enquanto cuTile e Triton expõem *zero*. Já as linhas de código, no GEMM, contam uma segunda história, não a de abstração *versus* manual (a versão WMMA idiomática mínima é enxuta, 24 linhas), mas a do *espectro* entre os dois modelos de alto nível: 16 linhas do cuTile contra 32 do Triton. Por isso, adota-se o número de primitivas como métrica principal de esforço no GEMM (0 *versus* 6/2, onde ela é mais eloquente) e o LOC como métrica principal na redução (6 *versus* 22, onde ela diferencia bem claramente): escolher a métrica adequada a cada padrão, com justificativa, é mais válido do que aplicar um único gráfico mecanicamente. A leitura completa do GEMM, então, é dupla: as abstrações escondem a maquinaria de *Tensor Core* (*sync* $6 \rightarrow 0$) *e* entregam $\approx 4\times$ mais desempenho que o WMMA manual idiomático; entre elas, o cuTile prioriza a concisão e o Triton, o desempenho.

Por fim, registra-se a ressalva já antecipada: parte da vantagem das abstrações sobre o WMMA vem de o *baseline* ser a versão idiomática mínima (sem *staging*/*pipelining*). Um WMMA "sério" fecharia parte dessa distância, o que não altera a conclusão sobre produtividade, mas delimita com honestidade o alcance da comparação de desempenho.

8 A Inclusão do Triton e o Ambiente de Execução

A entrada do Triton como terceiro vértice da comparação, o segundo modelo de "abstração", ao lado do cuTile, exigiu um ambiente Linux, pois o Triton não roda em Windows. Isso motivou a migração de todo o estudo para um ambiente *único* (WSL2), no qual os *backends* passaram a ser medidos lado a lado. A unificação do ambiente é condição de validade da comparação triangular: só assim as diferenças observadas são atribuíveis aos modelos, e não a diferenças de plataforma. Esta seção documenta a configuração, os percalços que se tornaram parte da metodologia e um achado sobre a *toolchain* de compilação que é, em si, um resultado.

8.1 Configuração e armadilhas do ambiente WSL2

Alguns obstáculos surgiram ao montar o ambiente, e vale registrá-los como ameaças à validade que foram identificadas e controladas:

- **Interoperabilidade Triton ↔ CuPy.** A *build* de Triton usada rejeita *arrays* CuPy; a solução foi a ponte *zero-copy* via DLPack para *torch*, descrita na Seção 6.2.
- ***Filesystem*: `/mnt/c` *versus* nativo (crítico).** Rodar a partir do disco do Windows montado no WSL2 (`/mnt/c`) tem I/O muito lento e *contaminou* as medições com *outliers* extremos (variações de 0,5 ms a 3,2 s na mesma configuração, fisicamente impossíveis para o *kernel*), originados de *stalls* de I/O do *cache* de compilação JIT, que afetam Triton e cuTile (que compilam e mantêm *cache* em disco), mas não o *baseline* `RawKernel`. A correção foi mover o projeto para o *filesystem* nativo (ext4) e recriar o ambiente virtual ali; os *outliers* desapareceram e os CV inter-processo caíram para menos de 3 % na maioria dos casos. A recomendação que fica é nunca rodar *benchmarks* de `/mnt/c` no WSL2. É a origem da correção de interpretação sobre o Triton na redução (Seção 6.5).

8.2 *Toolchain* de compilação e desempenho: um achado

O percalço mais instrutivo tornou-se um resultado. No Windows, o cuTile rendia ≈85 % do teto no GEMM ($S = 4096$); nas primeiras execuções no WSL, caiu para ≈58 %, de forma estável (CV baixo) e *apenas* para o cuTile, enquanto cuBLAS e os dois *baselines* permaneceram idênticos entre os ambientes. Essa localização (só o cuTile regride) foi a chave do diagnóstico: o cuTile é um *compilador*, e seu desempenho depende do *back-end* de compilação (`tileiras` → `ptxas`/`libnvvm`), que gera o SASS final. O Windows usava o CUDA Toolkit de sistema na versão 13.3 (`ptxas` V13.3), enquanto a instalação inicial no WSL trouxe o *back-end* como pacotes Python na versão 13.2. O mesmo *kernel* cuTile rendia ≈58 % com `ptxas` 13.2 e ≈84 % com 13.3. O cuBLAS não é afetado (é binário pré-compilado) e os *baselines* usam o `ptxas` do CuPy (via NVRTC), não o do cuTile. Alinhar a *toolchain* do WSL ao 13.3 restaurou os ≈84 %, confirmando a hipótese.

Este é um resultado legítimo e relevante para o estudo: a demonstração empírica de que o desempenho de um modelo de programação baseado em *compilador* depende *materialmente* da versão da *toolchain* de compilação por trás dele: o mesmo *kernel*, sem qualquer alteração de código, varia de ≈58 % para ≈84 % do teto conforme a versão do `ptxas`. É um fator de validade que qualquer comparação envolvendo modelos compiladores precisa controlar e reportar. (Um aviso cosmético de conflito de dependências entre o *torch* e o `cuda-toolkit` 13.3 foi verificado como inócuo: as bibliotecas mantêm compatibilidade dentro da mesma *major* 13.x, confirmado por um *matmul* bf16 real na GPU.)

8.3 Ambiente reprodutível

Todos os números de produção deste relatório provêm de um ambiente único, aqui registrado para reprodutibilidade: GPU NVIDIA GeForce RTX 4060 Laptop (*compute capability* 8.9, 8 GB, L2 de 34 MB, *driver* r580); execução em WSL2 (Ubuntu) com o projeto no *filesystem* nativo Linux (não em `/mnt/c`); Python 3.14; *toolchain* de compilação do cuTile (`tileiras`/`nvcc`/`nvvm`) na versão 13.3; `cuda-tile` 1.5.0; *torch* 2.13.0 (*build* com CUDA); além das versões exatas de CuPy e Triton, a anotar. A régua calibrada neste ambiente tem pico *read-only* de ≈248 GB/s e teto de *Tensor Core* bf16 de ≈30 TFLOPS. Os números coletados anteriormente no Windows nativo passam a ser referência histórica, úteis apenas para ilustrar o impacto de ambiente/*toolchain*

(Seção 8.2). A comparação triangular oficial é conduzida inteiramente neste ambiente unificado. Todo o código-fonte e os dados brutos de todas as execuções estão disponíveis publicamente em `https://github.com/zephyr-i/cudatile-study`, permitindo a reprodução integral dos resultados apresentados.

9 Terceiro Padrão Diferenciador: *Stencil* 2D (Equação do Calor)

Os dois primeiros padrões cobrem os extremos do *roofline*: a redução, *memory-bound* sem reúso, na base da rampa de banda; o GEMM, *compute-bound*, colado no teto de *Tensor Core*. O *stencil* 2D de diferenças finitas ocupa o *meio* desse espectro (intensidade aritmética ≈0,87 FLOP/byte) e introduz um terceiro tipo de cooperação, o *halo*, além de um domínio distinto, o da simulação numérica em física. É também o padrão que revela o cenário *oposto* ao do GEMM: aqui a abstração *perde* para o esforço manual, o que completa o espectro de conclusões da tese.

9.1 Motivação e escolha do padrão

O stencil foi escolhido por três razões. Primeiro, ele **preenche uma lacuna no *roofline***: enquanto a redução (AI $\approx 0{,}25$) e o GEMM (AI na casa dos milhares) são os extremos, o stencil fica no meio (AI $\approx 0{,}87$): ainda *memory-bound*, mas com um padrão de acesso qualitativamente diferente. Segundo, ele exercita um **terceiro tipo de cooperação, o *halo***: cada ponto depende dos quatro vizinhos, de modo que, ao tilar o domínio, os pontos de borda de um *tile* precisam de dados do *tile* vizinho (as *ghost cells*). É uma cooperação distinta da árvore da redução e da multiplicação de fragmentos do GEMM, e testa como cada modelo expressa vizinhança e bordas. Terceiro, ele traz um **domínio distinto** (física/simulação), reforçando a generalidade das conclusões para além da IA (GEMM) e das primitivas de algoritmos (redução).

Entre os stencils possíveis, adotou-se a equação do calor de cinco pontos por ser o mais limpo: operador simples, referência fácil de validar e exemplo canônico.

9.2 Definição da operação e modelo de custo

A operação é um único passo de Jacobi *out-of-place*: lê-se `u` e escreve-se `u_new`, o que evita dependência de leitura/escrita no mesmo passo e torna a validação determinística. As bordas são fixas (Dirichlet): os pontos de borda não são atualizados e permanecem com valor constante, o *kernel* atualiza apenas o interior. Para o ponto interior,

$$u_{\text{new}}[i,j] = u[i,j] + \alpha\big(u[i-1,j] + u[i+1,j] + u[i,j-1] + u[i,j+1] - 4\,u[i,j]\big),$$

com $\alpha = 0{,}24$ (abaixo de 0,25, por estabilidade). Mede-se um único passo (um *kernel*), por consistência metodológica com a redução e o GEMM; a camada de memória é CuPy (o Triton, como na redução, recebe os dados pela ponte DLPack).

O modelo de custo dá flops $\approx 7N^2$ (quatro somas, uma subtração, uma multiplicação e uma soma por ponto interior) e bytes $= 8N^2$ (lê `u` uma vez e escreve `u_new`; o reúso dos vizinhos é servido por *cache*/registradores e não conta como tráfego adicional de DRAM). A intensidade aritmética resultante, ≈0,87 FLOP/byte, confirma o padrão como *memory-bound*, no meio do *roofline*. Não há teto *vendor*: não existe biblioteca NVIDIA canônica de stencil (ao contrário do cuBLAS para o GEMM), e para um *kernel memory-bound* o teto é a banda de memória.

9.3 A régua: casar ao reúso de leitura

A definição da régua do stencil passou por duas correções, ambas instrutivas. É a terceira vez no projeto que reaparece a ideia de *casar a régua ao comportamento real do* kernel*, não ao rótulo do padrão*. A primeira tentativa usou o pico de *cópia* (leitura + escrita), pelo raciocínio de que o stencil é read+write; o raciocínio acertava o *tipo* de acesso, mas produziu eficiência acima de 100 %. A causa é o reúso: o stencil de cinco pontos lê cada elemento cinco vezes (centro e quatro vizinhos), e quatro dessas leituras são servidas por *cache*, de modo que a banda efetiva supera a de uma cópia pura (que não tem reúso). A régua mais adequada para um *kernel memory-bound* com forte reúso de leitura é o **pico *read-only*** (≈249 GB/s): o gargalo prático é ler os vizinhos, e a eficiência volta a ficar $\leq 100\,\%$.

Vale a ressalva de que nenhuma das duas réguas é perfeita: a de cópia subestima (ignora o reúso) e a *read-only* ignora que o stencil também escreve; a verdade está entre as duas. Adotou-se a *read-only* por ser a defensável, a comum na literatura de *roofline* para stencils, e por manter a eficiência $\leq 100\,\%$.

9.4 As três implementações e as lições de API

O *baseline* (CUDA C++, `cp.RawKernel`) faz *tiling* em memória compartilhada com *halo*: cada bloco carrega um *tile* $(16{+}2)^2$, os 16×16 pontos internos mais a moldura de *halo* dos vizinhos, para a memória compartilhada, sincroniza com `__syncthreads()` e calcula o stencil a partir dela. É a expressão idiomática do padrão, e o carregamento do *halo* e o tratamento de borda são exatamente a maquinaria de baixo nível que as abstrações tentam esconder (20 LOC, 1 primitiva de sincronização). O Trecho 3 mostra essa maquinaria: quatro condicionais só para preencher a moldura de *halo*, mais a barreira que garante o *tile* completo antes do cálculo.

```
#define BLK 16
extern "C" __global__
void stencil5(const float* __restrict__ u, float* __restrict__ un,
              int n, float alpha) {
    __shared__ float s[BLK + 2][BLK + 2];
    int tx = threadIdx.x, ty = threadIdx.y;
    int gx = blockIdx.x * BLK + tx;          // global col
    int gy = blockIdx.y * BLK + ty;          // global row
    int sx = tx + 1, sy = ty + 1;            // shared idx (offset for halo)

    if (gx < n && gy < n) s[sy][sx] = u[gy * n + gx];          // center
    // halo: left/right columns and top/bottom rows of the frame
    if (tx == 0 && gx > 0 && gy < n)           s[sy][0]       = u[gy * n + (
        gx - 1)];
    if (tx == BLK - 1 && gx + 1 < n && gy < n) s[sy][BLK + 1] = u[gy * n + (
        gx + 1)];
    if (ty == 0 && gy > 0 && gx < n)           s[0][sx]       = u[(gy - 1) *
        n + gx];
    if (ty == BLK - 1 && gy + 1 < n && gx < n) s[BLK + 1][sx] = u[(gy + 1) *
        n + gx];
    __syncthreads();

    if (gx >= 1 && gx < n - 1 && gy >= 1 && gy < n - 1) {
        float lap = s[sy][sx - 1] + s[sy][sx + 1] + s[sy - 1][sx] + s[sy +
            1][sx]
                  - 4.0f * s[sy][sx];
        un[gy * n + gx] = s[sy][sx] + alpha * lap;
    }
```

```
}
```

Trecho de código 3: *Baseline* CUDA C++: *tile* em memória compartilhada com moldura de *halo*. As quatro condicionais de carga do *halo* e a barreira são a maquinaria que as abstrações escondem.

O cuTile expressa a vizinhança com `ct.gather`/`ct.scatter`, e o caminho até essa forma rendeu várias lições de API que vale registrar como parte do esforço de desenvolvimento. O `ct.load` *não* faz acesso deslocado por elemento (localiza apenas *tiles* alinhados por índice de *tile*), de modo que o acesso à vizinhança usa `ct.gather(u, (x, y))` com índices de coordenada explícitos, construídos com `ct.arange` e *broadcasting* (os vizinhos são $x \pm 1$, $y \pm 1$); o `ct.gather` retorna zero para índices fora dos limites. O escalar precisa ser convertido com `float(...)`, pois o `ct.launch` recusa `numpy.float32`. E, como o `ct.scatter` (Python) não tem argumento `mask`, as bordas fixas foram obtidas mascarando o *valor*, não a escrita: um indicador de interior como *float* (produto de comparações, sem `&` booleano) multiplica o laplaciano, de modo que na borda o resultado é o valor inalterado (o *scatter* vira *no-op*) e no interior é $u + \alpha \nabla^2 u$ (17 LOC, 0 primitivas de sincronização). O Trecho 4 mostra a forma resultante: sem *staging* nem barreira, a vizinhança é expressa como cinco `ct.gather` deslocados.

```
@ct.kernel
def stencil_kernel(u, un, alpha, ts: ct.Constant[int]):
    bi = ct.bid(0)
    bj = ct.bid(1)
    x = (bi * ts + ct.arange(ts, dtype=cp.int32))[:, None]    # rows  -> dim 0
    y = (bj * ts + ct.arange(ts, dtype=cp.int32))[None, :]    # cols  -> dim 1
    c  = ct.gather(u, (x, y))
    up = ct.gather(u, (x - 1, y))    # neighbor one row up
    dn = ct.gather(u, (x + 1, y))    # one row down
    lf = ct.gather(u, (x, y - 1))    # one col left
    rt = ct.gather(u, (x, y + 1))    # one col right
    lap = up + dn + lf + rt - 4.0 * c
    # fixed border: ct.scatter has no mask, so mask the VALUE instead
    n = u.shape[0]
    im = ((x >= 1).astype(cp.float32) * (x < n - 1).astype(cp.float32)
          * (y >= 1).astype(cp.float32) * (y < n - 1).astype(cp.float32))
    out = c + alpha * lap * im
    ct.scatter(un, (x, y), out)
```

Trecho de código 4: cuTile: vizinhança por `ct.gather` com índices de coordenada explícitos. O indicador de interior mascara o *valor*, não a escrita.

O Triton usa cinco `tl.load` deslocados (centro e quatro vizinhos), cada um com máscara de borda, e preserva as bordas fixas com `mask=interior` no `tl.store`. Uma lição qualitativa apareceu aqui: o Triton *não* traça funções Python aninhadas dentro do *kernel* (uma função auxiliar interna resultou em erro de compilação), então tudo precisa ser *inline*, um contraste com o cuTile, que aceitou a função auxiliar de *swizzle* no GEMM. O Trecho 5 evidencia o contraste com o cuTile: a mesma vizinhança, mas com a aritmética de ponteiros e uma máscara escrita à mão para cada um dos cinco acessos.

```
@triton.jit
def stencil_kernel(u_ptr, un_ptr, n, alpha, BS: tl.constexpr):
    pid_i = tl.program_id(0)    # row block
    pid_j = tl.program_id(1)    # col block
    i = (pid_i * BS + tl.arange(0, BS))[:, None]
    j = (pid_j * BS + tl.arange(0, BS))[None, :]
    interior = (i >= 1) & (i < n - 1) & (j >= 1) & (j < n - 1)
    # explicit masked loads (Triton cannot trace nested helper functions)
```

```
    c  = tl.load(u_ptr + i * n + j,       mask=(i >= 0) & (i < n) & (j >= 0)
        & (j < n), other=0.0)
    up = tl.load(u_ptr + (i - 1) * n + j, mask=(i - 1 >= 0) & (j >= 0) & (j <
         n),       other=0.0)
    dn = tl.load(u_ptr + (i + 1) * n + j, mask=(i + 1 < n) & (j >= 0) & (j <
        n),        other=0.0)
    lf = tl.load(u_ptr + i * n + (j - 1), mask=(i >= 0) & (i < n) & (j - 1 >=
         0),       other=0.0)
    rt = tl.load(u_ptr + i * n + (j + 1), mask=(i >= 0) & (i < n) & (j + 1 <
        n),        other=0.0)
    out = c + alpha * (up + dn + lf + rt - 4.0 * c)
    tl.store(un_ptr + i * n + j, out, mask=interior)
```

Trecho de código 5: Triton: cinco `tl.load` deslocados, cada um com sua máscara de borda; `mask=interior` no `tl.store` preserva as bordas fixas.

9.5 Resultados do stencil

Com a régua fixa (pico *read-only* ≈248 GB/s) e no ambiente unificado (Seção 8), a Figura 7 posiciona os três *backends* no *roofline*: todos caem sobre a rampa de banda, na intensidade aritmética ≈0,87 FLOP/byte, abaixo do teto, mas agora, ao contrário da redução, *sem* coincidirem, e com o *baseline* no ponto mais alto.

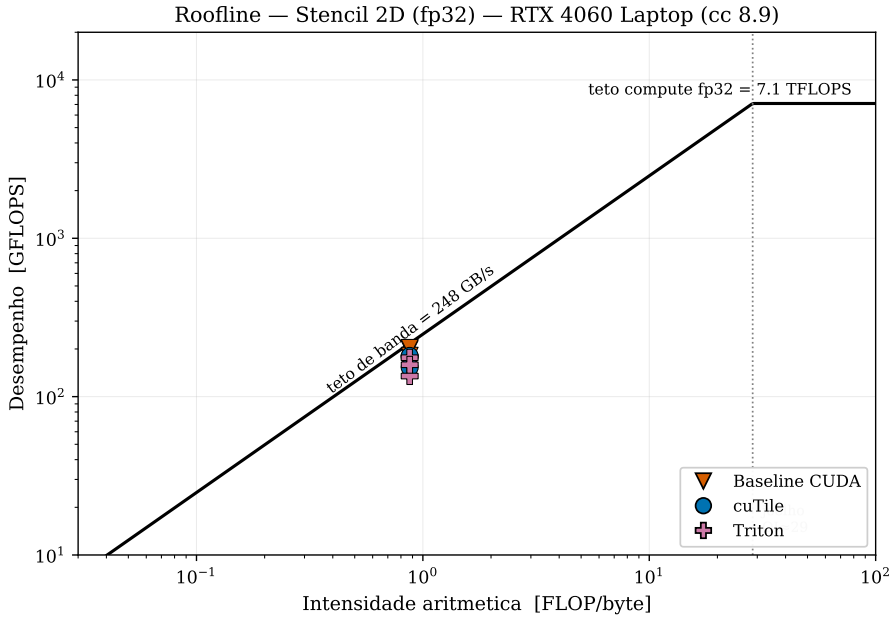

Figura 7: *Roofline* do stencil (fp32) na RTX 4060 Laptop (cc 8.9). Os três *backends* ficam sobre a rampa de banda, na intensidade aritmética ≈0,87 FLOP/byte (o meio do *roofline*), com o *baseline* acima das abstrações: o oposto do que se viu no GEMM.

A Figura 8 mostra a largura de banda efetiva em função do tamanho, com barras de erro entre processos. A ordenação *baseline* > cuTile > Triton é estável nas cinco execuções e em todos os tamanhos.

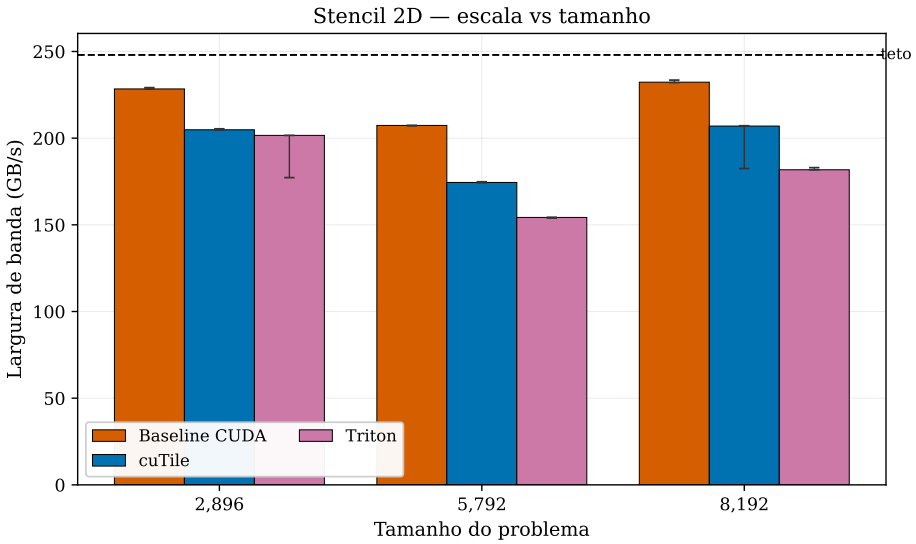

Figura 8: Escala do stencil: largura de banda efetiva *versus* tamanho do problema. A ordenação *baseline* > cuTile > Triton é estável; as barras de erro refletem a dispersão entre processos.

A Tabela 9 consolida desempenho e esforço, e a Figura 9 traz o gráfico de *desempenho* × *esforço* no tamanho limpo de 5792^2 (o ponto *dram-bound* sem ambiguidade térmica). A leitura é o oposto da do GEMM: aqui o canto "alto desempenho, pouco código" está *vazio*. O *baseline* entrega o melhor desempenho (≈83 %) ao maior custo de código (20 LOC, 1 primitiva); o Triton, o menor código (14 LOC, 0 primitivas), tem o pior desempenho (≈62 %); e o cuTile fica no meio dos dois eixos (17 LOC, ≈70 %). Trocar código por simplicidade custou desempenho.

Tabela 9: Desempenho e esforco no padrao stencil 2D (fp32, equacao do calor). Padrao memory-bound com reuso de vizinhanca: o baseline manual (staging em shared memory) vence as abstracoes idiomaticas, que trocam 15–25% de desempenho por codigo bem mais simples.

Modelo	*N*	Tempo (ms)	Banda (GB/s)	Efic. (%)	LOC	Sync
Baseline CUDA	2896	0.294	228.4	91.5	20	1
Baseline CUDA	5792	1.294	207.3	83.1	20	1
Baseline CUDA	8192	2.311	232.3	93.1	20	1
cuTile	2896	0.328	204.8	82.1	17	0
cuTile	5792	1.538	174.5	69.9	17	0
cuTile	8192	2.594	207.0	82.9	17	0
Triton	2896	0.333	201.6	80.8	14	0
Triton	5792	1.740	154.3	61.8	14	0
Triton	8192	2.954	181.8	72.8	14	0

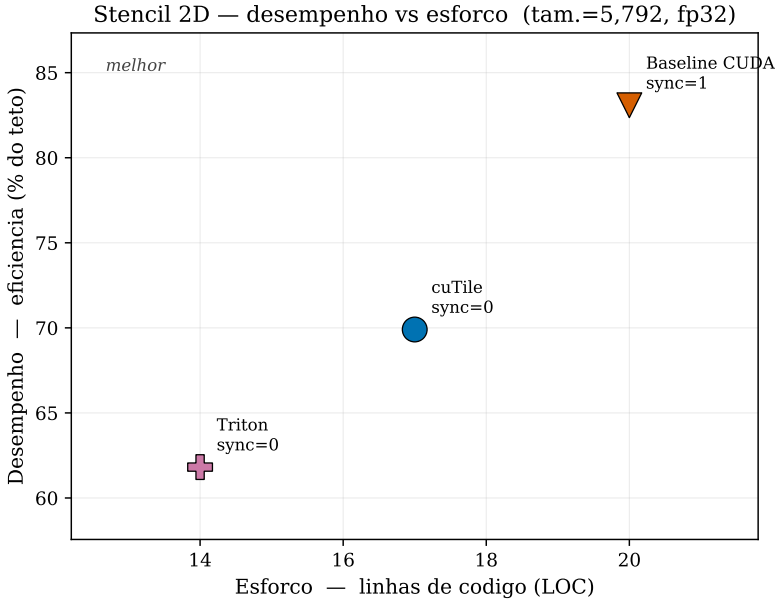

Figura 9: Stencil — desempenho *versus* esforço em $N = 5792^2$ (fp32). Ao contrário do GEMM, não há *backend* no canto "alto desempenho, pouco código": o *baseline* lidera em desempenho ao custo de mais código, e as abstrações trocam desempenho por concisão.

O achado central do stencil é que **aqui a abstração perde para o esforço manual**, o inverso do GEMM. Na redução os três empataram no teto; no GEMM as abstrações superaram o *baseline* manual; no stencil, no ponto *dram* limpo, o *baseline* vence (≈83 %) o cuTile (≈70 %) e o Triton (≈62 %). A causa é o **reúso de vizinhança via memória compartilhada**: o *baseline* faz *staging* explícito do *tile* com *halo* na memória compartilhada (Trecho 3), pagando uma leitura da memória global por ponto e servindo as cinco leituras por ponto *on-chip*; as abstrações, na forma idiomática, fazem acessos deslocados (Trechos 4 e 5) e dependem da *cache* para o reúso, o que é menos eficiente. Em padrões *memory-bound* com forte reúso, o controle manual da memória compartilhada ainda é onde está o desempenho.

O contraste de *código*, porém, continua a favor da abstração: cuTile 17 LOC e Triton 14 LOC, ambos com zero primitivas de sincronização, contra 20 LOC e uma primitiva do *baseline*, que carrega toda a maquinaria de *halo*. O *trade-off* do stencil é, assim, diferente do GEMM: código mais simples, ao preço de ≈15–25 % menos desempenho.

Uma decisão de escopo. Reportam-se deliberadamente as versões *idiomáticas* das abstrações, com acesso deslocado, sem *staging* manual em memória compartilhada. Fazer o *staging* manual no cuTile ou no Triton derrotaria o propósito dessas ferramentas (esconder a memória); a versão idiomática é a mais representativa do que elas propõem, e o fato de ficar abaixo do *baseline* é o resultado honesto. Registra-se que o *gap* poderia ser reduzido com *staging* manual, ao custo justamente da simplicidade que motiva o uso dessas abstrações.

10 Quarto Padrão Diferenciador: *Attention* e a Fusão de *Kernels*

Os três padrões anteriores comparam *como* expressar um único *kernel*. A atenção por produto escalar escalonado, $O = \mathrm{softmax}(QK^\top/\sqrt{D})V$ [11], introduz um eixo que nenhum deles testava: *quantos kernels*. A matriz intermediária S ($M \times N$) pode ser materializada na memória global (três *kernels*) ou nunca chegar a existir (um *kernel* fundido, no estilo *FlashAttention*). É o padrão com

mais *backends* e o que produz o contraste de desempenho mais extremo de todo o trabalho.

10.1 Motivação e conceitos

A atenção foi escolhida como padrão final por três razões. A primeira é o **eixo novo da fusão**, descrito acima. A segunda é a **relevância**: é a operação central dos *transformers* e o alvo de otimização mais estudado em GPU nos últimos anos. A terceira é a **disponibilidade de material**: implementações CUDA de referência foram cedidas por um colega do orientador [21] para estudo dos baselines manuais, e tanto o cuTile quanto o Triton publicam *kernels* FMHA de exemplo, o que permitiu comparar implementações *idiomáticas de cada projeto*, em vez de reconstruções próprias.

O problema que a fusão resolve é o crescimento de S: ela cresce com o quadrado do comprimento da sequência, enquanto as entradas e a saída crescem linearmente.

Dois conceitos sustentam a solução. O *softmax online* mantém o máximo m e a soma l correntes [10], corrigindo retroativamente por $\exp(m_{\text{antigo}} - m_{\text{novo}})$ quando chega um bloco com máximo maior; é um algoritmo *exato*, não uma aproximação, e permite normalizar sem ver a linha inteira. O *FlashAttention* combina o *softmax* online com *tiling* e consciência de I/O para eliminar a materialização de S [9]; FMHA (*Fused Multi-Head Attention*) é o nome genérico do *kernel* que implementa isso.

A fusão não é automática. Cabe uma ressalva conceitual que orienta a leitura de toda a seção: nem o cuTile nem o Triton fundem coisa alguma sozinhos. Os *kernels* FMHA de ambos são implementações *escritas à mão* do FlashAttention. O que a abstração oferece é tornar o algoritmo *expressável*: *tiles* como valores de primeira classe, redução ao longo de um eixo do *tile* em uma linha, sem gerenciar memória compartilhada nem *layouts* de fragmento. Esse enquadramento é mais razoável, e mais interessante, do que afirmar que "a abstração otimiza sozinha".

10.2 Decisões de desenho

A **precisão** é bf16 com acumulação em fp32. A decisão inicial havia sido fp32 (na esperança de não alterar o código cedido, tendo assim um baseline otimizado externo como ponto de análise) mas foi revertida com base em evidência: o *kernel* FMHA do cuTile só é exercitado em meia precisão, e aqui a restrição mais impeditiva, os *backends* `CUDNN_ATTENTION` e `FLASH_ATTENTION` do `scaled_dot_product_attention` *exigem* meia precisão. Em fp32 o *torch* recairia no *backend* "math" (não-fundido), destruindo o teto *vendor*. O bf16 ainda reaproveita a régua já calibrada no GEMM.

A configuração é **multi-cabeça,** $B = 1$ **e** $H = 8$. Começou-se com cabeça única ($B = H = 1$) e os números ficaram inutilizáveis: apenas 8–32 blocos numa GPU de 24 SMs, com eficiências de 18–58 % causadas por *subocupação*, não por qualidade de *kernel*. Com $H = 8$ são 64–256 blocos, e as eficiências subiram para 71–87 %. O valor $H = 8$ também mantém o S do *baseline* dentro da memória disponível. Adotou-se $D = 64$ (dimensão de cabeça típica) e $L \in \{1024, 2048, 4096\}$, no caso não-causal, sem GQA e sem *KV-cache*, pois os *kernels* manuais não implementam nada disso, e os fundidos rodam na mesma configuração mínima. A camada de memória é o *torch* (bf16 nativo), com os *backends* manuais acessando os tensores pela ponte *zero-copy* reaproveitada do GEMM.

O modelo de custo é flops $= 4BHMND$ (as duas GEMMs), *idêntico para todos os* backends, fundidos ou não. Por isso, os GFLOPS são a métrica de comparação justa, e a vantagem da fusão aparece naturalmente como *throughput* maior. Os *bytes* do modelo, $2BH(2MD + 2ND)$,

contam Q, K, V e O em bf16, isto é, o *mínimo fundido*; o tráfego adicional dos não-fundidos é quantificado na análise (Seção 10.4), não no modelo de custo. A intensidade aritmética resultante (512–2048 FLOP/byte) é firmemente *compute-bound*, e a régua é o mesmo teto bf16 de *Tensor Core* do GEMM ($\approx$30 TFLOPS). A eficiência significa, portanto, "fração da capacidade de *matmul* bf16 da máquina", e fica abaixo de 100 % em parte porque o *softmax* entre as duas MMAs não é trabalho de *matmul*. Esse custo é justamente parte do que o padrão mede.

10.3 Os cinco *backends*

O desenho experimental usa cinco implementações, resumidas na Tabela 10. A peça-chave é o `baseline_wmma`: ele usa o *mesmo hardware* das abstrações (*Tensor Cores*), de modo que a distância entre ele e os fundidos é atribuível à **estrutura** (a fusão), e não a unidades de execução diferentes.

Tabela 10: Os cinco *backends* do padrão attention e seus papéis.

Backend	**Estrutura**	***Hardware***	**Papel**
`baseline_simt`	3 *kernels*, materializa S	CUDA cores	piso "primeira tentativa"
`baseline_wmma`	3 *kernels*, materializa S	*Tensor Cores*	**isola a fusão**
cuTile	1 *kernel* fundido (FMHA)	*Tensor Cores*	abstração
Triton	1 *kernel* fundido (FlashAttention)	*Tensor Cores*	abstração
cuDNN	1 *kernel* fundido (SDPA)	*Tensor Cores*	teto *vendor*

O `baseline_simt` foi adaptado da implementação *naive* cedida para bf16; S permanece em fp32 (é intermediário), o que deixou o *kernel* de *softmax* intocado, só os dois *matmuls* mudaram. Seu *softmax* usa uma *thread* por linha, em três varreduras seriais. O `baseline_wmma`, por sua vez, foi escrito do **zero**, em vez de portar o SGEMM fp32 cedido, justamente para eliminar o confundimento de *hardware*; segue o mesmo idioma WMMA do GEMM (um *warp* por *tile* 16×16, fragmentos direto da memória global, sem *staging*) e usa *softmax* online paralelo. Um detalhe merece registro: $QK^\top$ não exige transpor K fisicamente: carregar K (armazenado (N, D) *row-major*) como fragmento `matrix_b` em *col_major* com $ld = D$ já entrega o *tile* de $K^\top$. É o equivalente manual do parâmetro `order` do `ct.load` do cuTile: os dois modelos resolvem a transposição sem mover dados.

Os dois *backends* de abstração usam os *kernels* de referência dos próprios projetos. O do cuTile é o FMHA da NVIDIA, copiado literalmente [20]; o arquivo mantém duas variantes, a **geral** (com causal, GQA, *KV-cache* e *padding* irregular, 72 LOC) e uma **especializada** para o caso deste estudo (35 LOC), que é a usada nas métricas. Como as *flags* são constantes de compilação, o compilador elimina os ramos mortos de qualquer forma e o desempenho é o mesmo; medir a versão geral, porém, compararia um *kernel* de produção totalmente genérico contra *kernels* manuais mínimos. O delta $35 \rightarrow 72$ LOC fica registrado como *o custo da generalidade*. O do Triton é o *kernel forward* do FlashAttention da própria distribuição [19], com duas edições documentadas: apenas o *forward* (o original traz uma `autograd.Function` completa, enquanto aqui o *kernel* é lançado diretamente, como em todos os outros *backends*) e a lista de parâmetros reformatada para densidade convencional (o original escreve um argumento por linha, o que inflaria artificialmente o LOC). Cada projeto foi medido com os *tiles* que ele próprio recomenda.

Contagem de argumentos: uma métrica independente de formatação. Como o LOC é sensível a estilo de escrita, o padrão attention permite uma segunda métrica de esforço, imune a esse efeito:

o número de argumentos do *kernel*. O do cuTile recebe **9** argumentos (objetos tensor, lendo formas e *strides* internamente); o do Triton recebe **30**, sendo 16 deles *strides* explícitos, além de ponteiros crus. Essa diferença é real, não de formatação, e captura o mesmo espectro de abstração que o GEMM já havia mostrado: o cuTile abstrai mais, o Triton expõe mais controle.

10.4 Resultados

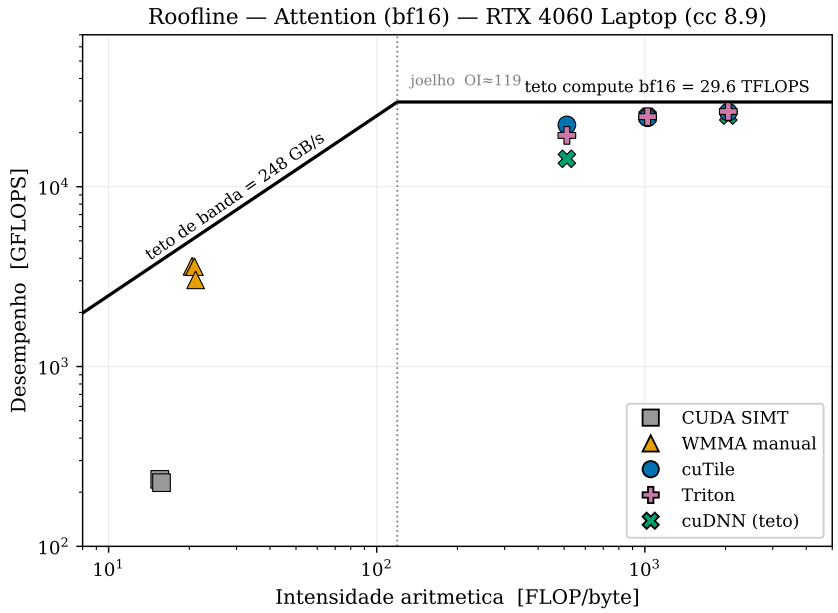

Figura 10: *Roofline* do attention (bf16). Os *backends* não-fundidos são posicionados pela intensidade **efetiva**: o tráfego que a estrutura obriga, incluindo escrever e reler S (e P), e caem à esquerda do joelho, na região limitada por banda; os fundidos permanecem na intensidade algorítmica (512–2048), na região limitada por compute. A tabela reporta a intensidade algorítmica, igual para todos: são grandezas distintas, não uma inconsistência.

A Figura 10 exige uma explicação metodológica. Os CSVs armazenam a intensidade *algorítmica* (flops/model_bytes), adequada para redução, GEMM e stencil, onde a estrutura da implementação não altera o tráfego. No attention, não: um *pipeline* não-fundido materializa S e move cerca de cem vezes mais *bytes* que o fundido. Plotar ambos na mesma posição horizontal colocaria um *kernel memory-bound* dentro da região *compute-bound*, contradizendo o próprio propósito do *roofline*, que é prever qual limite se aplica. Por isso os não-fundidos são posicionados pela **intensidade efetiva**, calculada com o tráfego que a estrutura obriga (Q, K, V, O, mais escrever e ler S, mais escrever e ler P), contando cada *buffer* na sua precisão real. A escolha é defensável por três propriedades: é propriedade da *estrutura*, não da qualidade do código; ignora acertos de *cache*, sendo portanto um limite inferior do tráfego real e, logo, um limite *superior* da intensidade, ou seja, conservadora a favor dos não-fundidos; e é aplicada num único ponto do *pipeline* de figuras, com os fundidos mantendo a intensidade algorítmica inalterada. O resultado é nítido: o `baseline_wmma` cai de 2048 para ≈21 e o `baseline_simt` para ≈16, ambos bem à esquerda do joelho (≈120), enquanto os fundidos permanecem à direita. A figura passa a mostrar visualmente que o não-fundido é limitado por banda e o fundido por *compute*, que é precisamente a tese do FlashAttention.

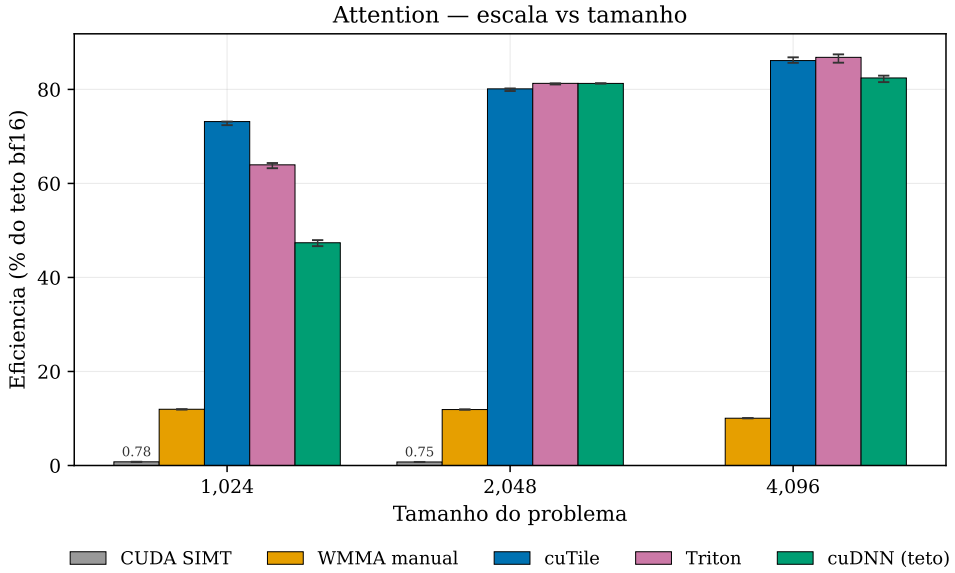

Figura 11: Escala do attention: eficiência (% do teto bf16) *versus* comprimento da sequência, com barras de erro inter-processo. Em $L = 1024$ os três *backends* fundidos estão em regime *launch-bound* e a figura mede parcialmente custo de *host*; afirmações sobre qualidade de *kernel* devem se apoiar em $L \geq 2048$.

A Tabela 11 consolida os resultados. Em $L = 2048$, onde os cinco *backends* rodam, os fatores isolam-se de forma limpa: `baseline_simt` → `baseline_wmma` rende ≈15,9× (*Tensor Cores* mais *softmax* paralelo); `baseline_wmma` → fundidos rende ≈6,7× (a fusão, com o *mesmo hardware*); e o composto, `baseline_simt` → fundidos, chega a ≈107×. Em $L = 4096$ o fator da fusão sobe para ≈8,6×: cresce com o tamanho, como esperado, já que S cresce com o quadrado.

Tabela 11: Desempenho e esforco no padrao de atencao (bf16, B=1, H=8, D=64, nao-causal). Os dois baselines materializam a matriz S ($M \times N$) e usam tres kernels; cuTile, Triton e cuDNN sao fundidos (FlashAttention). Como `baseline_wmma` usa os mesmos Tensor Cores das abstracoes, a distancia entre ele e os fundidos isola o efeito da fusao. LOC do cuTile refere-se a variante especializada (a versao geral, com mascaramento causal, GQA e KV-cache, custa 72).

Modelo	L	Tempo (ms)	TFLOPS	Efic. (%)	LOC	Sync
CUDA SIMT	1,024	9.113	0.2	0.8	43	0
CUDA SIMT	2,048	38.007	0.2	0.7	43	0
WMMA manual	1,024	0.595	3.6	12.0	92	17
WMMA manual	2,048	2.392	3.6	11.9	92	17
WMMA manual	4,096	11.323	3.0	10.1	92	17
cuTile	1,024	0.097	22.1	73.2	35	0
cuTile	2,048	0.355	24.2	80.1	35	0
cuTile	4,096	1.322	26.0	86.1	35	0
Triton	1,024	0.111	19.3	64.0	56	0
Triton	2,048	0.350	24.5	81.3	56	0
Triton	4,096	1.312	26.2	86.8	56	0
cuDNN (teto)	1,024	0.150	14.3	47.4	–	–
cuDNN (teto)	2,048	0.350	24.5	81.3	–	–
cuDNN (teto)	4,096	1.381	24.9	82.4	–	–

O quadrante (Figura 12) é o mais informativo dos quatro padrões justamente porque os pontos se espalham pelos quatro cantos. O `baseline_wmma` ocupa o canto ruim: 92 LOC, 17 primitivas de sincronização e apenas ≈12 % do teto, enquanto cuTile e Triton ficam no canto bom, com 35 e 56 linhas, zero primitivas e ≈80 %. O `baseline_simt`, com 43 linhas, entrega 0,75 %.

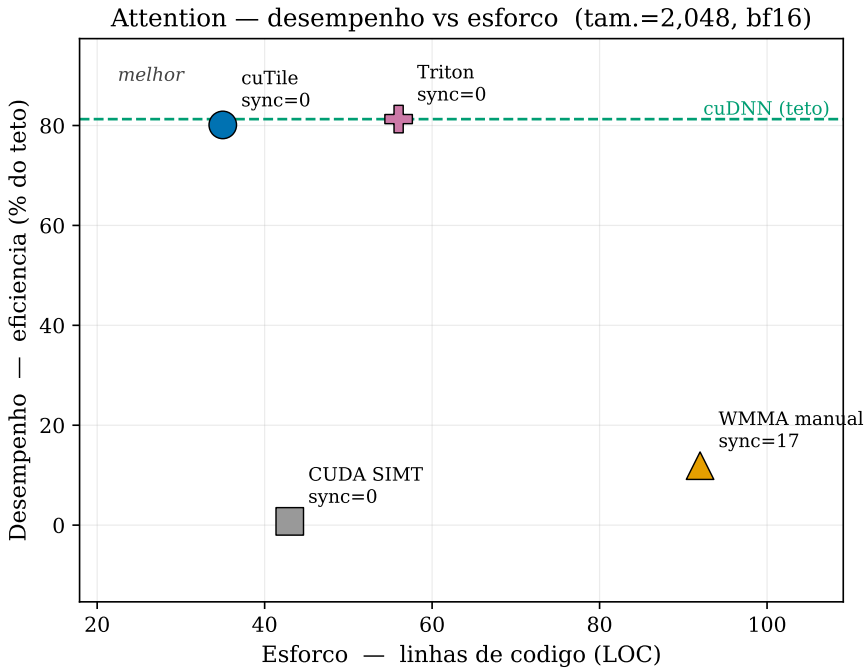

Figura 12: Attention — desempenho *versus* esforço em $L = 2048$ (bf16), o único tamanho em que os cinco *backends* rodam. É o quadrante mais informativo do estudo: os pontos ocupam os quatro cantos, inclusive o ruim (muito código *e* pouco desempenho), onde está o WMMA manual. O teto *vendor* aparece como linha horizontal, por ser chamada de biblioteca e não ter LOC.

Decomposição por *kernel*: evidência direta da fusão. Os *pipelines* não-fundidos foram cronometrados estágio a estágio, com eventos registrados *entre* os *kernels* (apenas marcadores, sem sincronização inserida no meio), numa passagem separada da medição principal para que os números de manchete continuassem comparáveis. A Figura 13 mostra o resultado, e ela refutou duas hipóteses iniciais, o que vale registrar como demonstração de rigor. Esperava-se que o *softmax* dominasse o `baseline_wmma`: ele é o maior estágio, mas fica em ≈34 %, com os três dividindo-se quase em terços. O custo da materialização não está concentrado no *softmax*, está *espalhado*, porque o $QK^\top$ paga por escrever S e o PV paga por reler P. Não há um estágio vilão: a estrutura não-fundida é que é o problema. E supunha-se que o `baseline_simt` fosse "dominado pelo *softmax* serial", o que é falso: o $QK^\top$ consome ≈64 % e o *softmax* apenas ≈21 %. A causa ali é o acesso não coalescido do *matmul* ingênuo. Isso reforça que o `baseline_simt` *não* deve ser usado para argumentar sobre fusão: ele mistura três efeitos distintos.

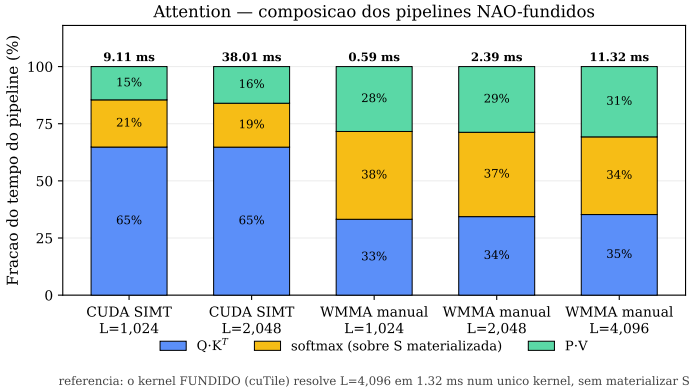

Figura 13: Composição do tempo dos *pipelines* não-fundidos (barras 100 % empilhadas, tempo absoluto anotado no topo). As proporções vêm da passagem de decomposição; o total absoluto vem da medição de manchete (mais iterações e mediana inter-processo), combinação deliberada de duas medições, cada uma usada para o que mede bem. No WMMA os três estágios convergem para terços: nenhum estágio é o gargalo.

O número mais forte do padrão sintetiza o argumento: o estágio de *softmax* realiza cerca de sete operações por elemento sobre $M \times N$ elementos, contra $M \times N \times D$ multiplicações-acumulações das duas GEMMs: **menos de 3 % do trabalho aritmético total**, mas consome ≈**34 % do tempo**. Ele existe apenas porque S foi materializada, e está no limite físico: em $L = 4096$ move ≈805 MB em ≈3,5 ms, ou ≈227 GB/s, colado no pico de cópia calibrado. Não há otimização possível ali, a única saída é não materializar.

Um achado metodológico: $L = 1024$ é *launch-bound*. Comparar a mediana com o *mínimo* de cada distribuição revela um efeito que a mediana sozinha esconderia. Em $L = 1024$, Triton e cuDNN têm distribuições assimétricas (mediana 20–25 % acima do mínimo), enquanto o cuTile é justo (≈1 %); em $L \geq 2048$ a assimetria desaparece nos três (razões de 1,00–1,01). Essa é a assinatura de *custo de host* (lançamento, despacho, alocação), não de variância de GPU: com *kernels* de ≈100 μs a CPU não consegue enfileirar trabalho à frente e o intervalo entre eventos inclui tempo ocioso da GPU; com *kernels* de ≈350 μs o trabalho de CPU fica escondido pelo *pipeline* assíncrono. As causas prováveis são identificáveis: o `scaled_dot_product_attention` não aceita `out=` e portanto aloca o tensor de saída a cada chamada, além de passar pelo *dispatcher* e pela seleção heurística de *backend*; o *launcher* Python do Triton processa argumentos e consulta o *cache* a cada lançamento; o `ct.launch` do cuTile, com *grid* pré-computado, mostrou-se o mais enxuto. A consequência é decisiva: pelo mínimo, a ordenação em $L = 1024$ é Triton > cuTile > cuDNN, *invertendo* a ordem entre Triton e cuTile em relação à mediana. Foi reportada a mediana por consistência com todos os padrões, mas fica o comentário sobre $L = 1024$ como regime *launch-bound*, e qualquer afirmação sobre qualidade de *kernel* apoia-se em $L \geq 2048$.

10.5 Conclusões do padrão

A fusão domina o restante. Os três *backends* fundidos ficam a menos de 6 % um do outro em $L = 2048$ e todos são ≈7–9× mais rápidos que a versão não-fundida com o *mesmo hardware*. A escolha *entre* cuTile, Triton e a biblioteca do fabricante importa muito menos do que a decisão de **fundir**.

A abstração torna a fusão acessível. Fundir à mão em WMMA esbarra num obstáculo concreto: o FlashAttention precisa de uma redução *por linha* sobre o acumulador e de um *rescaling*

por linha, mas a API `wmma` esconde deliberadamente o *layout* do fragmento acumulador: não há garantia de qual linha ou coluna corresponde a cada elemento. As saídas seriam despejar o acumulador em memória compartilhada a cada bloco ou descer para `mma.sync` em PTX (o caminho do CUTLASS [16]). No cuTile, a mesma redução é uma chamada de uma linha (`ct.max` ao longo de um eixo do *tile*). A abstração não é mais rápida por mágica: ela expõe a operação certa no nível certo. Esse é, possivelmente, o argumento mais forte de todo o trabalho a favor das abstrações, e ele é sobre *expressividade*, não sobre otimização automática.

O espectro de abstração confirma-se num quarto padrão, agora com duas métricas independentes: cuTile mais conciso (35 LOC, 9 argumentos), Triton mais verboso e com mais controle (56 LOC, 30 argumentos), ambos com zero primitivas de sincronização.

O *sync* $= 0$ do `baseline_simt` é significativo, não um defeito de contagem. Sem memória compartilhada, ele não sincroniza nada: paga a cooperação inteiramente em tráfego de memória global. É um contraste conceitual interessante com a redução e o stencil, onde o *baseline* sincroniza justamente *para evitar* tráfego.

O teto *vendor* não é intocável. Em $L = 4096$, cuTile e Triton ficam ≈4–5 pontos percentuais à frente do cuDNN, e a diferença persiste quando se compara pelos mínimos, não sendo portanto artefato de *overhead*. É o mesmo fenômeno já observado no GEMM com o Triton: heurísticas de biblioteca *vendor*, calibradas sobretudo para GPUs de *datacenter*, não escolhem necessariamente o melhor *kernel* numa GPU de consumo com $B \cdot H$ pequeno. O enquadramento mais honesto é "iguala ou supera marginalmente nesta configuração e nesta GPU", nunca uma afirmação geral.

Nota de ambiente. O Triton emite um aviso de depreciação (`tl.make_block_ptr`) apenas na primeira execução, durante a compilação; a partir da segunda o *kernel* vem do *cache* JIT em disco. Optou-se por *não* migrar para a API sucessora: ela é construída sobre TMA (*hardware* Hopper, sm_90+), não traria benefício na arquitetura Ada, e migrar significaria medir uma reescrita própria em vez da implementação de referência.

11 Síntese Comparativa

Com quatro padrões diferenciadores fechados, o espectro da tese fica completo, e a mensagem central é que *o valor da abstração depende do padrão* (Tabela 12). Na redução, *memory-bound* sem reúso, os três modelos empatam no teto de banda e todo o ganho das abstrações é de produtividade. No GEMM, *compute-bound*, as abstrações superam o esforço manual idiomático em desempenho *e* produtividade. No stencil, *memory-bound* com forte reúso, o controle manual da memória compartilhada vence as abstrações idiomáticas. E no attention, o ganho é o maior de todos, mas por uma razão diferente das anteriores: não é a qualidade do *kernel*, e sim a **estrutura** que a abstração torna expressável.

Tabela 12: O espectro completo da tese: o valor da abstração depende do padrão.

Padrão	Natureza	Resultado de desempenho
Redução	*memory-bound* (sem reúso)	os três *empatam* no teto de banda
GEMM	*compute-bound* (*Tensor Cores*)	abstrações *superam* o manual idiomático
Stencil	*memory-bound* (com reúso)	o manual *vence* as abstrações
Attention	*compute-bound* (fusão)	abstrações *dominam* (≈7×), por estrutura

A Figura 14 condensa o trabalho inteiro: para cada padrão, a razão entre o tempo de cada abstração e o do *melhor baseline* manual *daquele mesmo padrão*. A abstração vai de *ligeiramente*

pior (0,74× no stencil) a *quase sete vezes melhor* (6,8× no attention), e cada ponto dessa variação tem explicação mecânica: sem reúso a gerenciar, empate; com *hardware* especializado, ganho; com reúso que o controle manual explora melhor, perda; com fusão, o ganho maior de todos.

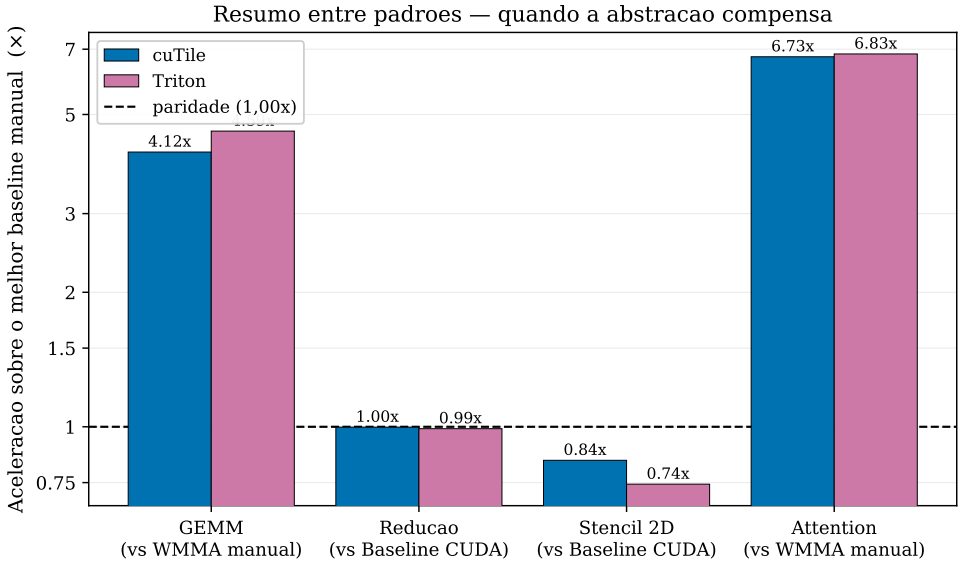

Figura 14: Aceleração das abstrações sobre o melhor *baseline* manual de cada padrão (razão de tempos de *kernel*; escala logarítmica; linha de paridade em 1,0). A referência é o *baseline* manual mais rápido do padrão, indicada no eixo horizontal; o teto *vendor* é excluído de propósito, por ser teto e não esforço de programação. O tamanho de comparação difere por padrão (GEMM 4096, attention 2048, redução $67{,}1\times10^6$, stencil 5792^2).

A leitura unificada que fica é: a abstração é pura vantagem quando não há reúso a gerenciar (redução), quando o *hardware* especializado domina (GEMM), ou quando ela torna expressável uma *estrutura* melhor (attention); mas paga um preço de desempenho quando o gargalo é um reúso de memória que o controle manual explora melhor (stencil). Em *todos* os casos, porém, a abstração reduz, levemente ou de forma considerável, o código e a sincronização explícita.

Referências

[1] NVIDIA CORPORATION. *CUDA C++ Programming Guide*. Disponível em: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/`. Acesso em: 2026.

[2] NVIDIA CORPORATION. *cuTile Python — Quickstart e Documentação*. Disponível em: `https://docs.nvidia.com/cuda/cutile-python/`. Acesso em: 2026.

[3] TILLET, P.; KUNG, H. T.; COX, D. *Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations*. In: ACM SIGPLAN MAPL, 2019.

[4] OKUTA, R. et al. *CuPy: A NumPy-Compatible Library for NVIDIA GPU Calculations*. In: NeurIPS Workshop on Machine Learning Systems, 2017. Disponível em: `https://cupy.dev/`.

[5] WILLIAMS, S.; WATERMAN, A.; PATTERSON, D. *Roofline: An Insightful Visual Performance Model for Multicore Architectures*. Communications of the ACM, v. 52, n. 4, p. 65–76, 2009.

[6] KIRK, D. B.; HWU, W. W. *Programming Massively Parallel Processors: A Hands-on Approach*. 4. ed. Morgan Kaufmann, 2022.

[7] CHENG, J.; GROSSMAN, M.; McKERCHER, T. *Professional CUDA C Programming*. Wrox Press, 2014.

[8] SANDERS, J.; KANDROT, E. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 2010.

[9] DAO, T.; FU, D. Y.; ERMON, S.; RUDRA, A.; RÉ, C. *FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness*. In: Advances in Neural Information Processing Systems (NeurIPS), 2022.

[10] MILAKOV, M.; GIMELSHEIN, N. *Online Normalizer Calculation for Softmax*. arXiv:1805.02867, 2018.

[11] VASWANI, A. et al. *Attention Is All You Need*. In: Advances in Neural Information Processing Systems (NeurIPS), 2017.

[12] PASZKE, A. et al. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. In: Advances in Neural Information Processing Systems (NeurIPS), 2019. Disponível em: `https://pytorch.org/`.

[13] NVIDIA CORPORATION. *cuBLAS Library — User Guide*. Disponível em: `https://docs.nvidia.com/cuda/cublas/`. Acesso em: 2026.

[14] NVIDIA CORPORATION. *cuDNN — Developer Guide*. Disponível em: `https://docs.nvidia.com/deeplearning/cudnn/`. Acesso em: 2026.

[15] NVIDIA CORPORATION. *CUDA Tile IR Documentation*. Disponível em: `https://docs.nvidia.com/cuda/tile-ir/latest/`. Acesso em: 2026.

[16] NVIDIA CORPORATION. *CUTLASS: CUDA Templates for Linear Algebra Subroutines*. Repositório GitHub. Disponível em: `https://github.com/NVIDIA/cutlass`. Acesso em: 2026.

[17] NVIDIA CORPORATION. *NVIDIA CUDA 13.1 Powers Next-Gen GPU Programming with NVIDIA CUDA Tile and Performance Gains*. NVIDIA Technical Blog, dez. 2025. Disponível em: `https://developer.nvidia.com/blog/nvidia-cuda-13-1-powers-next-gen-gpu-programming-with-nvidia-cuda-tile-and-performance-gains/`. Acesso em: 2026.

[18] NVIDIA CORPORATION. *Advancing GPU Programming with the CUDA Tile IR Backend for OpenAI Triton*. NVIDIA Technical Blog, jan. 2026. Disponível em: `https://developer.nvidia.com/blog/advancing-gpu-programming-with-the-cuda-tile-ir-backend-for-openai-triton/`. Acesso em: 2026.

[19] OPENAI. *Triton: Development Repository for the Triton Language and Compiler*. Repositório GitHub. Disponível em: `https://github.com/triton-lang/triton`. Acesso em: 2026.

[20] NVIDIA CORPORATION. *cuTile Python*. Repositório GitHub. Disponível em: `https://github.com/NVIDIA/cutile-python`. Acesso em: 2026.

[21] CLÉTO, Jhonatan. *attention-bench: implementações de atenção em CPU, OpenMP e CUDA*. Repositório GitHub, 2025. Disponível em: `https://github.com/cl3to/attention-bench`. Acesso em: 2026.