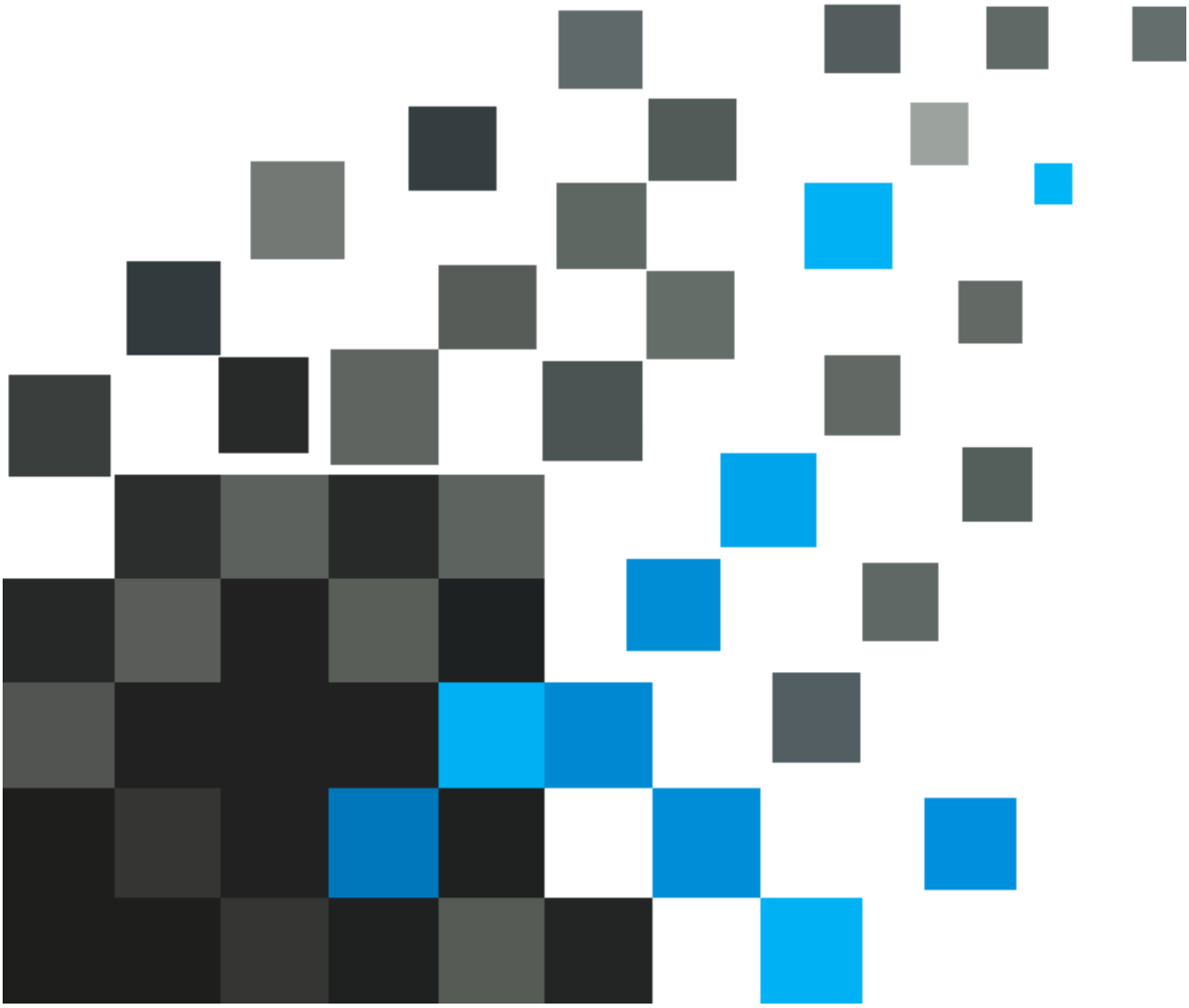
unesp

UNIVERSIDADE ESTADUAL PAULISTA

PROGRAMA DE PÓS-GRADUAÇÃO
EM CIÊNCIA DA COMPUTAÇÃO



ESTRATIFICAÇÃO DE MEMÓRIA PARA INFERÊNCIA DE GRANDES MODELOS DE LINGUAGEM EM AMBIENTES LIMITADOS A CPU

LUÍSA LEONA CATTAI

INSTITUTO DE GEOCIÊNCIAS E CIÊNCIAS EXATAS
RIO CLARO

UNIVERSIDADE ESTADUAL PAULISTA
"Júlio de Mesquita Filho"
Instituto de Geociências e Ciências Exatas
Câmpus de Rio Claro

LUÍSA LEONA CATTAI

ESTRATIFICAÇÃO DE MEMÓRIA PARA INFERÊNCIA DE GRANDES MODELOS DE LINGUAGEM EM AMBIENTES LIMITADOS A CPU

Dissertação de Mestrado apresentada ao Instituto de Geociências e Ciências Exatas do Câmpus de Rio Claro, da Universidade Estadual Paulista "Júlio de Mesquita Filho", como parte dos requisitos para obtenção do título de Mestre em Ciência da Computação.

Orientador: Prof. Dr. Alexandro José Baldassin

Rio Claro - SP
2026

C368e

Cattai, Luísa Leona
Estratificação de memória para inferência de grandes modelos de linguagem em ambientes limitados a CPU / Luísa Leona Cattai. -- Rio Claro, 2026
83 p. : il., tabs.

Dissertação (mestrado) - Universidade Estadual Paulista (UNESP), Instituto de Geociências e Ciências Exatas, Rio Claro
Orientador: Alexandro José Baldassin

1. Ciência da computação. 2. Sistemas de computação. 3. Computação de alto desempenho. 4. Inteligência artificial. I. Título.

Sistema de geração automática de fichas catalográficas da Unesp. Dados fornecidos pelo autor(a).

IMPACTO POTENCIAL DESTA PESQUISA

Impacto Potencial desta Pesquisa:
Esta pesquisa viabiliza a execução de Grandes Modelos de Linguagem em CPUs de baixo custo via estratificação de memória, reduzindo em até 4 GB o consumo de DRAM sem comprometer a interatividade. O trabalho democratiza a IA generativa em infraestruturas computacionais limitadas.

Potential Impact of this Research:
This research enables running Large Language Models on low-cost CPUs through memory tiering, freeing up to 4 GB of DRAM without compromising interactive performance. The work democratizes generative AI deployment on resource-constrained computing infrastructures.

UNIVERSIDADE ESTADUAL PAULISTA
"Júlio de Mesquita Filho"
Instituto de Geociências e Ciências Exatas
Câmpus de Rio Claro

LUÍSA LEONA CATTAI

ESTRATIFICAÇÃO DE MEMÓRIA PARA INFERÊNCIA DE GRANDES MODELOS DE LINGUAGEM EM AMBIENTES LIMITADOS A CPU

Dissertação de Mestrado apresentada ao Instituto de Geociências e Ciências Exatas do Câmpus de Rio Claro, da Universidade Estadual Paulista "Júlio de Mesquita Filho", como parte dos requisitos para obtenção do título de Mestre em Ciência da Computação.

Comissão Examinadora

Prof. Dr. ALEXANDRO JOSÉ BALDASSIN
IGCE / UNESP / Rio Claro (SP)

Prof. Dr. CÉSAR LINCOLN CAVALCANTE MATTOS
UFC / Fortaleza (CE)

Prof. Dr. CAETANO MAZZONI RANIERI
IGCE / UNESP / Rio Claro (SP)

Conceito: Aprovado.

Rio Claro (SP), 12 de agosto de 2026.

Agradecimentos

O presente trabalho foi realizado com apoio da Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP), Brasil. Processo nº 2024/02372-7. As opiniões, hipóteses e conclusões ou recomendações expressas neste material são de responsabilidade do(s) autor(es) e não necessariamente refletem a visão da FAPESP.

Resumo

Grandes Modelos de Linguagem (LLMs) têm sido cada vez mais implantados em ambientes de CPU sem aceleradores, motivados pelo custo elevado e pela disponibilidade restrita de GPUs. Nesse cenário, o crescimento contínuo do tamanho dos modelos e a expansão das janelas de contexto tornam a capacidade de memória DRAM o principal fator limitante de custo e viabilidade para a execução de modelos de grande escala. Contornar essa restrição sem recorrer a hardware especializado é um desafio de crescente relevância prática, sobretudo em infraestruturas econômicas. As arquiteturas de memória heterogênea, que combinam DRAM com camadas de maior capacidade e maior latência, apresentam-se como uma estratégia promissora, porém carecem de uma caracterização sistemática que indique *quais* componentes de um LLM podem ser realocados para a memória lenta sem comprometer o desempenho. Diante dessa lacuna, esta dissertação investiga a alocação seletiva dos componentes internos do modelo (pesos de Atenção, pesos da FFN e KV-Cache) entre a DRAM e uma camada de memória de maior capacidade, sob uma abordagem de estratificação explícita. Para tanto, a metodologia instrumentou o *framework* llama.cpp para mapear seletivamente esses componentes entre a DRAM e uma camada de memória lenta (Intel Optane DC via acesso direto/DAX). A avaliação experimental cobriu 8 configurações de alocação e 5 cenários operacionais (prompts de 64 a 7.000 tokens), utilizando os modelos LLaMA 2 (7B, Multi-Head Attention, MHA) e LLaMA 3 (8B, Grouped-Query Attention, GQA) quantizados em Q4_K_M. Os resultados demonstram que a degradação de desempenho é governada por um único mecanismo: o volume de bytes relidos a cada passo de geração. Em contextos curtos, a alocação da KV-Cache na memória lenta causa impacto negligenciável (apenas $-2,3\%$ na vazão), enquanto mover a FFN reduz o desempenho em até $59,1\%$. Em contextos longos, a sensibilidade se inverte: a KV-Cache sofre perdas de até $-52,7\%$, ao passo que os pesos de Atenção sob arquitetura GQA (LLaMA 3) mostram-se altamente tolerantes, caindo apenas $-5,5\%$ (contra $-21,5\%$ no MHA). Por fim, a formalização do problema como uma otimização bi-objetivo (DRAM liberada *versus* vazão de inferência) comprovou a viabilidade de liberar mais de 4 GB de DRAM mantendo a vazão acima do limite de aceitabilidade interativa ($3,0$ tokens/s), culminando em diretrizes práticas para decisões de alocação conscientes do regime operacional.

Palavras-chave: Grandes Modelos de Linguagem; inferência em CPU; memória heterogênea; estratificação de memória; KV-Cache; Intel Optane.

Abstract

Memory Stratification for Large Language Model Inference in CPU-Constrained Environments

Large Language Models (LLMs) have been increasingly deployed in CPU-only environments without accelerators, driven by the high cost and limited availability of GPUs. In this scenario, the continuous growth in model size and the expansion of context windows make DRAM capacity the main limiting factor for the cost and feasibility of running large-scale models. Overcoming this constraint without resorting to specialized hardware is a challenge of growing practical relevance, particularly in cost-constrained infrastructures. Heterogeneous memory architectures, which combine DRAM with higher-capacity, higher-latency tiers, emerge as a promising strategy; however, they lack a systematic characterization indicating *which* components of an LLM can be offloaded to the slow memory without compromising performance. Motivated by this gap, this dissertation investigates the selective allocation of the model's internal components (Attention weights, FFN weights, and KV-Cache) between DRAM and a higher-capacity memory tier, under an explicit tiering approach. To this end, the methodology instrumented the llama.cpp *framework* to selectively map these components between DRAM and a slow memory tier (Intel Optane DC via direct access/DAX). The experimental evaluation covered 8 allocation configurations and 5 operational scenarios (prompts ranging from 64 to 7,000 tokens), using the LLaMA 2 (7B, Multi-Head Attention) and LLaMA 3 (8B, Grouped-Query Attention) models quantized in Q4_K_M. The results show that performance degradation is governed by a single mechanism: the volume of bytes reread at each generation step. Under short contexts, allocating the KV-Cache to slow memory has a negligible impact (only -2.3% in throughput), whereas moving the FFN reduces performance by up to 59.1%. Under long contexts, the sensitivity reverses: the KV-Cache suffers losses of up to -52.7%, while the Attention weights under the GQA architecture (LLaMA 3) prove highly tolerant, dropping by only -5.5% (compared to -21.5% in MHA). Finally, formalizing the problem as a bi-objective optimization (freed DRAM *versus* inference throughput) demonstrated the feasibility of freeing more than 4 GB of DRAM while keeping throughput above the interactive acceptability threshold (3.0 tokens/s), culminating in practical guidelines for regime-aware allocation decisions.

Keywords: Large Language Models; CPU inference; heterogeneous memory; memory tiering; KV-Cache; Intel Optane.

Lista de ilustrações

Figura 1 – Diagrama da arquitetura dos Transformers. Extraído de Vaswani et al. (2017). 26
Figura 2 – Hierarquia Memória-Armazenamento com camada de Memória Persistente. Extraído de Scargall (2020). 30
Figura 3 – Degradação percentual do *decode* TPS em relação à `Baseline` para as configurações isoladas (`KV`, `Attn`, `FFN`), por cenário. 53
Figura 4 – *Decode* TPS por cenário (C1–C5) para a `Baseline` e as configurações isoladas, em LLaMA 2 e LLaMA 3. 56
Figura 5 – *Prefill* TPS por cenário (C1–C5) para a `Baseline` e as configurações isoladas, em LLaMA 2 e LLaMA 3. 56
Figura 6 – Magnitude da degradação relativa do *decode* TPS (%) das configurações combinadas, por cenário. 62
Figura 7 – Fronteira de Pareto por cenário: LLaMA 2 (MHA, esq.) e LLaMA 3 (GQA, dir.). Pontos cheios e ligados pertencem à fronteira; pontos esmaecidos são dominados. A linha tracejada é o piso de aceitabilidade (3,0 tok/s); configurações abaixo dela são inviáveis. 67
Figura 8 – Mapas de calor da degradação relativa do *decode* TPS (%) por configuração (linhas) e cenário (colunas). Tons mais escuros indicam maior degradação. A comparação entre os painéis evidencia o efeito do GQA sobre o espaço de decisão de alocação: no LLaMA 3 (b), as configurações que preservam a KV-Cache em DRAM exibem células sistematicamente mais claras do que no LLaMA 2 (a), sobretudo sob contexto longo (C1, C4). 69

Lista de tabelas

Tabela 1 – Comparativo de desempenho físico entre DRAM DDR4 e Intel Optane DC PMem em modo *App Direct*, por módulo DIMM individual (YANG et al., 2020). 31
Tabela 2 – Trabalhos de *offloading* com foco em GPU para inferência de LLMs. 36
Tabela 3 – Trabalhos de inferência com apenas CPU para LLMs. 38
Tabela 4 – Trabalhos de memória heterogênea e posicionamento de dados para inferência de LLMs. 39
Tabela 5 – Síntese comparativa de trabalhos representativos. ✓ indica suporte; — indica ausência. 41
Tabela 6 – Comparação arquitetural entre os modelos LLaMA 2 7B e LLaMA 3 8B empregados neste trabalho, ambos no formato de quantização `Q4_K_M`. . . . 44
Tabela 7 – Estratégias de posicionamento usadas nos experimentos. 46
Tabela 8 – Cenários de uso para avaliação da estratificação de memória. 47
Tabela 9 – Vazão média (tokens/s) por cenário e configuração para o modelo LLaMA 2. Valores em média ± desvio-padrão. 52
Tabela 10 – Vazão média (tokens/s) por cenário e configuração para o modelo LLaMA 3. Valores em média ± desvio-padrão. 52
Tabela 11 – Degradação relativa do *decode* TPS (%) das configurações isoladas face à `Baseline`, por cenário e modelo, com os cenários ordenados por contexto efetivo crescente. A componente *mais tolerante* em cada cenário aparece em **negrito**. A hierarquia rotaciona: a KV-Cache migra de mais tolerante (contexto curto) a menos tolerante (contexto longo). 54
Tabela 12 – Penalidade da configuração `KV` sobre o *decode* TPS, por cenário e modelo. Os cenários são ordenados pelo tamanho do contexto (n_prompt + n_gen, conforme Tabela 8). A degradação cresce com o contexto acessado durante a geração, ainda que não estritamente monotônica. 57
Tabela 13 – Magnitude da degradação relativa do *decode* TPS (%) das duas combinações disputantes ao ponto ótimo, por cenário e modelo. A combinação menos penalizadora (*em negrito*) migra de `KV+Attn`, sob contexto curto, para `Attn+FFN`, sob contexto longo. 62
Tabela 14 – Estimativa do tamanho de cada bloco de componentes em memória. Os pesos (Atenção e FFN) são estimados sob `Q4_K_M` (≈4,5 bits/parâmetro); a KV-Cache é estimada em FP16 para os cenários experimentais (batch = 1). O total do modelo quantizado (`.gguf`) inclui também *embeddings* e demais tensores auxiliares. As estimativas condizem com o que foi verificado empiricamente durante a experimentação. 63

Tabela 15 – DRAM liberada e degradação de decode TPS (%) por configuração combinada, nos cenários extremos C5 (contexto curto) e C4 (contexto longo). A coluna “DRAM lib.” soma os componentes deslocados para a Optane; para a KV-Cache, usa-se o valor do cenário correspondente. 65
Tabela 16 – Pertinência à fronteira de Pareto por configuração e cenário, em ambos os modelos. ✓: Pareto-eficiente; ×: dominada (existe configuração com mais DRAM liberada e vazão igual ou maior); ∅: inviável (decode TPS $<$ 3,0 tok/s). DRAM liberada conforme Tabela 15; vazão conforme Tabelas 9 e 10. 66

Lista de abreviaturas e siglas

Attn	Attention
AWQ	Activation-aware Weight Quantization
CPU	Central Processing Unit
CXL	Compute Express Link
DNN	Deep Neural Network
DRAM	Dynamic Random Access Memory
FFN	Feed-Forward Network
GGML	Georgi Gerganov Machine Learning
GGUF	GPT-Generated Unified Format
GPTQ	Post-Training Quantization for GPT
GQA	Grouped-Query Attention
HBM	High Bandwidth Memory
HPC	High-Performance Computing
H_2O	Heavy-Hitter Oracle
IA	Inteligência Artificial
IC	Intervalo de Confiança
I/O	Input/Output
KV	Key-Value
LLM	Large Language Model
MHA	Multi-Head Attention
ML	Machine Learning
MLP	Multi-Layer Perceptron
MQA	Multi-Query Attention

NUMA	Non-Uniform Memory Access
NVMe	Non-Volatile Memory Express
PCIe	Peripheral Component Interconnect Express
PLN	Processamento de Linguagem Natural
PMem	Persistent Memory
QP	Questão de Pesquisa
RoPE	Rotary Position Embedding
SO	Sistema Operacional
SSD	Solid State Drive
TLB	Translation Lookaside Buffer
TPP	Transparent Page Placement
TPS	Tokens Per Second

Sumário

1 INTRODUÇÃO 19
1.1 Contextualização 20
1.2 Motivação 20
1.3 Problema de pesquisa 21
1.4 Objetivos e contribuições 22
1.5 Organização do Texto 23

2 FUNDAMENTAÇÃO TEÓRICA 25
2.1 Grandes Modelos de Linguagem (LLMs) 25
2.1.1 Parâmetros e Ocupação de Memória 26
2.1.2 Inferência: Prefill, Decode e KV-Cache 27
2.1.3 A Família LLaMA 29
2.2 Memória em Sistemas Computacionais Modernos 29
2.2.1 Hierarquia Convencional e Memórias Heterogêneas 29
2.2.2 Estratificação de Memória 32

3 TRABALHOS RELACIONADOS 35
3.1 Redução da Pegada de Memória 35
3.1.1 Quantização de pesos 35
3.1.2 Redução da KV-cache 36
3.1.3 Posicionamento 36
3.2 *Offloading* com Foco em GPU 36
3.2.1 Posicionamento 37
3.3 Inferência em Sistemas com apenas CPU 37
3.3.1 Posicionamento 39
3.4 Memória Heterogênea e Posicionamento de Dados para LLMs . . . 39
3.4.1 Posicionamento 40
3.5 Síntese e Posicionamento desta Dissertação 40

4 METODOLOGIA 43
4.1 Ambiente Experimental 43
4.1.1 Modelos empregados 44
4.1.2 Software de inferência 45
4.2 Estratégias de posicionamento 46
4.3 Cenários de uso 47
4.3.1 C1 – *Prompt* Longo, Resposta Curta 47

4.3.2 C2 – *Prompt* Curto, Resposta Longa 47
4.3.3 C3 – Balanceado 48
4.3.4 C4 – Contexto Quase Cheio 48
4.3.5 C5 – Controle Mínimo 48

5 RESULTADOS E ANÁLISE 51
5.1 QP1 – Existência de um componente universalmente mais tolerante 53
5.1.1 Síntese 55
5.2 QP2 – Variação da sensibilidade com o regime de operação 55
5.2.1 Relação com o *prefill* 57
5.2.2 Síntese 58
5.3 QP3 – Efeito das diferenças arquiteturais 58
5.3.1 A hierarquia de componentes 58
5.3.2 O paradoxo do *Grouped-Query Attention* 59
5.3.3 A assimetria entre fases 59
5.3.4 Síntese 60
5.4 QP4 – Configurações combinadas 60
5.4.1 Combinações vs. componentes isolados 61
5.4.2 Comparação entre combinações 61
5.4.3 Síntese 63
5.5 Ocupação de memória dos componentes 63
5.6 Síntese e resposta à questão principal 64
5.6.1 Compromissos entre DRAM e vazão 65
5.6.2 Visão integrada: mapas de calor 68
5.6.3 Diretrizes de estratificação 69
5.6.4 Resposta à questão principal 70

6 CONCLUSÃO 73
6.1 Limitações 73
6.2 Trabalhos Futuros 74

REFERÊNCIAS 75

1 Introdução

Grandes Modelos de Linguagem (*Large Language Models* - LLMs), fundamentados na arquitetura Transformer (VASWANI et al., 2017), revolucionaram o campo do processamento de linguagem natural por sua capacidade de captura eficiente de dependências de longo alcance em textos. Nas últimas gerações, o número de parâmetros desses modelos tem crescido de forma exponencial, atingindo centenas de bilhões e exibindo capacidades impressionantes em tarefas como raciocínio, sumarização e geração de código (RADFORD et al., 2019; TOUVRON et al., 2023a; GRATTAFIORI et al., 2024).

O ciclo de vida de um LLM compreende duas fases principais: o treinamento, em que o modelo ajusta seus parâmetros a partir de grandes volumes de dados, e a inferência, em que o modelo treinado é utilizado para gerar respostas a partir de novas entradas. Embora o treinamento demande recursos computacionais massivos, é na inferência que a pressão sobre a memória se manifesta de maneira persistente (MIAO et al., 2024).

A memória consumida durante a inferência distribui-se entre três componentes principais. Os dois primeiros são estáticos: os pesos das camadas de atenção, responsáveis por capturar as relações entre os elementos da sequência, e os pesos das camadas MLP (*Multi-Layer Perceptron*) ou FFN (*Feed-Forward Network*), que adicionam capacidade de representação não linear e tipicamente concentram a maior parcela dos parâmetros do modelo. O terceiro componente é dinâmico: a KV-Cache (*Key-Value Cache*), que armazena os estados de atenção (chaves e valores) já computados em passos anteriores, evitando seu recálculo durante a fase de geração (*decode*). Enquanto os pesos permanecem fixos após o treinamento, a KV-Cache cresce linearmente com o comprimento do contexto processado, podendo atingir volumes de memória comparáveis ou superiores aos dos próprios pesos em cenários de contextos longos (LUO et al., 2025).

O principal desafio da inferência em LLMs reside na sua natureza autorregressiva: na fase de geração, o modelo produz texto de forma sequencial, um token por vez[1], condicionado a todo o contexto anterior. A cada token gerado, o processador precisa ler da memória a totalidade dos pesos do modelo, que podem ocupar vários gigabytes mesmo após quantização. Essa mecânica faz com que a geração seja severamente limitada não pela capacidade aritmética do processador, mas pela largura de banda da memória, conforme evidenciado na literatura (YUAN et al., 2024).

[1] Há técnicas que geram múltiplos tokens por passo, como a decodificação especulativa (LEVIATHAN; KALMAN; MATIAS, 2023). Este trabalho foca na geração token a token.

1.1 Contextualização

Embora aceleradores de hardware dedicados, como GPUs com memória HBM (*High Bandwidth Memory*), dominem os cenários de inferência de alto desempenho, a dependência exclusiva dessas plataformas apresenta barreiras significativas de custo e infraestrutura (JIANG et al., 2025). Nem todo ambiente computacional dispõe de GPUs de última geração. Cenários como computação de borda (*edge computing*), servidores de menor custo em pequenas e médias empresas e implantações locais com restrições orçamentárias exigem soluções alternativas baseadas exclusivamente em CPU (NA et al., 2025). Nessas arquiteturas, a DRAM do sistema atua como único recurso de armazenamento volátil rápido e precisa acomodar simultaneamente os processos do sistema operacional, o volume massivo de pesos do modelo e a KV-Cache gerada dinamicamente durante a inferência. Consequentemente, as limitações inerentes de capacidade e custo por gigabyte da DRAM emergem como gargalos importantes para a viabilidade da inferência de LLMs em CPUs.

O problema se agrava com duas tendências convergentes. Primeiro, modelos cada vez maiores: mesmo com técnicas de quantização que reduzem a precisão numérica dos pesos de 16 para 4 bits (DETTMERS et al., 2022), modelos na faixa de 7 a 13 bilhões de parâmetros ainda ocupam entre 4 e 8 GB comprimidos. Segundo, janelas de contexto cada vez mais longas: aplicações modernas de *Retrieval-Augmented Generation* (RAG) (LEWIS et al., 2020) e assistentes conversacionais multi-turno demandam contextos de milhares a centenas de milhares de tokens, inflando proporcionalmente a KV-Cache.

Para mitigar esse gargalo, o uso de arquiteturas de memória heterogênea organizadas em múltiplas camadas, ou estratificação de memória (*memory tiering*), desponta como uma alternativa promissora (LEE et al., 2023; SONG et al., 2025). Essa abordagem permite acesso a byte a diferentes classes de memória, que podem variar em capacidade, latência de acesso, largura de banda, custo e consumo de energia. Um exemplo típico, adotado neste trabalho, é um sistema composto por memória DRAM e módulos Intel Optane DC (YANG et al., 2020). Atualmente, a tecnologia Compute Express Link (CXL) (SHARMA; BLANKENSHIP; BERGER, 2024) está sendo adotada por muitos fabricantes da indústria de semicondutores, viabilizando a implementação de sistemas com memórias heterogêneas endereçadas a byte.

1.2 Motivação

Em ambientes exclusivamente baseados em CPU, toda a memória usada durante a fase de inferência de um LLM compete pelo mesmo recurso físico: a DRAM do sistema. Para dimensionar o problema, considere uma máquina com 32 GB de DRAM e sem GPU. Um modelo LLaMA 2 7B quantizado em 4 bits ocupa aproximadamente 3,8 GB de pesos, aos quais se somam cerca de 2 GB de KV-Cache para uma janela de 4096 tokens, excluindo ainda a sobrecarga do sistema operacional. Nesse cenário, o sistema opera confortavelmente. Entretanto,

ao escalar para um modelo de 30 bilhões de parâmetros em 8 bits, o volume de pesos sobe para aproximadamente 32 GB, e a soma total ultrapassa a capacidade disponível de DRAM.

A adição de uma outra camada de memória endereçável a byte, como por exemplo um Optane DC de 128 GB, viabiliza cenários anteriormente impossíveis. As motivações concretas para essa estratégia são três:

- **Viabilidade**: modelos cujos pesos e KV-Cache não cabem integralmente na DRAM simplesmente não executam. Uma segunda camada viabiliza a inferência desses modelos sem recorrer a descarregamento (*offloading*) para armazenamento em bloco (SSD), que introduz latências ordens de magnitude superiores (REN et al., 2025);
- **Custo**: a DRAM apresenta custo por gigabyte significativamente superior ao de memórias persistentes ou módulos de memória utilizadas como expansões via CXL (LASCH et al., 2022). Substituir parte da DRAM por uma camada mais densa reduz o custo total de propriedade do sistema (*Total Cost of Ownership*, TCO);
- **Escalabilidade**: com janelas de contexto crescentes (32K, 128K, 1M tokens), a KV-Cache pode consumir dezenas a centenas de gigabytes, ultrapassando qualquer configuração razoável de DRAM. O uso de memórias multicamada oferece um caminho de escalabilidade natural.

1.3 Problema de pesquisa

A organização multicamada permite expandir substancialmente a capacidade de memória disponível, possibilitando a execução de grandes modelos em infraestruturas baseadas exclusivamente em CPU. O desafio central, contudo, reside em como distribuir os três componentes do modelo (pesos da atenção, pesos da MLP e KV-Cache) entre essas camadas, de modo a alcançar um equilíbrio adequado entre capacidade, custo e desempenho. Diante deste cenário desafiador, a pergunta de pesquisa que esta dissertação pretende responder é:

Como a alocação seletiva de componentes internos de um LLM (pesos de atenção, pesos da FFN e KV-Cache) entre DRAM local e camadas de memória de maior capacidade e maior latência afeta o compromisso entre desempenho e uso de DRAM na inferência em ambientes sem aceleradores?

Para isolar o efeito da arquitetura de atenção sobre as decisões de estratificação, são usados dois modelos da família LLaMA (TOUVRON et al., 2023a) de mesma escala (8,B), com arquiteturas distintas: Multi-Head Attention (MHA) no LLaMA2(TOUVRON et al., 2023b) e Grouped-Query Attention (GQA) no LLaMA3(GRATTAFIORI et al., 2024).

Consideramos como plataformas alvos aquelas compostas por desktops ou servidores com memória DRAM limitada, sem nenhum acelerador, e que possuem pelo menos duas camadas de memória. Nos experimentos em particular, consideramos uma camada de memória DRAM e outra com a Intel Optane DC. Cabe destacar que a Intel Optane DC é utilizada neste trabalho como representante concreta de uma classe de tecnologias de memória de segundo nível. Embora tenha sido descontinuada como produto comercial, suas características (latência superior à DRAM porém inferior à de SSDs, acesso endereçável por byte, e capacidade por módulo superior à DRAM) são compartilhadas pelas tecnologias emergentes baseadas no padrão CXL. Assim, entendemos que os resultados obtidos neste trabalho são transferíveis para futuros sistemas com memória expandida via CXL.

Cabe também ressaltar que a adoção de um sistema de memória de duas camadas não contradiz a premissa de acessibilidade econômica que motiva os ambientes sem aceleradores. Aceleradores como GPUs de datacenter representam investimentos da ordem de dezenas de milhares de dólares por unidade, aos quais se somam custos de infraestrutura elétrica, refrigeração e interconexão especializada (NA et al., 2025; STOJKOVIC et al., 2025). Em contraste, a expansão da capacidade de memória via tecnologias como módulos de memória persistente ou dispositivos conectados por CXL opera dentro do ecossistema convencional de servidores, utilizando barramentos e interfaces já padronizados. O custo incremental de adicionar uma camada de memória mais densa e lenta é substancialmente inferior ao de provisionar aceleradores dedicados, situando-se tipicamente em uma fração do custo total do servidor (LI et al., 2023; BERGER et al., 2026). Assim, a proposta deste trabalho não substitui uma solução cara por outra igualmente onerosa, mas sim investiga como maximizar a capacidade efetiva de memória em plataformas de custo moderado, tornando viável a inferência de modelos que, de outra forma, exigiriam hardware significativamente mais dispendioso.

1.4 Objetivos e contribuições

O objetivo geral deste trabalho é investigar estratégias de estratificação de memória para inferência de LLMs em sistemas sem aceleradores com memória heterogênea, quantificando o impacto da alocação seletiva de componentes do modelo (pesos da atenção, pesos da FFN e KV-Cache) sobre o desempenho da inferência e derivando, a partir dessa quantificação, diretrizes práticas de alocação sensíveis ao contexto.

Para alcançar esse objetivo, este trabalho oferece as seguintes contribuições:

1. **Caracterização empírica granular do impacto de cada componente interno de um LLM sob hierarquia de memória heterogênea.** Diferentemente de abordagens que tratam o modelo como bloco monolítico, este trabalho isola e quantifica a degradação de vazão causada pelo deslocamento individual e combinado da KV-Cache, dos pesos de

atenção e dos pesos da FFN/MLP para memória de maior latência (Intel Optane), em cinco cenários de uso e em dois modelos com arquiteturas de atenção distintas (MHA e GQA);

2. **Identificação de que o gargalo de estratificação é sensível ao contexto e governado por um único mecanismo.** Os experimentos demonstram que o componente de maior impacto na degradação não é fixo (a FFN domina sob contexto curto, a KV-Cache sob contexto longo), rotação que decorre do volume de bytes relidos por passo de geração (fixo para os pesos, crescente para o estado da KV-Cache). Esse invariante é consistente entre os dois modelos e explica inclusive o efeito do GQA, que, ao reduzir a KV-Cache, desloca o benefício da estratificação do estado para os pesos de atenção;

3. **Formalização da alocação como problema bi-objetivo e derivação de diretrizes de decisão.** O compromisso entre DRAM liberada e vazão de *decode* é modelado pela fronteira de Pareto combinada a um piso de aceitabilidade de vazão, arcabouço que reduz o espaço de decisão às configurações racionais para cada regime. Desse modelo emergem cinco diretrizes práticas (D1–D5) que orientam o praticante a navegar o compromisso entre economia de memória e desempenho de maneira informada e sensível ao contexto.

1.5 Organização do Texto

Esta dissertação está organizada em seis capítulos, sendo esta sua introdução. O restante tem a seguinte organização.

O Capítulo 2 apresenta a fundamentação teórica necessária para a compreensão da pesquisa, abordando os conceitos fundamentais sobre Grandes Modelos de Linguagem (LLMs) baseados na arquitetura *Transformer*, com ênfase na mecânica de uso de memória durante a fase de inferência. O capítulo também detalha a hierarquia de memória tradicional, as tecnologias emergentes de armazenamento (como a Intel Optane PMem e os padrões CXL) e os conceitos de estratificação (*tiering*) em sistemas computacionais.

O Capítulo 3 discute os trabalhos relacionados na literatura, dividindo-os em abordagens focadas na otimização de memória para inferência de LLMs e estratégias de *tiering* de memória aplicadas a cargas de trabalho de Aprendizado de Máquina. Ao final do capítulo, estabelece-se o posicionamento deste trabalho em relação ao estado da arte.

O Capítulo 4 detalha a metodologia adotada para o estudo empírico, descrevendo a arquitetura do ambiente experimental bem como as métricas e o protocolo rigoroso de execução.

O Capítulo 5 é dedicado à apresentação e análise aprofundada dos resultados obtidos. O capítulo examina os gargalos de desempenho revelados durante o mapeamento de pesos, contrastando o impacto das matrizes *Feed-Forward* (FFN) na fase de *Decode* com os fenômenos de mascaramento de latência observados na fase de *Prefill*. Com base nessa análise de sensibilidade,

o capítulo culmina na proposição de diretrizes práticas e otimizadas para a alocação de memória em arquiteturas restritas.

Por fim, o Capítulo 6 sintetiza as conclusões alcançadas por esta pesquisa, retoma as contribuições principais do estudo, reconhece as limitações do escopo experimental e aponta direções promissoras para trabalhos futuros na área de eficiência de inferência baseada em estratificação de memória.

2 Fundamentação Teórica

Este capítulo apresenta os principais conceitos utilizados no desenvolvimento desta dissertação. Primeiramente, na Seção 2.1, são descritos aqueles relacionados aos Grandes Modelos de Linguagem (LLMs). Posteriormente, na Seção 2.2, são apresentados os principais conceitos sobre memória, com ênfase na estratificação.

2.1 Grandes Modelos de Linguagem (LLMs)

A revolução no Processamento de Linguagem Natural (PLN) e a ascensão dos Grandes Modelos de Linguagem (LLMs) ancoram-se fundamentalmente na arquitetura *Transformer*, introduzida por Vaswani et al. (VASWANI et al., 2017) e ilustrada pela Figura 1. Diferentemente das redes neurais recorrentes clássicas, que processavam sequências de forma estritamente sequencial, o *Transformer* dispensa a recorrência e baseia-se em mecanismos de autoatenção (*Self-Attention*), o que permite a paralelização massiva durante a fase de treinamento e a captura de dependências de longo alcance em textos.

No núcleo dessa arquitetura, cada camada (ou bloco) do Transformer combina dois componentes principais. O primeiro é o mecanismo de Atenção, tradicionalmente o *Multi-Head Attention* (MHA), que projeta a entrada em múltiplos subespaços de representação (cabeças de atenção) para calcular a relevância de cada *token* em relação aos demais (VASWANI et al., 2017). Em modelos mais recentes, visando eficiência de memória, adotou-se o *Grouped-Query Attention* (GQA), que compartilha as matrizes de Chave (*Key*) e Valor (*Value*) entre grupos de cabeças de Consulta (*Query*), reduzindo drasticamente o volume da KV-Cache e o consumo de banda de memória durante a inferência (AINSLIE et al., 2023). O segundo componente é o bloco *Feed-Forward Network* (FFN) ou *Multilayer Perceptron* (MLP), aplicado individualmente a cada posição após a camada de atenção. Este bloco atua como uma memória associativa densa, sendo responsável por armazenar a maior parte do conhecimento factual do modelo (GEVA et al., 2021).

Um LLM moderno empilha dezenas dessas camadas idênticas em profundidade, totalizando bilhões de parâmetros. É essa escala que torna a memória, e não apenas o poder de cálculo, o recurso crítico para a sua execução. Funcionalmente, esses modelos são treinados de forma autorregressiva, aprendendo a estimar a probabilidade do próximo token condicionada a toda a sequência anterior. A geração de texto consiste, portanto, em amostrar sucessivamente dessa distribuição, um token de cada vez. Os LLMs contemporâneos adotam predominantemente a configuração *decoder-only*, na qual uma única pilha de blocos Transformer realiza tanto a interpretação do contexto quanto a geração da resposta. Compreender como essa quantidade massiva de parâmetros se traduz em ocupação concreta de memória é, portanto, o ponto de

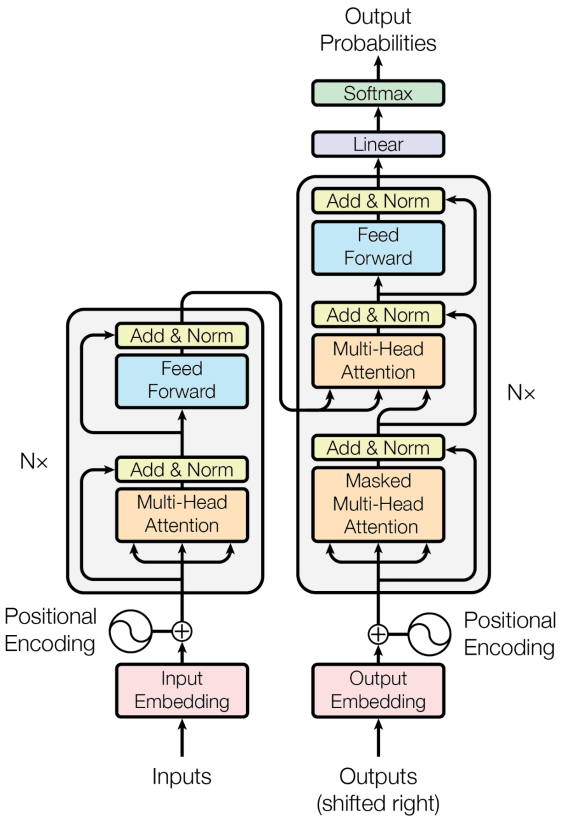


Figura 1 – Diagrama da arquitetura dos Transformers. Extraído de Vaswani et al. (2017).

partida para analisar a viabilidade da execução desses modelos.

2.1.1 Parâmetros e Ocupação de Memória

Os pesos (ou parâmetros) de um LLM são os valores numéricos aprendidos durante o treinamento e que ficam fixos no momento da inferência. Eles estão organizados majoritariamente nas matrizes de projeção dos dois componentes descritos anteriormente, ou seja, os blocos de Atenção e as camadas Feed-Forward, replicadas ao longo de todas as camadas da pilha Transformer. Adicionalmente, a camada de *embedding* e a projeção de saída (frequentemente compartilhadas) contribuem com uma parcela proporcional ao tamanho do vocabulário.

A distribuição desses pesos entre Atenção e *Feed-Forward* não é arbitrária: ela decorre diretamente da geometria da arquitetura e pode ser estimada analiticamente. Seja d a dimensão do modelo e d_{ff} a dimensão oculta do bloco *Feed-Forward*. Desconsiderando termos de menor magnitude (*bias*, normalizações e *embeddings*), a contagem de parâmetros por camada é dominada pelas matrizes de projeção. No bloco de Atenção *Multi-Head* clássico, as quatro projeções (Consulta, Chave, Valor e saída) somam:

$$P_{attn} \approx 4d^2 \tag{2.1}$$

Já o bloco *Feed-Forward* tradicional, composto por duas matrizes (expansão e contração), contribui com:

$$P_{ffn} \approx 2\,d\,d_{ff} \tag{2.2}$$

Adotando a razão canônica do *Transformer* original, $d_{ff} = 4d$ (VASWANI et al., 2017), obtém-se $P_{ffn} \approx 8d^2$. A fração de parâmetros concentrada no bloco *Feed-Forward* é, portanto:

$$\frac{P_{ffn}}{P_{attn} + P_{ffn}} \approx \frac{8d^2}{4d^2 + 8d^2} = \frac{2}{3} \approx 67\% \tag{2.3}$$

Esse resultado revela uma propriedade estrutural, e não específica de um modelo, das arquiteturas *decoder-only*: aproximadamente dois terços dos parâmetros residem nas camadas *Feed-Forward*, contra cerca de um terço no mecanismo de Atenção. Duas otimizações comuns nos LLMs contemporâneos deslocam ligeiramente esses valores, sem alterar a ordem de grandeza. A primeira é a ativação *SwiGLU*, que introduz uma terceira matriz no bloco *Feed-Forward*. Para preservar a contagem total de parâmetros, reduz-se d_{ff} a aproximadamente $\frac{2}{3} \cdot 4d$ (SHAZEER, 2020). A segunda é o *Grouped-Query Attention*, que diminui o número de matrizes de Chave e Valor, reduzindo a fração da Atenção (AINSLIE et al., 2023). O efeito líquido mantém a predominância do bloco *Feed-Forward*, tipicamente em torno de 60% a 67% do total.

Definida a distribuição interna dos pesos, a ocupação de memória é determinada pelo produto entre o número total de parâmetros e o número de *bytes* usados para representar cada valor. Em precisão de ponto flutuante de 16 bits (FP16/BF16), formato comum para inferência, cada parâmetro ocupa 2 bytes. Assim, um modelo de 7 bilhões de parâmetros demanda aproximadamente 14 GB apenas para armazenar seus pesos estáticos, antes de contabilizar quaisquer estruturas dinâmicas. Essa relação direta entre contagem de parâmetros e bytes alocados explica por que a memória se torna o principal fator limitante na execução de LLMs, sobretudo em ambientes restritos como os baseados exclusivamente em CPU.

É importante notar que esses pesos não são acessados de modo uniforme ao longo da inferência. Como será detalhado adiante, a fase de geração autorregressiva requer a leitura da totalidade das matrizes de pesos a cada token produzido, o que torna não apenas o volume total de parâmetros relevante, mas também a largura de banda disponível para movê-los da memória até o processador. Combinada com a distribuição derivada acima, essa observação antecipa que o bloco *Feed-Forward*, por concentrar a maioria dos parâmetros, será o principal responsável pela pressão sobre o barramento de memória.

2.1.2 Inferência: Prefill, Decode e KV-Cache

Sob a perspectiva da execução, a inferência divide-se em duas fases com perfis de hardware diametralmente opostos:

- Fase de *Prefill* (Processamento do *Prompt*): O modelo recebe o texto de entrada (contexto) e o processa em paralelo. Esta fase envolve a multiplicação de matrizes densas de grandes dimensões. Por exigir intensos cálculos aritméticos simultâneos, é classificada como limitada pela capacidade computacional do processador (*compute-bound*);

- Fase de *Decode* (Geração Autorregressiva): O modelo gera um *token* de cada vez, utilizando o *token* previamente gerado como entrada para o próximo ciclo. Esta etapa exige que as matrizes de pesos de todo o modelo sejam lidas da memória física para cada única palavra gerada. Por não permitir paralelização temporal, o desempenho passa a ser severamente ditado pela largura de banda da memória (*memory-bound*). Conforme derivado anteriormente, o bloco *Feed-Forward* concentra a maioria dos parâmetros e, por isso, domina o tráfego de memória nesta fase.

Em ambientes de execução limitados pela CPU, a viabilidade da inferência depende estritamente do gerenciamento da memória principal (DRAM), que deve acomodar dois grandes volumes de dados: os pesos estáticos do modelo, já discutidos, e as estruturas de estado dinâmico da inferência. Assim, além dos pesos estáticos, a inferência autorregressiva depende criticamente do mecanismo de *KV-Cache*. Para evitar recalcular a atenção dos *tokens* anteriores a cada nova palavra gerada, o modelo armazena em memória as representações de Chave e Valor de todos os *tokens* já processados. O tamanho da *KV-Cache* cresce linearmente com o comprimento da sequência e com o tamanho do *batch*, agravando o consumo de memória em cenários com *batch sizes* elevados ou janelas de contexto muito longas, podendo facilmente superar dezenas de gigabytes e rivalizar com o tamanho do próprio modelo (LUO et al., 2025).

Para mitigar essa explosão no consumo de memória DRAM, podem ser aplicadas técnicas de quantização, que reduzem a precisão numérica dos pesos de ponto flutuante de 16 ou 32 bits (FP16/FP32) para representações inteiras menores (FRANTAR et al., 2023). Formatos de quantização como o Q8_0 (8 bits simétricos), Q4_0 (4 bits) e os métodos avançados *k-quants* como o Q4_K_M (que utiliza diferentes níveis de precisão para matrizes distintas baseadas na sua sensibilidade ao erro) permitem comprimir o modelo estruturalmente de 2 a 4 vezes, com degradações marginais na perplexidade e na qualidade da resposta gerada (LIN et al., 2024).

Neste ecossistema, *frameworks* de inferência otimizados para arquiteturas de uso geral, notadamente o *llama.cpp* e sua biblioteca tensorial subjacente *GGML* (GERGANOV et al., 2023), destacam-se por permitir o carregamento dinâmico desses modelos quantizados. Projetado primariamente para eficiência em CPU, o *llama.cpp* expõe interfaces manuais de gerenciamento de memória (através de *memory mapping* e manipulação de tensores) que são indispensáveis para experimentações de hardware e alocação heterogênea.

2.1.3 A Família LLaMA

Os fundamentos da arquitetura *Transformer* discutidos anteriormente materializam-se de forma concreta na família de modelos LLaMA (*Large Language Model Meta AI*), introduzida por Meta AI (TOUVRON et al., 2023a). Tornada um dos principais *baselines* acadêmicos para LLMs de código aberto, essa família adota uma arquitetura *decoder-only* otimizada para inferência, na qual uma única pilha de blocos *Transformer* realiza tanto a interpretação do contexto quanto a geração autorregressiva da resposta.

Em relação ao *Transformer* original, o LLaMA incorpora três refinamentos hoje consolidados em modelos de código aberto: a normalização *RMSNorm* aplicada na entrada de cada subcamada (ZHANG; SENNRICH, 2019), a função de ativação *SwiGLU* no bloco *Feed-Forward* (SHAZEER, 2020) e a codificação posicional rotativa (RoPE), que representa a posição dos *tokens* diretamente no mecanismo de atenção (SU et al., 2024). Disponível em múltiplas escalas e gerações, a família LLaMA constitui, assim, uma referência representativa das arquiteturas *decoder-only* contemporâneas, sendo empregada neste trabalho.

2.2 Memória em Sistemas Computacionais Modernos

A evolução das arquiteturas de computadores é historicamente marcada pelo fenômeno conhecido como *Memory Wall*, que descreve a disparidade crescente entre o desempenho dos processadores e a latência e largura de banda das memórias principais (WULF; MCKEE, 1995). Para mitigar esse descasamento, os sistemas computacionais organizam a memória em uma hierarquia de níveis, estruturada sobre os princípios de localidade temporal e espacial. Com o surgimento de tecnologias que não se encaixam na pirâmide convencional, como memórias persistentes e expansores de memória via barramento, essa hierarquia tornou-se *heterogênea*, exigindo mecanismos dedicados de governança. Esta seção apresenta os fundamentos dessa hierarquia e as abordagens de estratificação que sustentam a proposta deste trabalho.

2.2.1 Hierarquia Convencional e Memórias Heterogêneas

Na pirâmide convencional de memória (veja Figura 2), o topo é ocupado por registradores e múltiplos níveis de memória *cache* SRAM (L1, L2 e L3), que oferecem latências na ordem de frações de nanossegundos, porém com capacidade extremamente limitada. Logo abaixo situa-se a memória principal dinâmica (*Dynamic Random-Access Memory* – DRAM), que tipicamente constitui o espaço de trabalho volátil do sistema operacional e aplicações, equilibrando latência intermediária e capacidade moderada. A base é ocupada pelo armazenamento secundário não volátil, tipicamente discos rígidos (HDDs) e unidades de estado sólido (*Solid-State Drives* – SSDs), que fornecem alta densidade a custos reduzidos, sob latências ordens de magnitude superiores (HENNESSY; PATTERSON, 2017).

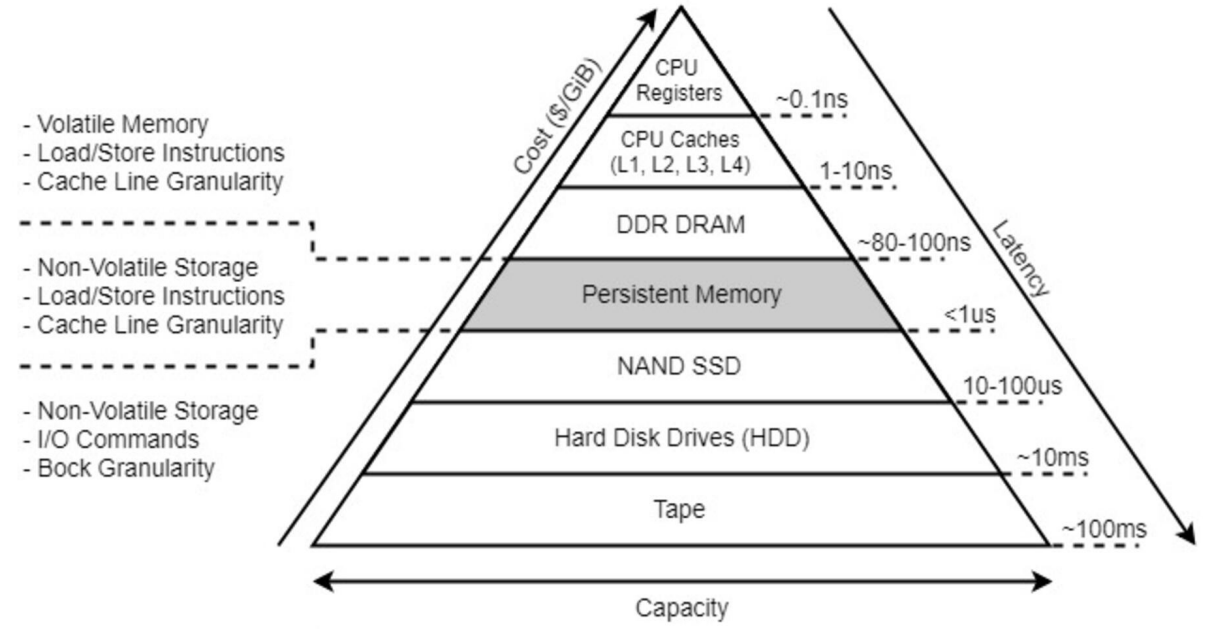


Figura 2 – Hierarquia Memória-Armazenamento com camada de Memória Persistente. Extraído de Scargall (2020).

A rigidez dessa pirâmide foi rompida com a introdução das memórias persistentes (*Persistent Memory*, PM) (BALDASSIN et al., 2021). Estas memórias provêm armazenamento persistente mas são rápidas o suficiente para permitir acesso de leitura/escrita na granularidade de byte. Elas também possuem densidades mais altas e melhor custo por bit quando comparadas às DRAM (SCARGALL, 2020). A tecnologia Intel Optane DC Persistent Memory (YANG et al., 2020), principal representante comercial dessa classe, conecta-se diretamente aos canais de memória do processador via *slots* DIMM, combinando persistência e alta densidade com endereçamento por *byte* e latências significativamente inferiores às do armazenamento convencional.

A tecnologia Intel Optane pode ser configurada para operar em dois modos de hardware mutuamente exclusivos, os quais alteram profundamente a visibilidade do mapa de memória pelo sistema operacional:

- **Memory Mode:** Neste modo, o sistema operacional enxerga a capacidade do dispositivo como a memória RAM total disponível no sistema. A DRAM física convencional passa a agir estritamente como uma camada de *cache* de hardware transparente (nível L4), ocultando a complexidade do arranjo. Embora elimine a necessidade de modificações no *software*, este modo pode introduzir imprevisibilidade em cargas de trabalho de alta intensidade por efetivamente considerar a DRAM como mais um nível de cache (HILDEBRAND et al., 2021);

- **App Direct Mode:** Neste modo, a DRAM e a Optane são expostas de forma independente, mapeadas como nós NUMA (*Non-Uniform Memory Access*) distintos. Cabe explicitamente

Tabela 1 – Comparativo de desempenho físico entre DRAM DDR4 e Intel Optane DC PMem em modo *App Direct*, por módulo DIMM individual (YANG et al., 2020).

Métrica	**DRAM DDR4**	**Optane PMem**
Latência de leitura (aleatória)	$\approx$ 80 ns	$\approx$ 346 ns ($\approx$ 4,3$\times$)
Largura de banda de leitura	$\approx$ 25 GB/s	$\approx$ 6,6 GB/s ($\approx$ 3,8$\times$ menor)
Largura de banda de escrita	$\approx$ 25 GB/s	$\approx$ 2,3 GB/s ($\approx$ 10,9$\times$ menor)
Endereçamento	Por byte	Por byte
Volatilidade	Volátil	Persistente

ao software decidir quais estruturas de dados residirão em quais memória (DRAM ou Optane). Este modo viabiliza a estratificação explícita em nível de aplicação, técnica central explorada neste trabalho, permitindo o isolamento controlado de componentes como a *KV-Cache* e os pesos de Atenção e FFN.

A Tabela 1 consolida as especificações operacionais típicas que caracterizam a assimetria física entre DRAM e Optane, conforme medições empíricas de Yang et al. (YANG et al., 2020). Os dados revelam uma assimetria pronunciada e não uniforme entre os tipos de acesso. A latência de leitura aleatória da Optane é aproximadamente 4,3 vezes superior à da DRAM ($\approx$346 ns contra $\approx$80 ns). A largura de banda de leitura sofre redução de cerca de 3,8 vezes (6,6 GB/s contra 25 GB/s por módulo DIMM), enquanto a largura de banda de escrita apresenta a degradação mais severa, operando a uma taxa aproximadamente 10,9 vezes inferior à da DRAM (2,3 GB/s contra 25 GB/s). Essa assimetria entre leitura e escrita é uma característica fundamental da tecnologia 3D XPoint subjacente à Optane e possui implicações diretas para a inferência de modelos de linguagem: componentes que demandam acessos intensos de escrita encontram na Optane um potencial gargalo físico, enquanto estruturas predominantemente de leitura podem tirar proveito de sua alta capacidade com penalidade de latência moderada.

Embora a linha Intel Optane tenha sido comercialmente descontinuada, os conceitos arquiteturais validados por ela constituem a base do padrão *Compute Express Link* (CXL) (SHARMA; BLANKENSHIP; BERGER, 2024). O CXL é um protocolo aberto de interconexão, executado sobre a camada física do barramento PCIe (a partir da geração 5.0), projetado para prover conexões de alta velocidade com coerência de *cache* entre a CPU e dispositivos acoplados. Dispositivos classificados como *CXL.mem* (Tipo 3) permitem a expansão da memória do servidor por meio de módulos conectados ao barramento PCIe. Do ponto de vista do sistema operacional, essas memórias são mapeadas como nós NUMA sem CPU local (*CPU-less NUMA nodes*), apresentando perfil de desempenho análogo ao da Optane em modo *App Direct*: latências ligeiramente superiores à DRAM local e largura de banda limitada pelas características do barramento (WANG et al., 2024; WU et al., 2024).

Essa equivalência funcional possui uma consequência metodológica direta para este trabalho: as experimentações e diretrizes de alocação desenvolvidas sobre a plataforma Optane

possuem *validade externa imediata* para ambientes CXL, servindo como modelo preditivo para a otimização de *engines* de inferência de LLMs na era da memória desagregada.

Há outras duas tecnologias importantes que estão sendo atualmente empregadas no contexto de LLMs, embora não sejam empregadas diretamente neste trabalho. A primeira é encontrada em plataformas com aceleradores como GPUs e TPUs, nos quais a memória principal do acelerador emprega tipicamente *High Bandwidth Memory* (HBM) (LEE et al., 2016), uma variante de DRAM que empilha verticalmente múltiplas camadas de *dies* de memória interconectadas por *through-silicon vias* (TSVs). Essa arquitetura 3D permite larguras de banda na ordem de TB/s, muito superiores às da DRAM convencional DDR, porém com capacidade limitada a dezenas de gigabytes por dispositivo e custo significativamente elevado. A capacidade restrita da HBM é justamente o que motiva as técnicas de offloading discutidas no Capítulo 3, nas quais dados são movidos da HBM para memórias de maior capacidade (DRAM, CXL ou SSD) durante a inferência.

A segunda tecnologia é uma abordagem complementar à expansão de capacidade, conhecida como *Processing-in-Memory* (PIM) (GHOSE et al., 2019; MUTLU et al., 2021), que integra unidades de computação diretamente nos módulos de memória, eliminando a necessidade de transferir dados para a CPU ou GPU antes de processá-los. Arquiteturas PIM exploram a alta largura de banda *interna* dos *dies* de memória, muito superior à largura de banda disponível no barramento externo, para executar operações como multiplicação matriz-vetor próximo aos dados. Embora o conceito de PIM seja antigo, avanços recentes em empilhamento 3D e no padrão CXL renovaram o interesse na abordagem (LEE; KIM et al., 2022). Contudo, dispositivos CXL-PIM ainda não estão disponíveis comercialmente, limitando sua adoção a protótipos e simulações de pesquisa.

2.2.2 Estratificação de Memória

A coexistência de múltiplos níveis de memória com capacidades e velocidades distintas (e.g., DRAM, Optane e memórias CXL em geral) não resulta automaticamente em ganhos de desempenho ou redução de custos. É necessário um mecanismo de governança capaz de decidir *quais dados* residem em *qual nível* físico a cada instante. Esse processo, denominado estratificação de memória (*memory tiering*) (LEE et al., 2023), busca posicionar os dados mais frequentemente acessados (*hot data*) nos níveis de maior desempenho e mover os dados de acesso esporádico (*cold data*) para os níveis de alta capacidade e menor custo.

A motivação para a estratificação é dupla. Do ponto de vista econômico, sistemas de memória heterogênea permitem ampliar a capacidade total sem o custo proibitivo de uma configuração inteiramente baseada em DRAM, fator particularmente relevante quando modelos de linguagem com bilhões de parâmetros precisam residir integralmente em memória endereçável por *byte*. Do ponto de vista de desempenho, a estratificação adequada evita que dados críticos sejam penalizados pela latência dos níveis mais lentos, ao mesmo tempo em que permite

que dados tolerantes a latência ocupem a memória expandida sem degradação perceptível na experiência do usuário final.

Em termos mais amplos, a estratificação pode ser classificada metodologicamente em duas abordagens, conforme o nível de controle que a aplicação exerce sobre o *hardware*: *transparente* (MARUF et al., 2023) e *explícita* (LEE et al., 2025).

2.2.2.a Abordagem Transparente

Na estratificação transparente, a aplicação não possui ciência da composição física da memória; o *kernel* do sistema operacional assume a responsabilidade de monitorar e migrar páginas de memória de forma reativa. No ecossistema Linux, o mecanismo clássico para essa função é o *AutoNUMA* (CORBET, 2012). Ele rastreia periodicamente os acessos das CPUs às páginas de memória por meio de falhas de página induzidas (*NUMA hinting faults*): ao detectar que um processo no Nó 0 acessa repetidamente uma página fisicamente localizada no Nó 1, o *kernel* migra automaticamente essa página para aproximá-la do processador.

Com o advento de memórias CXL expostas como nós NUMA dedicados, o Linux introduziu um mecanismo de estratificação automático que estende as funcionalidades do AutoNUMA, baseado no trabalho desenvolvido por Maruf et al. (MARUF et al., 2023), conhecido como TPP (*Transparent Page Placement*). O TPP estabelece uma distinção explícita entre nós de primeiro nível (DRAM local) e nós de segundo nível (*slow tiers*). Quando a pressão de alocação na DRAM aproxima-se do limite de saturação, o TPP executa de forma assíncrona o *page demotion* (rebaixamento), movendo páginas menos utilizadas para a memória lenta. Se uma página rebaixada voltar a ser intensamente requisitada, ocorre o *page promotion* (promoção), trazendo-a de volta ao nível rápido.

2.2.2.b Abordagem Explícita

Embora os mecanismos transparentes facilitem a execução de aplicações legadas, eles podem apresentar limitações em cargas de trabalho de inferência de LLMs. O sistema operacional opera sob a granularidade rígida de páginas de memória (tipicamente 4 KB ou *huge pages* de 2 MB) e é completamente alheio à semântica das estruturas de dados internas do modelo. O *kernel* não é capaz de distinguir se uma determinada página contém chaves voláteis de atenção (*KV-Cache*), pesos estáticos de uma camada *Feed-Forward* ou parâmetros de *embedding*. Como consequência, a migração automática pode provocar o fenômeno de *thrashing* (GALVIN et al., 2018), ou seja, promoções e rebaixamentos contínuos que exaurem a largura de banda do barramento, além de introduzir latências de migração imprevisíveis durante a geração de *tokens*.

Em contrapartida, a inferência de LLMs possui uma característica que a torna particularmente adequada à estratificação *explícita*: seus padrões de acesso à memória são *estruturalmente previsíveis*. Conforme derivado na Seção 2.1.1, a distribuição de parâmetros entre os blocos *Feed-Forward* e de Atenção é determinística e conhecida antes do início da inferência. Além

disso, a *KV-Cache* cresce de forma monotônica e previsível durante a fase de decodificação. Essa previsibilidade permite que a decisão de alocação seja tomada *a priori*, no momento do carregamento do modelo, eliminando a necessidade de monitoramento reativo e migração em tempo de execução.

Portanto, na abordagem explícita a decisão de alocação é transferida para a aplicação, que utiliza APIs e bibliotecas especializadas para atribuir cada estrutura de dados ao nível de *hardware* desejado desde o momento de sua criação. Há duas bibliotecas principais que podem ser utilizadas:

- **libmemkind:** Construída sobre o alocador extensível *jemalloc*, a `libmemkind` (CANTALUPO et al., 2015) provê abstrações de alto nível que permitem definir partições de memória denominadas *kinds*. Por meio de chamadas como `memkind_malloc()`, a aplicação solicita explicitamente que um bloco de memória seja criado no *pool* DRAM convencional ou mapeado nas memórias Optane/CXL configuradas como nós KMEM do Linux;
- **libpmem:** Parte do pacote PMDK (*Persistent Memory Development Kit*) (SCARGALL, 2020), a `libpmem` oferece primitivas de baixo nível para acesso direto (DAX –*Direct Access*). Ela permite que arquivos mapeados em memória (`mmap`) localizados na PM sejam lidos e gravados sem passar pelo *page cache* do sistema operacional, eliminando chamadas de sistema tradicionais de E/S.

A vantagem fundamental da abordagem explícita reside na possibilidade de explorar o *conhecimento semântico* que a aplicação possui sobre seus próprios dados. Quando os padrões de acesso à memória são estruturalmente previsíveis, a alocação estática supera a migração reativa tanto em eficiência quanto em previsibilidade de latência. Essa propriedade é especialmente relevante para cargas de trabalho cuja distribuição interna de dados entre componentes estáticos e dinâmicos é conhecida antes do início da execução, como é o caso da inferência de modelos de linguagem de grande escala discutida na Seção 2.1, sendo assim a forma preferida para realização dos experimentos conduzidos nesta dissertação.

3 Trabalhos Relacionados

A inferência de Grandes Modelos de Linguagem é fundamentalmente limitada por memória. Modelos com dezenas a centenas de bilhões de parâmetros exigem capacidade e largura de banda de memória que frequentemente excedem os recursos disponíveis em plataformas existentes, sejam GPUs de alta capacidade ou servidores com apenas CPUs. A literatura ataca esse problema por múltiplos ângulos: reduzir o tamanho dos dados em memória (quantização, compressão, esparsificação), mover dados entre dispositivos com diferentes características (*offloading*), ou expandir a memória disponível com tecnologias heterogêneas (CXL, PM).

Essas abordagens são *ortogonais e combináveis* e, como argumentaremos ao final deste capítulo, nenhuma explora plenamente a estratificação de memória principal heterogênea em cenário com apenas CPU. Este capítulo organiza os trabalhos relacionados em quatro eixos: abordagens de redução da pegada (*footprint*) (Seção 3.1), *offloading* com foco em GPU (Seção 3.2), inferência somente em CPU (Seção 3.3) e memória heterogênea com posicionamento de dados para LLMs (Seção 3.4). A Seção 3.5 sintetiza as lacunas identificadas e articula a contribuição desta dissertação.

3.1 Redução da Pegada de Memória

Antes de discutir estratégias de movimentação e posicionamento de dados, é necessário reconhecer um conjunto de técnicas que atacam o problema de memória pela raiz: reduzir o volume de dados que precisa residir em memória. Essas técnicas são ortogonais à estratificação e podem ser combinadas com ela.

3.1.1 Quantização de pesos

A quantização reduz a precisão numérica dos parâmetros do modelo, diminuindo proporcionalmente a pegada de memória. GPTQ (FRANTAR et al., 2023) realiza quantização pós-treinamento para INT4 usando informação de segunda ordem (Hessiana aproximada por bloco), alcançando compressão de 4× com degradação mínima de acurácia. AWQ (LIN et al., 2024) observa que nem todos os pesos têm igual importância e protege um pequeno subconjunto de canais *salient* durante a quantização, melhorando a qualidade para formatos INT3–INT4. O formato GGUF, utilizado pelo framework `llama.cpp` (GERGANOV et al., 2023), oferece múltiplos esquemas de quantização (Q4_0, Q4_K_M, Q5_K_M, entre outros) que permitem executar modelos de grande porte em hardware convencional com memória limitada.

Tabela 2 – Trabalhos de *offloading* com foco em GPU para inferência de LLMs.

Trabalho	**Publicação**	**Pesos**	**KV-cache**	**Dispositivos**
FlexGen (SHENG et al., 2023)	ICML '23	✓	✓	GPU → CPU → SSD
DAK (LIN; GUO; LIN, 2025)	arXiv '25	✓	✓	GPU HBM ↔ CPU DRAM
KVDrive (LIN et al., 2025)	arXiv '25	—	✓	HBM → DRAM → SSD
Ganjihal (GANJIHAL, 2025)	arXiv '25	—	✓	6 tiers (HBM→PFS)
Kareto (ZHENG et al., 2025)	arXiv '25	—	✓	Multi-tier (Pareto)
FlexInfer (NA et al., 2025)	MLSys '25	✓	✓	GPU ↔ CPU (dinâmico)
HybridGen (LIN et al., 2025)	arXiv '25	—	✓	GPU + CPU + CXL

3.1.2 Redução da KV-cache

A KV-cache armazena os estados intermediários de atenção e cresce linearmente com o comprimento do contexto, tornando-se um gargalo de capacidade para sequências longas. Arquiteturas como Grouped-Query Attention (GQA) (AINSLIE et al., 2023) e Multi-Query Attention (MQA) (SHAZEER, 2019) reduzem estruturalmente o tamanho da KV-cache ao compartilhar projeções de chave e valor entre múltiplas *heads* de atenção. Técnicas de despejo como H_2O (ZHANG et al., 2023) e StreamingLLM (XIAO et al., 2024) descartam seletivamente tokens menos relevantes do cache durante a geração, permitindo janelas de contexto efetivas maiores sob restrições de memória fixa.

3.1.3 Posicionamento

Essas técnicas reduzem a pegada de memória, mas não eliminam o problema de capacidade: modelos com dezenas de bilhões de parâmetros, mesmo quantizados em INT4, continuam a requerer dezenas de gigabytes de memória principal. Além disso, são complementares à estratificação: pesos quantizados ainda podem se beneficiar de posicionamento inteligente entre camadas de memória com diferentes latências e larguras de banda. Esta dissertação assume que quantização pode ser aplicada como pré-processamento, e foca na camada seguinte do problema: onde posicionar os dados já comprimidos.

3.2 *Offloading* com Foco em GPU

A abordagem mais explorada na literatura consiste em utilizar a memória da CPU (DRAM) e/ou armazenamento (SSD) como extensão da memória limitada da GPU (HBM), movendo tensores entre esses níveis durante a inferência. A Tabela 2 sumariza os trabalhos descritos a seguir, identificando o local de publicação, se os pesos ou KV-cache são considerados e as memórias dos dispositivos considerados.

FlexGen (SHENG et al., 2023) foi um dos primeiros sistemas a tratar o *offloading* de LLMs de forma holística, movendo pesos, KV-cache e ativações entre GPU, CPU e disco. FlexGen formula o problema de escalonamento como programação linear, buscando padrões

eficientes de armazenamento e acesso a tensores sob restrições de hardware. Executando o modelo OPT-175B em uma única GPU de 16 GB, FlexGen atinge vazão de geração de 1 token/s com tamanho de *batch* efetivo de 144. DAK (LIN; GUO; LIN, 2025) habilita acesso direto da GPU à memória remota via Tensor Memory Accelerator (TMA), eliminando a necessidade de *prefetch* para HBM e alcançando agregação de largura de banda próxima do ótimo em sistemas NVLink-C2C (3× sobre baselines) e PCIe (1,8×).

Trabalhos subsequentes refinaram o *offloading* com foco na KV-Cache, que domina a movimentação de dados em contextos longos. KVDrive (LIN et al., 2025) propõe um sistema multi-camada (GPU HBM → DRAM → SSD) que, diferentemente de abordagens anteriores focadas em aumentar a esparsidade do cache, otimiza o volume de transferência distribuindo blocos de KV entre as camadas de acordo com a frequência de acesso. O trabalho de Ganjihal (GANJIHAL, 2025) propõe uma hierarquia de 6 camadas (HBM → DRAM → CXL → NVMe → RDMA → sistema de arquivos paralelo) com predição Bayesiana de reuso de blocos KV, alcançando taxa de acerto em cache de 70–84%. Kareto (ZHENG et al., 2025) formula a estratificação de KV-Cache como otimização multi-objetivo (custo × largura de banda × latência) e aproxima a fronteira de Pareto no espaço de configurações de armazenamento.

FlexInfer (NA et al., 2025) observa que CPUs modernas com aceleradores de GEMM dedicados (Intel AMX) podem ser competitivas com GPUs na fase de decode, e propõe seleção dinâmica da política de execução (CPU vs. GPU) para cada fase da inferência (prefill e decode), com base nas características do hardware e parâmetros de runtime. HybridGen (LIN et al., 2025) estende a computação de atenção para execução colaborativa CPU–GPU em sistemas com memória expandida via CXL, usando paralelismo de *logits* de atenção e mapeamento semântico da KV-Cache considerando penalidades NUMA de memórias heterogêneas.

3.2.1 Posicionamento

Essa categoria é a mais ativa na literatura. Porém, todos os trabalhos são centrados na GPU: a GPU permanece como unidade de computação principal e CPU/armazenamento secundário servem como um escape. O problema de colocação na memória é formulado do ponto de vista do escalonamento de dados para a GPU. Em cenários com apenas CPU, esses mecanismos não se aplicam diretamente, pois a hierarquia de memória e os padrões de acesso são fundamentalmente diferentes.

3.3 Inferência em Sistemas com apenas CPU

Um corpo crescente de trabalhos aborda a inferência de LLMs sem GPU ou com recursos de GPU extremamente limitados, visando implantação em dispositivos de borda, *smartphones* ou servidores. A Tabela 3 compara os principais trabalhos nesta linha, mostrando o local de publicação, a abordagem adotada e os dispositivos envolvidos.

Tabela 3 – Trabalhos de inferência com apenas CPU para LLMs.

Trabalho	**Publicação**	**Abordagem**	**Dispositivos**
llama.cpp (GERGANOV et al., 2023)	Open-source	Inferência CPU, quant. GGUF, mmap	DRAM ↔ disco (implícito)
LLM Flash (ALIZADEH et al., 2024)	ACL '24	Modelo de custo flash-aware, *windowing*, *row-column bundling*	DRAM ↔ flash
FlexInfer (DU et al., 2025)	EuroMLSys '25	*Offloading* adaptativo com *prefetch*, *memory locking*	DRAM ↔ SSD (carga adaptável)
CENT (GU et al., 2025)	ASPLOS '25	GPU-free com CXL	CXL-PIM (hardware customizado)

O arcabouço `llama.cpp` (GERGANOV et al., 2023) é a implementação de referência para inferência em apenas CPU, suportando múltiplos esquemas de quantização via formato GGUF e utilizando `mmap` para mapear pesos diretamente do armazenamento, permitindo que o sistema operacional gerencie implicitamente a paginação entre DRAM e disco.

LLM in a Flash (ALIZADEH et al., 2024), publicado no ACL 2024, aborda o cenário em que o modelo excede a capacidade de DRAM disponível, armazenando pesos em memória flash e carregando-os sob demanda. Os autores constroem um modelo de custo de inferência que incorpora as características do flash (latência, largura de banda sequencial vs. aleatório) para guiar duas otimizações: *windowing*, que reutiliza neurônios previamente ativados para reduzir o volume de transferência, e *row-column bundling*, que agrupa leituras em blocos contíguos alinhados às características de acesso sequencial do flash. Explorando esparsidade de ativação, o sistema executa modelos com até 2× o tamanho da DRAM disponível, com speedups de 4–5× em CPU e 20–25× em GPU sobre carregamento ingênuo.

FlexInfer (DU et al., 2025), publicado no EuroMLSys 2025, propõe um arcabouço de *offloading* otimizado para inferência em dispositivos sem grande poder computacional, implementado sobre o `llama.cpp`. FlexInfer introduz três técnicas complementares: (i) *asynchronous prefetching* por camada, sobrepondo E/S e computação para esconder a latência de acesso ao armazenamento; (ii) *balanced memory locking*, que particiona tensores para maximizar o paralelismo entre transferência e computação; e (iii) *flexible tensor preservation*, que gerencia em granularidade fina quais tensores manter em memória DRAM, adaptando-se ao budget de memória especificado pelo usuário. Nos experimentos, FlexInfer alcança até 12,5× melhor largura de banda comparado a métodos existentes sob restrições severas de memória, com desempenho que escala linearmente com a memória disponível.

CENT (GU et al., 2025), publicado no ASPLOS 2025, propõe um sistema que combina expansão de memória via CXL com unidades de processamento em memória (PIM) para fornecer a largura de banda necessária para inferência de LLMs sem depender de GPUs. Embora o CENT utilize memória CXL e, portanto, opere sobre um substrato heterogêneo, sua proposta depende

Tabela 4 – Trabalhos de memória heterogênea e posicionamento de dados para inferência de LLMs.

Trabalho	Publicação	Escopo	Dispositivos
Fang et al. (FANG et al., 2025)	arXiv '25	KV-Cache (analítico)	HBM ↔ DRAM (GPU-centric)
Beluga (YANG et al., 2025)	arXiv '25	KV-Cache	GPU HBM + CXL pool
Tang et al. (TANG et al., 2024)	NeurIPS Wksp. '24	KV-Cache	GPU HBM + CXL
PAM (LIU et al., 2025)	arXiv '25	KV-Cache	HBM + CXL-PIM

de unidades PIM customizadas acopladas aos módulos CXL para realizar computação próxima à memória. Assim, a heterogeneidade em CENT serve como veículo para PIM, e não como mecanismo de estratificação diferenciado de estruturas de dados entre camadas de memória com características distintas.

3.3.1 Posicionamento

Estes trabalhos demonstram a viabilidade e o interesse crescente na inferência com apenas CPU. O LLM in a Flash de Alizadeh et al. (ALIZADEH et al., 2024) é particularmente relevante por demonstrar que modelar as características do meio de armazenamento gera ganhos substanciais de desempenho, um raciocínio que esta dissertação estende para a camada de memória principal heterogênea, onde a assimetria entre DRAM e memória CXL apresenta trade-offs análogos de latência e largura de banda. O FlexInfer de Du et al. (DU et al., 2025) é igualmente relevante por operar sobre o mesmo arcabouço (`llama.cpp`) e por demonstrar que *offloading* adaptativo com gerenciamento fino de tensores pode entregar desempenho escalável. Entretanto, todos esses trabalhos tratam a memória principal como homogênea (DRAM pura) e realizam posicionamento apenas considerando DRAM e armazenamento secundário (SSD/flash). Mesmo o CENT, que utiliza memória CXL, o faz como substrato para PIM e não como camada de memória complementar à DRAM. Nenhum explora a memória principal heterogênea como mecanismo de posicionamento diferenciado, como DRAM combinada com PM ou DRAM combinada com memória CXL, para posicionamento de estruturas de dados do *transformer* com base em seus padrões de acesso.

3.4 Memória Heterogênea e Posicionamento de Dados para LLMs

Os trabalhos mais próximos da proposta desta dissertação são aqueles que conectam tecnologias de memória heterogênea diretamente à inferência de LLMs, investigando o posicionamento de dados em múltiplas camadas de memória. A Tabela 4 sumariza esses trabalhos, especificando o local de publicação, escopo do posicionamento e os dispositivos envolvidos.

Fang et al. (FANG et al., 2025) investigam o posicionamento dinâmico da KV-Cache em sistemas de memória heterogênea compostos por HBM e DRAM. Diferentemente dos demais trabalhos nesta área, a contribuição é primariamente analítica: os autores formulam o problema de posicionamento como otimização para maximizar a utilização agregada de largura de banda sob restrições de capacidade, e derivam um limite superior teórico de desempenho que qualquer política de escalonamento poderia alcançar. Os autores afirmam ser a primeira formalização matemática do problema de posicionamento de dados para KV-Cache de LLMs em memória heterogênea.

Beluga (YANG et al., 2025), desenvolvido pelo Alibaba Cloud, propõe uma arquitetura de memória baseada em CXL que permite a GPUs e CPUs acessar um *pool* de memória compartilhado em larga escala via CXL, com semântica nativa de leitura/escrita. O foco é o armazenamento da KV-Cache além da capacidade de DRAM do hospedeiro, entregando latência próxima à memória local.

Tang et al. (TANG et al., 2024), da ByteDance, exploram o uso de memória CXL para armazenar a KV-Cache em sistemas de *serving* (i.e., sistemas responsáveis por disponibilizar modelos treinados para atendimento de requisições de inferência em produção) de LLMs. Seus resultados mostram que a interconexão CXL–CPU oferece desempenho comparável à interconexão CPU–GPU em latência e largura de banda, permitindo aumento de 30% no tamanho de *batch* em relação a recomputação completa da KV-Cache.

PAM (LIU et al., 2025) propõe um sistema de *serving* que coordena dispositivos PIM heterogêneos em uma arquitetura hierárquica para operações centradas na KV-Cache, com computação distribuída entre HBM e memória CXL equipada com PIM.

3.4.1 Posicionamento

Esses são os trabalhos mais próximos desta dissertação no eixo de memória heterogênea. Fang et al. é particularmente relevante por oferecer a primeira formalização analítica do problema de posicionamento para LLMs. Entretanto, todos compartilham três limitações em relação ao escopo desta dissertação: (i) são centrados na GPU: a hierarquia parte da HBM da GPU; (ii) focam exclusivamente na KV-Cache: não abordam o posicionamento de pesos do modelo e ativações intermediárias; e (iii) dependem de hardware especializado (CXL-PIM, interconexões NVLink) não disponível em servidores.

3.5 Síntese e Posicionamento desta Dissertação

A Tabela 5 cruza um subconjunto representativo dos trabalhos discutidos neste capítulo ao longo de cinco dimensões relevantes para esta dissertação: unidade de computação, escopo do *offloading* (pesos e/ou KV-Cache), uso de memória heterogênea e, por último, os dispositivos envolvidos. Para concisão, a tabela inclui apenas os trabalhos que melhor ilustram cada combi-

Tabela 5 – Síntese comparativa de trabalhos representativos. ✓ indica suporte; — indica ausência.

Trabalho	Un. Comp.	Pesos	KV-Cache	Mem. Het.	Dispositivos
Redução de pegada (Seção 3.1)					
GPTQ / AWQ / GGUF	GPU/CPU	—	—	Não	—
Offloading centrado na GPU (Seção 3.2)					
FlexGen (SHENG et al., 2023)	GPU	✓	✓	Não	GPU→CPU→SSD
FlexInfer (NA et al., 2025)	**GPU+CPU**	✓	✓	Não	GPU↔CPU
KVDrive (LIN et al., 2025)	GPU	—	✓	Parcial	HBM→DRAM→SSD
Inferência apenas na CPU (Seção 3.3)					
llama.cpp (GERGANOV et al., 2023)	**CPU**	mmap	—	Não	DRAM↔storage
LLM Flash (ALIZADEH et al., 2024)	**CPU/GPU**	✓	—	Não	DRAM↔flash
FlexInfer (DU et al., 2025)	**CPU**	✓	✓	Não	DRAM↔SSD
CENT (GU et al., 2025)	**CPU (PIM)**	✓	✓	Parcial†	CXL-PIM customizado
Memória heterogênea para LLMs (Seção 3.4)					
Fang et al. (FANG et al., 2025)	GPU	—	✓	**Sim**	HBM↔DRAM
Beluga (YANG et al., 2025)	GPU	—	✓	**Sim**	HBM↔CXL
Esta dissertação	**CPU**	✓	✓	**Sim**	**DRAM↔PMem/CXL**

†CENT utiliza memória CXL, mas como substrato para unidades PIM customizadas, sem realizar posicionamento diferenciado de estruturas de dados entre camadas de memória.

nação de dimensões; trabalhos adicionais dentro de cada eixo (e.g., DAK, HybridGen, Kareto e Ganjihal na Seção 3.2; Tang et al. e PAM na Seção 3.4) são discutidos em suas respectivas seções e compartilham as mesmas limitações já identificadas.

A análise revela que nenhum trabalho existente combina simultaneamente:

1. Inferência **apenas em CPU**, sem dependência de GPU;
2. Posicionamento de **pesos do modelo e KV-Cache** em camadas diferenciadas;
3. Uso de **memória principal heterogênea** (DRAM + PMem ou DRAM + CXL), sem hardware customizado como PIM.

O trabalho que mais se aproxima em filosofia desta dissertação é o *LLM in a Flash* (ALIZADEH et al., 2024), que demonstra que modelar as características do meio de armazenamento e explorar esparsidade permitem ganhos substanciais de desempenho. Contudo, sua estratificação

opera entre DRAM e flash, com latências na ordem de μs a ms, e foca exclusivamente em pesos, sem contemplar a KV-Cache. Esta dissertação estende esse raciocínio para a camada de memória principal heterogênea, onde a assimetria entre DRAM e PMem/CXL apresenta latências na ordem de ns, permitindo posicionamento mais fino e abrangendo tanto pesos quanto KV-Cache.

Desta forma, esta dissertação preenche uma importante lacuna na literatura ao propor e avaliar estratégias de posicionamento de memória para inferência de LLMs em servidores com apenas CPU, equipados com memória heterogênea. Os capítulos seguintes apresentam a metodologia empregada na avaliação experimental (Capítulo 4) seguido dos resultados com as estratégias de posicionamento consideradas (Capítulo 5).

4 Metodologia

Este capítulo descreve a metodologia empregada na realização dos experimentos e na avaliação dos resultados reportados no Capítulo 5. Os experimentos práticos foram estruturados com o rigor necessário para isolar variáveis arquiteturais e de hardware, garantindo a reprodutibilidade e a precisão na medição dos gargalos de memória. O repositório com o código fonte e scripts para execução dos experimentos está disponível em domínio público [1].

No restante deste capítulo descrevemos o ambiente experimental (Seção 4.1), as estratégias de posicionamento dos dados consideradas no estudo (Seção 4.2) e os cenários de uso da LLM utilizados nos experimentos (Seção 4.3).

4.1 Ambiente Experimental

Todos os experimentos foram realizados em um ambiente com sistema operacional Linux de 64 bits, kernel 6.8, equipada com dois processadores Intel(R) Xeon(R) Gold 5317, de 12 cores cada, operando a 3.00GHz. O sistema conta com 256GB de memória DRAM DDR3 e 8 módulos de memória Intel Optane DC, série 200, perfazendo um total de 1TB. Para garantir rigor científico e mitigar a interferência de ruídos sistêmicos, o protocolo experimental adotou medidas estritas de controle de variáveis. Em particular, apenas um processador é usado (máximo de 12 cores) para evitar possíveis efeitos NUMA que potencialmente adicionariam ruídos aos experimentos. O modo *Turbo Boost* e *hyperthreading* também foram desabilitados. Antes de cada bateria de experimentos, uma execução de *warm-up* é realizada para garantir o preenchimento inicial das caches L1/L2/L3 e tabelas TLB (*Translation Lookaside Buffer*.

Os dados foram coletados ao longo de $N = 10$ repetições independentes para cada configuração (definidas a seguir). As métricas finais são reportadas através da média aritmética μ, acompanhadas do desvio-padrão amostral σ e do Intervalo de Confiança (IC) de 95%. Para as comparações entre configurações, adota-se como critério de distinguibilidade a não-sobreposição dos intervalos de confiança de 95%: duas configurações são consideradas estatisticamente distinguíveis quando seus ICs não se interceptam, e indistinguíveis caso contrário. Este critério é deliberadamente mais conservador que um teste-*t* pareado, adotado porque, dada a baixa variância das medições na fase de *decode* (desvios-padrão da ordem de 10^{-2} TPS), diferenças numericamente pequenas poderiam ser declaradas significativas sem possuírem relevância prática.

Distingue-se, portanto, ao longo da análise do próximo capítulo, entre significância estatística (não-sobreposição de ICs) e relevância operacional (magnitude suficiente para alterar

[1] Acessível em <https://github.com/luisa-leona/mestrado-llm-experimento>

Tabela 6 – Comparação arquitetural entre os modelos LLaMA 2 7B e LLaMA 3 8B empregados neste trabalho, ambos no formato de quantização `Q4_K_M`.

Característica	**LLaMA 2 7B**	**LLaMA 3 8B**
Parâmetros	7 bilhões	8 bilhões
Mecanismo de Atenção	Multi-Head (MHA)	Grouped-Query (GQA)
Janela de contexto	4.096 *tokens*	8.192 *tokens*
Tamanho do vocabulário	32.000	128.256
Tokenizer	SentencePiece (BPE)	Tiktoken (BPE)
Normalização	RMSNorm (pré-camada)	RMSNorm (pré-camada)
Ativação (MLP)	SwiGLU	SwiGLU
Cod. posicional	RoPE	RoPE
Quantização	`Q4_K_M`	`Q4_K_M`

Fontes: TOUVRON et al. e GRATTAFIORI et al..

decisões de alocação de memória); as conclusões apoiam-se sempre nesta última. Por fim, os resultados do próximo capítulo são reportados por meio das vazões das fases de *prefill* e *decode*, representadas em tokens por segundo (TPS).

4.1.1 Modelos empregados

Os experimentos realizados utilizam modelos LLaMA. A escolha do LLaMA como objeto de estudo justifica-se por três fatores diretamente alinhados aos objetivos deste trabalho. Primeiro, a disponibilidade de pesos abertos e inspecionáveis, condição indispensável para a manipulação manual de tensores e a contagem de parâmetros por bloco realizadas nos experimentos. Segundo, o suporte de primeira classe no ecossistema `llama.cpp`/GGML, que viabiliza o carregamento de versões quantizadas e o gerenciamento explícito de memória descritos na Seção 2.1.2. Terceiro, a existência da família em múltiplas escalas e gerações, o que permite contrastar diferentes configurações de modelos sob uma mesma metodologia.

Especificamente, este trabalho emprega o LLaMA 2 de 7 bilhões de parâmetros (TOUVRON et al., 2023b) e o LLaMA 3 de 8 bilhões de parâmetros (GRATTAFIORI et al., 2024), ambos no formato de quantização `Q4_K_M` (veja Tabela 6). Esta seleção não é incidental: os dois modelos representam um contraste arquitetural relevante para a análise de memória derivada na Seção 2.1.1. O LLaMA 2 7B emprega o mecanismo de Atenção *Multi-Head* clássico, ao passo que o LLaMA 3 8B adota o *Grouped-Query Attention* (GQA) em todos os seus tamanhos (GRATTAFIORI et al., 2024). Como o GQA reduz as matrizes de Chave e Valor, a fração de parâmetros associada à Atenção é menor no LLaMA 3, acentuando ainda mais a predominância do bloco *Feed-Forward* prevista pela Equação 2.3. Dessa forma, os dois modelos oferecem, respectivamente, uma realização antes e depois dessa otimização, dentro de uma mesma linhagem arquitetural.

Um segundo contraste para a ocupação de memória reside no tamanho do vocabulário. Enquanto o LLaMA 2 utiliza um *tokenizer* de 32.000 *tokens*, o LLaMA 3 amplia esse vocabulário

para 128.256 *tokens* (GRATTAFIORI et al., 2024). Esse crescimento quadruplica aproximadamente o número de parâmetros das matrizes de *embedding* e de projeção de saída, um termo tratado como secundário na Seção 2.1.1, mas que, no LLaMA 3 8B, deixa de ser desprezível na composição total da memória estática. Adicionalmente, a janela de contexto expande-se de 4.096 *tokens* no LLaMA 2 para 8.192 no LLaMA 3, com impacto direto sobre o crescimento máximo da KV-Cache.

Em conjunto, essas características fazem do par LLaMA 2 7B e LLaMA 3 8B um objeto de estudo adequado para investigar estratégias de alocação de memória: ambos são pequenos o suficiente para execução em ambientes restritos, porém arquiteturalmente distintos o bastante para que as variações na distribuição de parâmetros, predominância do bloco *Feed-Forward*, efeito do GQA e peso crescente do vocabulário, se manifestem de forma mensurável nos experimentos conduzidos neste trabalho.

4.1.2 Software de inferência

O software de inferência utilizado foi o `llama.cpp` (*commit* 47a268e), escolhido por sua maturidade na execução de modelos quantizados em CPU e por expor a estrutura interna dos tensores de forma acessível à instrumentação. O código original foi estendido neste trabalho para permitir a alocação seletiva dos principais blocos do modelo, pesos da rede de atenção, pesos da FFN/MLP e a KV-Cache, entre as memórias DRAM e Optane DC [2].

A extensão introduziu duas capacidades de controle da estratificação de memória, acionadas por novos argumentos de linha de comando:

- **Posicionamento dos pesos estáticos.** Por meio de uma sintaxe de configuração baseada em intervalos de camadas, é possível especificar, para cada camada (ou faixa de camadas), em qual memória residem os pesos da atenção e da FFN, de forma independente. Sufixos `.attn` e `.ffn` permitem o controle separado de cada subcomponente dentro de uma mesma camada, além do posicionamento das matrizes de *embedding* e de saída;
- **Posicionamento da KV-Cache.** Uma opção dedicada permite que a KV-Cache seja mantido em DRAM (comportamento padrão) ou alocada em Optane, desacoplando a sua localização da dos pesos estáticos.

Em ambos os casos, a residência em Optane é obtida mapeando a memória em um arquivo sobre um sistema de arquivos com suporte a DAX, utilizando `mmap`, que estabelece acesso direto à memória persistente, contornando o mecanismo de *page cache* do sistema operacional. Os tensores assim posicionados apontam diretamente para a região mapeada, de modo que a inferência ocorre integralmente em CPU, sem cópias intermediárias para a DRAM.

[2] As modificações foram desenvolvidas com auxílio do LLM Claude Opus 4.7, empregado como ferramenta de apoio à programação, e posteriormente revisadas e validadas experimentalmente pelo autor.

Tabela 7 – Estratégias de posicionamento usadas nos experimentos.

Config	KV-Cache	Pesos Atenção	Pesos FFN	Racional
Baseline	DRAM	DRAM	DRAM	Referência (tudo na DRAM)
KV	Optane	DRAM	DRAM	KV-Cache tolera Optane?
Attn	DRAM	Optane	DRAM	Atenção tolera Optane?
FFN	DRAM	DRAM	Optane	MLP tolera Optane?
Attn+FFN	DRAM	Optane	Optane	Só KV-Cache na DRAM
KV+Attn	Optane	Optane	DRAM	Só MLP na DRAM
KV+FFN	Optane	DRAM	Optane	Só atenção na DRAM
Tudo	Optane	Optane	Optane	Tudo no Optane (pior caso)

Ao final da fase de alocação, o software reporta a distribuição efetiva de memória entre DRAM e Optane, discriminada por categoria (atenção, FFN e demais tensores), o que permite verificar a conformidade da configuração aplicada.

Para garantir a reprodutibilidade dos experimentos, a amostragem operou em modo *Greedy Decoding* (temperatura fixada em 0,0) com semente constante (*seed* 0). A geração precoce do *token* de parada foi ignorada (`-ignore-eos`), obrigando a CPU a gerar a meta exata de *tokens* requerida em cada experimento, utilizando um *prompt* sintético padronizado.

4.2 Estratégias de posicionamento

Neste trabalho consideramos três elementos das LLMs para alocação: os pesos da FFN, os pesos da rede de atenção e a KV-Cache. Cada um desses elementos pode ser alocado na memória DRAM convencional ou nos módulos de memória Optane. Logo, há um total de 8 configurações possíveis, como representado na Tabela 7.

A primeira configuração, chamada de *Baseline*, serve de referência e mantém todos os elementos na memória DRAM. Teoricamente esta configuração é a que reporta o melhor desempenho. As configurações `KV`, `Attn` e `FFN` colocam cada um dos elementos, KV-Cache, pesos atenção, pesos FFN, respectivamente, na memória Optane. Elas permitem analisar isoladamente o impacto em se colocar cada elemento exclusivamente na Optane.

As configurações `Attn+FFN`, `KV+Attn` e `KV+FFN` sempre mantém um único elemento na DRAM, colocando os outros dois na Optane. Essas configurações permitem explorar uma relação de custo benefício com relação a desempenho e quantidade de memória DRAM utilizada, já que sempre mantemos apenas um dos três elementos da LLM nesta memória. Finalmente, a configuração `Tudo` é o extremo oposto da configuração `Baseline`, ou seja, aqui todos os elementos são alocados na memória Optane. Esta configuração portanto representa o pior cenário em termos de desempenho, já que todos os dados estariam em um dispositivo mais lento.

Tabela 8 – Cenários de uso para avaliação da estratificação de memória.

Cenário	Caso de uso	Foco do estresse	O que revela sobre estratificação	n_prompt	n_gen
C1	Sumarização, RAG	Leitura de cache grande e estável	Custo fixo por token com cache grande no Optane	4096	128
C2	Geração de texto, código	Cache crescendo continuamente	Degradação progressiva (curva de latência)	128	4096
C3	Chatbot, Q&A	Caso balanceado	Viabilidade prática para uso geral	1024	1024
C4	Multi-turno longo, RAG extenso	Máxima vazão de leitura	Pior caso realista / limite da estratificação	7000	1000
C5	Controle experimental	Nenhum (cache desprezível)	Gargalo fixo do Optane	64	64

4.3 Cenários de uso

Para responder às perguntas de pesquisa, criamos cinco cenários de uso que acreditamos cobrir uma vasta gama de uso das LLMs. Esses cenários são determinados por dois fatores principais: 1) tamanho do prompt (`n_prompt`) e 2) tamanho da resposta gerada (`n_gen`). A Tabela 8 mostra os diferentes cenários considerados, foco do estresse, qual fator da estratificação é revelado, e o valores utilizados para os três fatores principais na fase de inferência.

Os diferentes casos de uso permitem variar a proporção entre *prompt* e resposta dentro de um contexto fixo ou variável, expondo comportamentos distintos da KV-Cache. Como descrito anteriormente, o desempenho e comportamento do KV-Cache varia de acordo com a fase da inferência: *prefill* e *decode*. A sensibilidade à camada de memória depende de qual modo domina e do tamanho da KV-cache sendo acessada. Por isso, os cenários foram desenhados para cobrir combinações extremas e intermediárias dessa relação. Note ainda que, como o LLaMA 2 possui janela nativa de 4096 tokens, C4 é efetivamente truncado a 4096 tokens neste modelo, ao passo que o LLaMA 3 (janela de 8192) processa o contexto integral.

A seguir apresentamos mais detalhes que justificam a escolha de cada cenário.

4.3.1 C1 – *Prompt* Longo, Resposta Curta

Casos de uso reais representados por este cenário incluem sumarização de documentos, extração de entidades, classificação de textos longos, perguntas sobre documentos extensos (RAG com contexto grande). O *prompt* longo força o *prefill* a construir uma KV-Cache grande de uma só vez. Em contraste, a resposta é relativamente curta e há poucos passos de decodificação, mas cada passo lê uma KV-Cache grande.

Neste cenário, a KV-Cache praticamente não cresce durante a decodificação. Assim, permite observar o impacto de ler um KV-Cache grande e estável no Optane. Se o Optane degrada a leitura da cache, esse cenário captura o custo por token de *decode* de forma limpa, sem a variável confundidora do crescimento da cache.

4.3.2 C2 – *Prompt* Curto, Resposta Longa

Casos de uso reais representados por este cenário incluem geração de texto em geral, escrita de código, geração de relatórios e *chatbots* com respostas elaboradas. O *prompt* curto tem

como consequência uma fase de *prefill* rápida, com a KV-Cache inicial pequena. Em contraste, a resposta é longa e há muitos passos de decodificação, cada um lendo e expandindo a KV-Cache.

Este é o cenário que melhor revela degradação progressiva. Se a KV-Cache está no Optane, os primeiros tokens do decode leem pouca cache (rápido), mas os últimos leem mais entradas (lento). Isso permite investigar a latência por token ao longo do tempo e observar se o desempenho decresce.

4.3.3 C3 – Balanceado

Casos de uso reais representados por este cenário incluem *chatbot* conversacional, perguntas e respostas (Q&A), e assistentes virtuais. O tamanho do prompt (1024) permite perguntas com contexto moderado (ex: trecho de texto + pergunta). As respostas também possuem tamanho moderado, com decodificação de duração razoável. Assim, nem o *prefill* nem a decodificação dominam excessivamente o tempo de resposta.

Este cenário representa, de certa forma, o centro do espaço experimental em relação a ponderação entre *prefill* e decodificação, servindo como ponto de referência prático. Se a degradação é aceitável, a estratificação é viável para um caso de uso bastante comum. Também permite interpolar entre os extremos C1 e C2.

4.3.4 C4 – Contexto Quase Cheio

Casos de uso reais representados por este cenário incluem conversa multi-turno longa (*chatbot* com histórico extenso), RAG com muitos documentos no contexto, e análise de código-fonte grande. O tamanho do *prompt* é grande, representando quase o limite da janela de contexto. A resposta é moderada, mas com a KV-Cache quase no máximo durante toda a geração.

Esta configuração possui o *prefill* mais pesado de todos os cenários (7000 tokens). A decodificação opera com a KV-Cache quase máxima durante toda a geração e cada token requer a leitura de 7000-8000 entradas, estressando bastante a vazão de leitura da camada de memória. Este é o pior caso realista. Se a estratificação funciona aqui com degradação aceitável, funciona em qualquer cenário. Se não funciona, define o limite superior de uso prático. É o cenário que mais vai diferenciar DRAM de Optane para a KV-Cache, porque a quantidade de dados lidos por token é máxima.

4.3.5 C5 – Controle Mínimo

Este cenário não representa nenhum caso de uso real direto, sendo usado como controle experimental. O *prompt* curto representa um mínimo funcional, da mesma forma que a resposta. A KV-Cache é portanto desprezível em tamanho e quase todo o tempo é gasto em overhead fixo (carregamento, inicialização). Assim, este cenário serve para isolar o overhead fixo do Optane.

Se houver degradação mesmo com KV-Cache minúscula, o problema não é a cache e sim o custo fixo de acessar a memória Optane.

5 Resultados e Análise

Este capítulo apresenta os resultados experimentais obtidos a partir da avaliação das estratégias de estratificação seletiva de componentes da LLM entre DRAM e Optane. O objetivo principal é responder a pergunta principal de pesquisa desta dissertação, como já formulada na introdução: *Como a alocação seletiva de componentes internos de um LLM (pesos de atenção, pesos da FFN e KV-Cache) entre DRAM local e camadas de memória de maior capacidade e maior latência afeta o compromisso entre desempenho e uso de DRAM na inferência em ambientes sem aceleradores?*

A pergunta principal é, por natureza, bi-objetivo: confronta um eixo de desempenho (a vazão de inferência) com um eixo de uso de DRAM, e toda decisão de alocação é um ponto de compromisso entre ambos. Esses dois eixos, contudo, têm características distintas. O eixo de uso de DRAM é estrutural: o volume que cada componente ocupa decorre diretamente dos hiperparâmetros arquiteturais do modelo escolhido e pode ser estimado *a priori*, sem experimentação, como quantificado na Seção 5.5. O eixo de desempenho é essencialmente empírico: a degradação de vazão imposta pelo deslocamento de cada componente para a camada de maior latência é conhecida por medição. Para melhor caracterizar este eixo empírico, foram elencadas quatro questões de pesquisa (QP) secundárias:

- **QP1:** Ao alocar isoladamente cada componente na camada de maior latência, existe um componente sistematicamente mais tolerante, ou a hierarquia de custos depende do regime de operação?
- **QP2:** Como a sensibilidade da alocação varia em função do regime de operação, isto é, do tamanho do contexto e da proporção de *prefill* e *decode*?
- **QP3:** Como diferenças arquiteturais entre modelos (MHA do LLaMA 2 vs. GQA do LLaMA 3) alteram a tolerância de cada componente à estratificação?
- **QP4:** É possível identificar configurações combinadas de alocação que reduzam significativamente o uso de DRAM mantendo degradação aceitável?

A estratégia para responder à questão principal procede em três etapas. Primeiro, os dados coletados são analisados à luz das quatro perguntas secundárias QP1–QP4 (Seções 5.1, 5.2, 5.3 e 5.4), caracterizando o eixo empírico de desempenho. Em seguida, a Seção 5.5 caracteriza o eixo estrutural de uso de DRAM, estimando a ocupação de cada componente a partir dos hiperparâmetros arquiteturais. Por fim, a Seção 5.6 cruza os dois eixos por meio da análise de Pareto, consolida os padrões em uma visão integrada, deriva as diretrizes de alocação e responde à pergunta principal.

Tabela 9 – Vazão média (tokens/s) por cenário e configuração para o modelo LLaMA 2. Valores em média ± desvio-padrão.

Config.	C1	C2	C3	C4	C5
Decode TPS					
Baseline	7.7 ± 0.04	10.2 ± 0.03	11.1 ± 0.02	7.9 ± 0.03	15.3 ± 0.02
KV	4.1 ± 0.01	6.4 ± 0.01	7.6 ± 0.02	4.3 ± 0.02	14.7 ± 0.02
Attn	6.1 ± 0.01	7.4 ± 0.01	7.9 ± 0.01	6.2 ± 0.01	9.5 ± 0.01
FFN	4.8 ± 0.01	5.5 ± 0.01	5.8 ± 0.01	4.8 ± 0.01	6.7 ± 0.01
KV+Attn	3.6 ± 0.01	5.2 ± 0.01	5.9 ± 0.01	3.7 ± 0.00	9.3 ± 0.01
KV+FFN	3.1 ± 0.04	4.2 ± 0.03	4.6 ± 0.02	3.2 ± 0.04	6.6 ± 0.01
Attn+FFN	4.0 ± 0.01	4.5 ± 0.01	4.7 ± 0.00	4.0 ± 0.01	5.2 ± 0.01
Tudo	2.7 ± 0.02	3.6 ± 0.01	3.9 ± 0.01	2.8 ± 0.01	5.1 ± 0.01
Prefill TPS					
Baseline	28.6 ± 0.11	85.2 ± 0.46	69.9 ± 0.33	30.9 ± 0.06	83.7 ± 1.75
KV	10.4 ± 0.10	86.2 ± 0.31	65.2 ± 1.85	11.6 ± 0.12	83.3 ± 0.96
Attn	31.1 ± 0.08	86.7 ± 0.44	74.0 ± 0.23	33.5 ± 0.08	82.7 ± 0.96
FFN	30.1 ± 0.05	81.5 ± 0.18	70.1 ± 0.49	32.6 ± 0.04	77.3 ± 0.57
KV+Attn	10.8 ± 0.04	87.0 ± 0.35	70.1 ± 1.16	12.0 ± 0.10	83.1 ± 0.58
KV+FFN	10.7 ± 0.09	81.3 ± 0.25	65.0 ± 1.62	11.8 ± 0.15	77.2 ± 0.40
Attn+FFN	30.4 ± 0.06	81.1 ± 0.24	70.6 ± 0.20	32.8 ± 0.05	77.4 ± 0.50
Tudo	10.7 ± 0.03	80.8 ± 0.16	66.1 ± 1.60	11.9 ± 0.09	76.5 ± 0.95

Tabela 10 – Vazão média (tokens/s) por cenário e configuração para o modelo LLaMA 3. Valores em média ± desvio-padrão.

Config.	C1	C2	C3	C4	C5
Decode TPS					
Baseline	7.5 ± 0.02	9.6 ± 0.01	10.5 ± 0.02	5.5 ± 0.02	13.2 ± 0.01
KV	4.1 ± 0.02	6.1 ± 0.04	7.3 ± 0.00	2.6 ± 0.01	12.9 ± 0.02
Attn	6.8 ± 0.01	8.1 ± 0.01	8.7 ± 0.01	5.2 ± 0.01	10.3 ± 0.01
FFN	4.2 ± 0.00	4.7 ± 0.01	4.9 ± 0.01	3.5 ± 0.01	5.4 ± 0.01
KV+Attn	3.8 ± 0.02	5.5 ± 0.02	6.4 ± 0.00	2.5 ± 0.01	10.1 ± 0.01
KV+FFN	2.9 ± 0.01	3.7 ± 0.01	4.1 ± 0.01	2.1 ± 0.01	5.3 ± 0.01
Attn+FFN	3.9 ± 0.01	4.3 ± 0.01	4.4 ± 0.01	3.3 ± 0.01	4.8 ± 0.01
Tudo	2.6 ± 0.01	3.3 ± 0.01	3.6 ± 0.00	1.9 ± 0.00	4.5 ± 0.01
Prefill TPS					
Baseline	42.8 ± 0.34	80.8 ± 0.29	69.1 ± 0.21	23.1 ± 0.10	77.6 ± 1.87
KV	40.3 ± 2.24	80.9 ± 0.57	69.1 ± 0.18	11.1 ± 0.21	78.0 ± 2.21
Attn	46.7 ± 0.16	81.2 ± 0.98	72.7 ± 0.27	25.8 ± 0.07	77.2 ± 1.71
FFN	44.8 ± 0.25	75.5 ± 0.40	68.7 ± 0.11	25.0 ± 0.06	71.5 ± 0.30
KV+Attn	46.7 ± 2.86	82.3 ± 0.61	72.7 ± 0.21	12.1 ± 0.11	77.4 ± 2.19
KV+FFN	43.7 ± 2.74	75.4 ± 0.52	68.5 ± 0.25	11.9 ± 0.13	71.0 ± 1.13
Attn+FFN	45.0 ± 0.19	75.7 ± 0.15	68.7 ± 0.20	25.3 ± 0.04	71.4 ± 0.68
Tudo	43.0 ± 2.77	75.2 ± 0.17	68.6 ± 0.17	12.0 ± 0.11	70.7 ± 0.28

Antes de prosseguir, as Tabelas 9 e 10 oferecem uma visão panorâmica da vazão coletada para o LLaMA 2 e o LLaMA 3, servindo de referência para a aferição dos valores absolutos ao longo de todas as análises. Já é possível antecipar desse panorama que as técnicas de estratificação impõem custos de magnitudes distintas, que variam conforme a fase de inferência, o cenário de carga e o modelo avaliado.

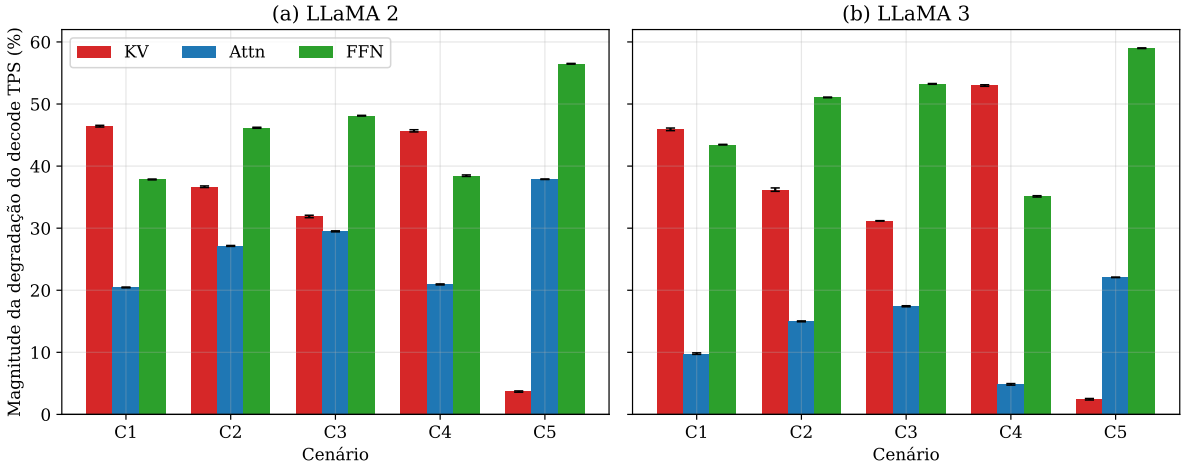


Figura 3 – Degradação percentual do *decode* TPS em relação à `Baseline` para as configurações isoladas (`KV`, `Attn`, `FFN`), por cenário.

5.1 QP1 – Existência de um componente universalmente mais tolerante

A primeira questão de pesquisa investiga se, ao alocar isoladamente cada componente da LLM na camada de maior latência, existe um componente sistematicamente mais tolerante à estratificação, ou se a hierarquia de custos entre eles depende do regime de operação. Foram consideradas as configurações `KV`, `Attn` e `FFN`, sempre em comparação com a configuração de referência `Baseline`. A análise é conduzida sobre a fase de *decode*, por duas razões. Primeiro, é nela que a KV-Cache é lida repetidamente a cada token gerado, de modo que o custo da estratificação se manifesta de forma recorrente e mensurável, tornando os componentes diretamente comparáveis entre si. Segundo, o *decode* é predominantemente limitado pela memória, condição em que a latência de acesso, exatamente a variável afetada pela alocação em Optane, determina a vazão; já o *prefill*, por ser majoritariamente limitado pela computação, amortiza esse custo e tende a obscurecer a hierarquia entre componentes.

O achado central da QP1 é negativo e, por isso mesmo, estruturante para todo o restante do capítulo: não existe um componente universalmente mais tolerante à estratificação. A ordenação de custos entre `KV`, `Attn` e `FFN` não é uma propriedade fixa de cada componente, mas depende do regime de operação. A Figura 3 e a Tabela 11 consolidam os cinco cenários para ambos os modelos; a leitura a seguir percorre o gradiente de contexto, dos extremos C5 (contexto mínimo) a C1/C4 (contexto longo), e expõe o mecanismo que produz a inversão da hierarquia.

Sob contexto curto (C5), a hierarquia é `KV` $<$ `Attn` $<$ `FFN`: isolar a KV-Cache custa apenas $-3{,}9\%$ no LLaMA 2 e $-2{,}3\%$ no LLaMA 3, tornando-a o componente mais tolerante, enquanto a FFN é o mais penalizador ($-56{,}2\%$ e $-59{,}1\%$, respectivamente). Isso se deve ao fato que, com contexto mínimo, o estado da KV-Cache é desprezível e quase não gera tráfego para a Optane, ao passo que os pesos de FFN, de tamanho fixo, são relidos integralmente a cada

Tabela 11 – Degradação relativa do *decode* TPS (%) das configurações isoladas face à `Baseline`, por cenário e modelo, com os cenários ordenados por contexto efetivo crescente. A componente *mais tolerante* em cada cenário aparece em **negrito**. A hierarquia rotaciona: a KV-Cache migra de mais tolerante (contexto curto) a menos tolerante (contexto longo).

Modelo	**Comp.**	**C5** curto	**C3**	**C2**	**C1**	**C4** longo
LLaMA 2	`KV`	**−3,9**	−31,5	−37,3	−46,8	−45,6
	`Attn`	−37,9	**−28,8**	**−27,5**	**−20,8**	**−21,5**
	`FFN`	−56,2	−47,7	−46,1	−37,7	−39,2
LLaMA 3	`KV`	**−2,3**	−30,5	−36,5	−45,3	−52,7
	`Attn`	−22,0	**−17,1**	**−15,6**	**−9,3**	**−5,5**
	`FFN`	−59,1	−53,3	−51,0	−44,0	−36,4

passo, independentemente do contexto.

Sob contexto longo (C1, C4), a ordem se inverte para `Attn` $<$ `FFN` $<$ `KV`: a Atenção torna-se a mais tolerante e a KV-Cache, a mais cara. A FFN, por sua vez, nunca ocupa a posição de mais tolerante, permanecendo intermediária ou pior em todos os cenários, o que a caracteriza como a candidata menos atraente à estratificação isolada em qualquer regime. Essa inversão entre os extremos é a evidência direta do achado central: qualquer conclusão sobre "o componente mais seguro de deslocar" extraída de um único cenário seria, por construção, enviesada.

Embora a inversão ocorra nos dois modelos, sua amplitude difere, e a diferença é informativa. A penalidade da `Attn` no LLaMA 3 é sistematicamente menor que no LLaMA 2 em todos os cenários, chegando a praticamente desaparecer sob contexto longo ($-5{,}5\%$ contra $-21{,}5\%$ em C4). Esse comportamento é compatível com a adoção do GQA no LLaMA 3, que compartilha um mesmo par chave/valor entre grupos de cabeças de *query*. O efeito primário do GQA é reduzir o tamanho da KV-Cache, mas ele também encolhe as matrizes de projeção W_K e W_V da camada de atenção, e são justamente esses pesos que a configuração `Attn` migra para a Optane. Com menos bytes a reler por passo, o custo de manter a Atenção na memória de maior latência cai, o que explica a tolerância quase total da `Attn` no LLaMA 3 em C4.

É importante notar que, embora o GQA reduza igualmente o volume da KV-Cache, esse benefício não se transfere para a configuração `KV` sob contexto longo. Diferentemente dos pesos de atenção, de tamanho fixo, o estado da KV-Cache é relido integralmente a cada token e cresce com o contexto. Quando este é extenso, o regime de limitação de memória domina e absorve a redução proporcionada pelo GQA. Por isso a degradação da `KV` em C4 é ainda maior no LLaMA 3 ($-52{,}7\%$) do que no LLaMA 2 ($-45{,}6\%$), apesar da KV-Cache mais enxuta. O benefício do GQA, portanto, manifesta-se na alocação dos pesos de atenção, não na do estado da KV-Cache, distinção que a QP3 (Seção 5.3) generaliza. A FFN, não afetada pelo GQA, permanece custosa em ambos os modelos.

Por fim, é importante notar que o comportamento observado não significa que os pesos

em `Attn` e `FFN` se tornam mais baratos sob contexto longo. O custo absoluto de relê-los a cada passo é aproximadamente fixo, pois independe do tamanho do contexto. O que muda é o referencial: a `Baseline` desacelera acentuadamente com o crescimento da KV-Cache (no LLaMA 3, de 13,2 tps em C5 para 5,5 tps em C4; Tabela 10), diluindo o peso relativo da penalidade fixa de `Attn` e `FFN`. Simultaneamente, o custo da `KV` cresce em termos absolutos, porque o volume de estado relido aumenta a cada token. A composição desses dois movimentos (penalidade relativa decrescente para os pesos e penalidade absoluta crescente para o estado da KV-Cache) é o mecanismo que inverte a hierarquia ao longo do gradiente de contexto.

5.1.1 Síntese

A QP1 estabelece, portanto, que a alocação isolada de componentes é viável, mas que não há um componente universalmente preferível: a inversão da hierarquia entre os extremos do gradiente de contexto prova que a escolha de menor risco é necessariamente condicionada ao regime de operação. Sob contexto curto, a KV-Cache é o componente mais seguro de deslocar para a Optane; sob contexto longo, esse papel passa aos pesos de Atenção, vantagem amplificada em arquiteturas GQA. A FFN situa-se sempre em posição intermediária a desfavorável, não sendo a melhor escolha em nenhum regime. Estabelecida a dependência da hierarquia em relação ao regime, a QP2 caracteriza como essa sensibilidade varia com o contexto e a fase de inferência.

5.2 QP2 – Variação da sensibilidade com o regime de operação

A QP1 estabeleceu que a hierarquia de tolerância entre componentes depende do regime de operação. A segunda questão de pesquisa caracteriza essa dependência: investiga como a sensibilidade da alocação varia em função do regime, isto é, do tamanho do contexto, da sua distribuição entre *prompt* e geração, e da proporção entre as fases de *prefill* e *decode*. Para isso, comparam-se os cinco cenários (C1–C5) em ambos os modelos, observando-se separadamente o comportamento em *prefill* e *decode*. As Figuras 4 e 5 apresentam, respectivamente, o *decode* e o *prefill* TPS por cenário, para a `Baseline` e as configurações isoladas. A discussão concentra-se na KV-Cache, por ser o único componente cuja penalidade é modulada pelo contexto no sentido próprio do termo: enquanto `Attn` e `FFN` têm pesos de tamanho fixo, o volume de estado da KV-Cache cresce a cada token gerado, o que a torna o eixo determinante da sensibilidade ao regime.

A Tabela 12 consolida a penalidade da configuração `KV` sobre o *decode*, organizando os cenários pelo tamanho do contexto. Os dados de ambos os modelos convergem para uma mesma leitura: a penalidade associada à KV-Cache não é constante, mas cresce de forma consistente com o volume de contexto efetivamente acessado durante a geração, ainda que sem monotonicidade estrita. No cenário de controle C5, em que a KV-Cache é desprezível, a degradação da configuração `KV` é inferior a 4% em ambos os modelos; à medida que o contexto

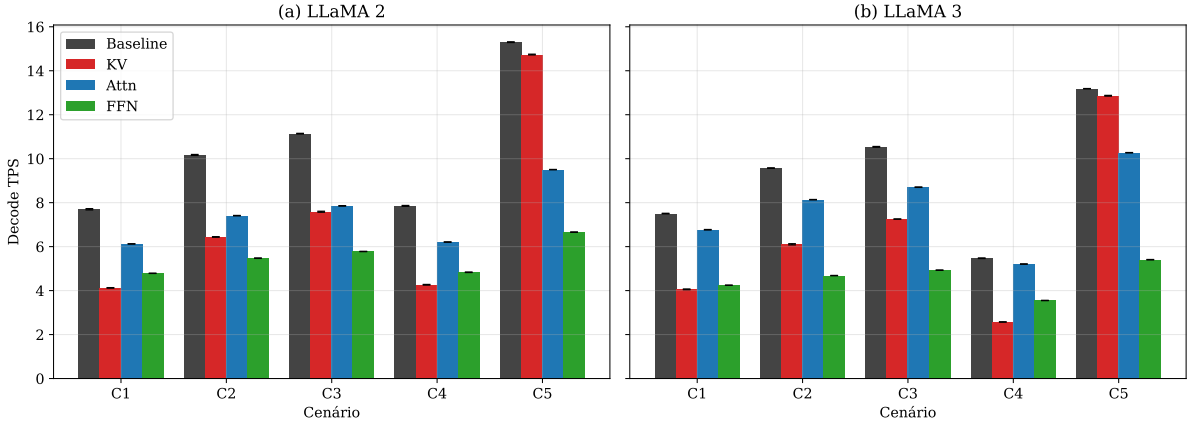


Figura 4 – *Decode* TPS por cenário (C1–C5) para a `Baseline` e as configurações isoladas, em LLaMA 2 e LLaMA 3.

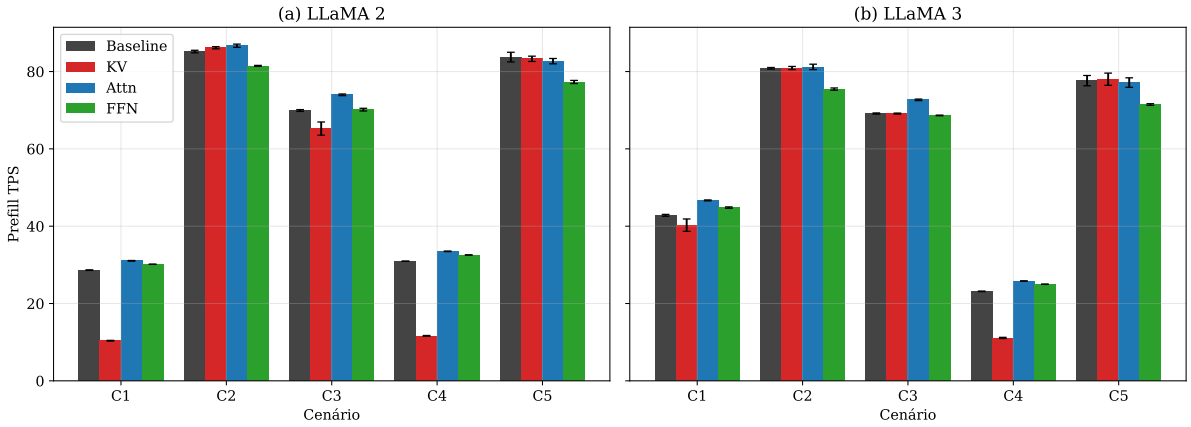


Figura 5 – *Prefill* TPS por cenário (C1–C5) para a `Baseline` e as configurações isoladas, em LLaMA 2 e LLaMA 3.

cresce (C3 $\rightarrow$ C2 $\rightarrow$ C1) ela se aproxima de -46%, de modo notavelmente consistente entre LLaMA 2 e LLaMA 3. A única quebra de monotonicidade ocorre entre C1 e C4 no LLaMA 2 ($-46{,}8\%$ e $-45{,}6\%$), valores estatisticamente indistinguíveis e coerentes com seu contexto efetivo equivalente ($\sim$4000 tokens).

O cenário de estresse C4 explicita o papel do contexto de maneira controlada: em LLaMA 2, configurado com $\sim$4000 tokens efetivos, a degradação é estatisticamente indistinguível da de C1; já em LLaMA 3, com $\sim$8000 tokens, a penalidade salta para $-52{,}7\%$. Isto é, sob contextos equivalentes, os dois modelos degradam de modo equivalente, e a diferença observada decorre exclusivamente do regime de contexto, não de uma propriedade intrínseca da arquitetura. Esse controle natural confirma que a variável de modulação dominante não é o componente alocado em si, mas a quantidade de dados lida a cada passo da geração.

A comparação entre C1 e C2 acrescenta uma nuance relevante e isola o efeito da distribuição do contexto. Os dois cenários possuem contexto efetivo equivalente, diferindo apenas na sua distribuição: C1 concentra o contexto no prompt (4096) com geração curta (128),

Tabela 12 – Penalidade da configuração `KV` sobre o *decode* TPS, por cenário e modelo. Os cenários são ordenados pelo tamanho do contexto (n_prompt + n_gen, conforme Tabela 8). A degradação cresce com o contexto acessado durante a geração, ainda que não estritamente monotônica.

			LLaMA 2		**LLaMA 3**	
Cenário	**n_prompt / n_gen**	**Ctx. approx.**	Base	Degr.	Base	Degr.
C5 (Controle)	64 / 64	$\sim$128	15,30	$-3{,}9\%$	13,18	$-2{,}3\%$
C3 (Conversação)	1024 / 1024	$\sim$2048	11,14	$-31{,}5\%$	10,54	$-30{,}5\%$
C2 (Geração)	128 / 4096	$\sim$4224	10,18	$-37{,}3\%$	9,57	$-36{,}5\%$
C1 (Sumarização)	4096 / 128	$\sim$4224	7,71	$-46{,}8\%$	7,50	$-45{,}3\%$
C4 (Estresse)	7000 / 1000	$\sim$8000†	7,86	$-45{,}6\%$	5,47	$-52{,}7\%$

† O contexto efetivo de C4 difere entre os modelos ($\sim$4000 em LLaMA 2; $\sim$8000 em LLaMA 3), o que explica a maior degradação observada apenas no segundo.

enquanto C2 inverte (prompt de 128, geração de 4096). Ainda assim, C1 degrada mais do que C2 ($-46{,}8\%$ contra $-37{,}3\%$ em LLaMA 2; $-45{,}3\%$ contra $-36{,}5\%$ em LLaMA 3). Como o volume total é o mesmo, a diferença isola um efeito de distribuição: quando a KV-Cache já é extensa desde o primeiro token gerado (C1), cada passo de *decode* relê um estado grande durante toda a geração; em C2, a cache parte de quase nada e só atinge seu tamanho máximo ao final, de modo que a leitura média por passo é menor. A variável governante, portanto, não é o tamanho final do contexto, mas a leitura média de estado por passo de geração.

5.2.1 Relação com o *prefill*

As Figuras 4 e 5 mostram que a análise por fase revela um comportamento mais complexo do que uma simples assimetria *prefill*/*decode*, e que difere entre os dois modelos. No LLaMA 3, em contexto moderado (C1, $\sim$4096 tokens), isolar a KV-Cache reduz o *prefill* em apenas $-5{,}8\%$ (42,8 $\rightarrow$ 40,3 tps), contra $-45{,}3\%$ no *decode*: o *prefill* processa o prompt de forma paralela e, enquanto o contexto cabe folgadamente na memória rápida, resiste à degradação. No LLaMA 2, ao contrário, o *prefill* já degrada $\sim -63\%$ em C1, indicando que esse modelo opera mais próximo do limite de memória mesmo em contexto moderado. A assimetria *prefill*/*decode*, portanto, não é universal e sim depende da margem de memória disponível em cada configuração modelo/dispositivo.

O cenário de estresse C4 torna esse mecanismo explícito no LLaMA 3. O *prefill* da `Baseline` já cai de 42,8 tps (C1) para 23,1 tps (C4) sem qualquer estratificação seletiva, evidenciando que um prompt de $\sim$7000 tokens, por si só, aproxima o modelo da saturação de memória. Nessas condições, isolar a KV-Cache elimina a folga remanescente e o *prefill* cai fortemente para 11,1 tps ($-51{,}9\%$), valor próximo ao do *decode* ($-53{,}0\%$). A convergência das duas fases sob contexto extenso não decorre de uma propriedade do componente isolado, mas do fato de que, uma vez ultrapassado o limiar de saturação, tanto o *prefill* quanto o *decode* passam a ser

dominados pela latência de acesso à memória secundária.

5.2.2 Síntese

Em síntese, a QP2 caracteriza a sensibilidade que a QP1 havia identificado: a penalidade da estratificação é governada pela leitura média de estado por passo de geração, e não pelo componente ou pelo tamanho final do contexto tomados isoladamente. Disso decorre que a viabilidade da estratificação está fortemente condicionada a três fatores: o regime de contexto, a fase de inferência e a margem de memória do par modelo/dispositivo. A estratégia mostra-se mais eficiente em aplicações de contexto curto (assistentes interativos, respostas breves), mas torna-se progressivamente menos eficiente à medida que o contexto cresce e o sistema se aproxima da saturação, ponto a partir do qual nem mesmo a paralelização do *prefill* ajuda.

5.3 QP3 – Efeito das diferenças arquiteturais

A terceira questão de pesquisa desloca o foco do componente individual, objeto das QP1 e QP2, para a interação entre a arquitetura do modelo e a estratificação. Enquanto as seções anteriores caracterizaram, dentro de cada modelo, qual componente é mais tolerante (QP1) e como essa tolerância varia com o regime de operação (QP2), a QP3 pergunta o que muda quando se troca a arquitetura: LLaMA 2, com atenção multi-cabeça plena (MHA), e LLaMA 3, com *Grouped-Query Attention* (GQA), vocabulário ampliado e janela de contexto maior.

A chave para interpretar todos os efeitos a seguir é uma única distinção física, já isolada nas QP1 e QP2: a KV-Cache é um estado dinâmico, relido integralmente a cada token, enquanto os pesos de Atenção e FFN têm volume fixo, independente do contexto. O GQA atua sobre ambos, mas de formas cujas consequências são opostas. Ao compartilhar um mesmo par chave/valor entre grupos de cabeças de *query*, ele encolhe simultaneamente o estado da KV-Cache e as matrizes de projeção W_K/W_V da camada de atenção. Sobre os pesos, de volume fixo, essa redução se traduz diretamente em menos bytes relidos por passo. Sobre o estado, ela é neutralizada: sob o regime limitado por memória que domina o *decode*, o que governa o custo não é o tamanho absoluto do estado, mas o fato de ele ser relido a cada token. O GQA, portanto, não torna o LLaMA 3 uniformemente mais tolerante à estratificação, ele redistribui a tolerância, barateando os pesos e, relativamente, encarecendo o estado. As três subseções seguintes mostram essa redistribuição manifestando-se em três planos: na hierarquia de componentes durante o *decode*, num paradoxo de projeto em torno do próprio GQA, e na assimetria entre as fases de inferência.

5.3.1 A hierarquia de componentes

O primeiro plano é a forma da hierarquia no *decode*, e não apenas suas magnitudes. Como evidencia a comparação lado a lado da Figura 3, e a partir dos valores consolidados na Tabela 11 (Seção 5.1), no LLaMA 2 os três componentes ocupam um intervalo relativamente estreito

($-21{,}5\%$ a $-45{,}6\%$ em C4). No LLaMA 3, a redistribuição descrita acima polariza esse intervalo: o barateamento dos pesos W_K/W_V faz a configuração `Attn` praticamente desaparecer como fonte de custo ($-5{,}5\%$), enquanto a `KV` se agrava ($-52{,}7\%$). A distância entre o componente mais tolerante e o menos tolerante salta de $\sim$24 para $\sim$47 pontos percentuais.

A consequência prática é que a política de alocação ótima é dependente da arquitetura, e de um modo qualitativo, não apenas quantitativo. Em MHA, nenhum componente é claramente o mais indicado para ser movido para a Optane, embora os pesos da Atenção ainda possuam uma leve vantagem; a decisão é um compromisso de margens estreitas. Já em GQA, a hierarquia se torna nítida: a Atenção emerge como candidata natural à estratificação agressiva, ao passo que a KV-Cache se torna o componente que mais se deve preservar na memória rápida. O GQA, ao redistribuir, também desambigua a decisão de alocação.

5.3.2 O paradoxo do *Grouped-Query Attention*

O segundo plano eleva a redistribuição a um resultado contraintuitivo de projeto. O GQA é uma técnica concebida para reduzir a KV-Cache; seria natural supor que, tendo uma KV-Cache mais enxuta, o LLaMA 3 tolerasse melhor a estratificação justamente desse estado. Os dados mostram o oposto: o LLaMA 3 não é mais tolerante à estratificação da `KV` que o LLaMA 2, ele degrada mais ($-52{,}7\%$ contra $-45{,}6\%$ em C4). O mecanismo é o já enunciado: o estado reduzido continua sendo relido a cada token, e o regime limitado por memória absorve integralmente o ganho de volume. O benefício da técnica não desaparece; ele migra, materializando-se nos pesos de volume fixo que a configuração `Attn` desloca.

A leitura de engenharia que a QP3 acrescenta é a generalização desse paradoxo. O argumento não depende de *como* o GQA reduz a KV-Cache, mas apenas de duas propriedades que ele compartilha com toda uma família de técnicas de compressão de estado: o estado é relido a cada passo de *decode*, ao passo que os pesos têm volume fixo; e encolher o estado não altera essa releitura por token, de modo que, sob regime limitado por memória, o ganho de volume satura. Disso decorre uma diretriz que extrapola o caso estudado: qualquer técnica cujo objetivo seja reduzir o tamanho da KV-Cache tende a oferecer mais retorno quando se estratificam os pesos de atenção do que quando se estratifica o próprio estado que ela visa encolher. O alvo natural da estratificação, sob compressão de estado, desloca-se do estado para os pesos.

5.3.3 A assimetria entre fases

O terceiro plano manifesta-se na interação entre arquitetura e fase de inferência, capturada no contraste entre as Figuras 4 e 5. Sob contexto moderado (C1), isolar a `KV` degrada o *prefill* do LLaMA 2 em $-63{,}6\%$ (28,6 $\rightarrow$ 10,4 tps; Tabela 9), enquanto o do LLaMA 3 resiste com apenas $-5{,}8\%$ (42,8 $\rightarrow$ 40,3 tps; Tabela 10), para um *decode* de degradação comparável em ambos ($\sim -46\%$). A QP2 havia mostrado que a proteção conferida pela paralelização do *prefill*

depende da margem de memória do par modelo/dispositivo; a QP3 identifica a origem dessa margem: em contexto moderado, ela é um produto da arquitetura. A KV-Cache mais enxuta do GQA mantém o LLaMA 3 distante da saturação durante o processamento paralelo do prompt, ao passo que o LLaMA 2, com MHA, já opera próximo do limite e perde essa proteção.

Essa proteção arquitetural, contudo, não é incondicional: ela é uma margem finita que o GQA amplia, mas que o crescimento do contexto consome. Sob o cenário de estresse C4 ($\sim$8000 tokens no LLaMA 3), a folga adicional conferida pelo GQA se esgota, e o *prefill* do LLaMA 3 também colapsa ao isolar a `KV` (23,1 $\rightarrow$ 11,1 tps, $-51{,}9\%$), aproximando-se do comportamento do LLaMA 2 (30,9 $\rightarrow$ 11,6 tps, $-62{,}5\%$). A vantagem do GQA torna-se então apenas relativa (queda de 52% contra 63%), não mais protetora em termos absolutos. A assimetria *prefill*/*decode*, portanto, não é uma propriedade fixa nem da fase nem da arquitetura isoladamente: ela emerge da interação entre a estrutura da fase (escrita paralela única no *prefill vs.* releitura sequencial no *decode*), a arquitetura de atenção (que dimensiona a margem inicial) e o regime de contexto (que a consome). O GQA desloca o limiar de contexto a partir do qual o *prefill* deixa de ser protegido, mas não o elimina.

5.3.4 Síntese

As diferenças arquiteturais, portanto, não são marginais nem se reduzem a um reescalonamento das penalidades observadas nas QP1 e QP2. Todas as suas manifestações decorrem de um único mecanismo: o GQA encolhe pesos de volume fixo e estado dinâmico simultaneamente, mas apenas o barateamento dos pesos sobrevive ao regime limitado por memória do *decode*. Desse mecanismo de *redistribuição* seguem três consequências: (i) a hierarquia de tolerância entre componentes se abre e se desambigua, ampliando a distância entre Atenção e KV-Cache de $\sim$24 para $\sim$47 pp (Tabela 11); (ii) o benefício de técnicas como o GQA migra do estado dinâmico para os pesos estáticos, invertendo a intuição de projeto e com implicações que extrapolam o caso específico estudado; e (iii) a arquitetura desloca o limiar de contexto a partir do qual a fase de *prefill* deixa de oferecer proteção sob estratificação, sem contudo eliminá-lo (Figuras 4 e 5). Para o projeto de sistemas de inferência em memória heterogênea, isso implica que a política de alocação não pode ser definida apenas em função do componente ou do regime de operação, mas deve incorporar a arquitetura do modelo como variável de primeira ordem.

5.4 QP4 – Configurações combinadas

A quarta questão de pesquisa investiga se existem configurações combinadas capazes de reduzir significativamente o uso de DRAM mantendo degradação aceitável. São examinadas as quatro combinações `KV+Attn`, `Attn+FFN`, `KV+FFN` e `Tudo`, comparadas à `Baseline` integralmente em DRAM e aos componentes isolados avaliados nas QP1–QP3. A análise procede em dois passos: primeiro, verifica-se se alguma combinação consegue degradar menos do que os

componentes que a compõem; em seguida, comparam-se as combinações entre si para identificar a de menor custo em cada regime de contexto.

5.4.1 Combinações vs. componentes isolados

Pelas Tabelas 9 e 10, no *decode*, nenhuma configuração combinada atinge vazão superior à do melhor dos seus componentes isolados: deslocar dois ou mais blocos para a Optane jamais produz desempenho que compense a carga adicional, e cada componente acrescenta um custo próprio. Esse padrão verifica-se em todos os cenários e em ambos os modelos, e tem consequência prática direta: quando a economia de DRAM obtida por um único componente já é suficiente, adicionar um segundo à Optane só pode piorar a vazão de decodificação.

Há, todavia, casos em que a combinação degrada menos do que um componente isolado que não a compõe. No LLaMA 3 em C2, por exemplo, `KV+Attn` degrada $-42{,}7\%$, menos do que a FFN *isolada* ($-51{,}0\%$). Embora não contradiga a regra anterior (a KV e a Atenção, tomadas individualmente, degradam bem menos que a FFN nesse cenário), o resultado é operacionalmente relevante: mostra que a natureza dos componentes deslocados importa mais do que a sua quantidade, e que uma combinação leve pode ser preferível a um único componente pesado.

No *prefill*, as diferenças entre combinações e componentes isolados são, em sua maioria, estatisticamente indistinguíveis (margens inferiores a 1σ). Os raros casos em que a combinação aparenta superar o isolado (e.g., LLaMA 3 em C4: `Attn+FFN` = 25,3 vs. FFN = 25,0) representam ganhos $< 1\%$ do *baseline*, compatíveis com variabilidade experimental. Esse comportamento é coerente com a natureza limitada pela computação do *prefill*: como os acessos à memória são amortizados pela reutilização intensiva dos pesos durante a multiplicação matricial do *prompt*, a localização dos tensores (DRAM vs. Optane) tem impacto marginal.

5.4.2 Comparação entre combinações

Estabelecido que nenhuma combinação supera seus componentes isolados no *decode*, restaria concluir que combinar componentes é sempre contraproducente. Essa leitura, porém, ignora a restrição que motiva a estratificação em primeiro lugar: o orçamento de DRAM. Quando a economia obtida ao deslocar um único componente é insuficiente para acomodar o modelo na memória disponível, deslocar dois ou mais deixa de ser uma escolha e torna-se uma imposição do hardware. Nesse cenário, a questão prática reduz-se a: dado que dois ou mais componentes precisam ser deslocados, qual combinação minimiza a degradação?

A Figura 6 e a Tabela 13 consolidam essa análise. A Figura 6 exibe a degradação relativa do decode TPS de todas as combinações por cenário e modelo; a Tabela 13 isola as duas combinações disputantes ao ponto ótimo, `KV+Attn` e `Attn+FFN`, para evidenciar o comportamento.

Da Figura 6 emergem dois padrões invariantes em todos os cenários e em ambos os

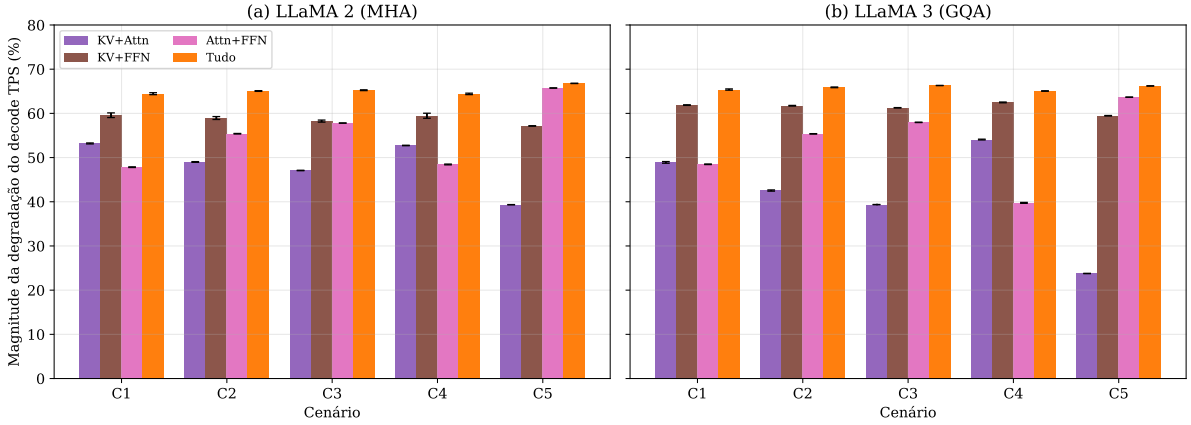


Figura 6 – Magnitude da degradação relativa do *decode* TPS (%) das configurações combinadas, por cenário.

Tabela 13 – Magnitude da degradação relativa do *decode* TPS (%) das duas combinações disputantes ao ponto ótimo, por cenário e modelo. A combinação menos penalizadora (*em negrito*) migra de `KV+Attn`, sob contexto curto, para `Attn+FFN`, sob contexto longo.

		LLaMA 2		**LLaMA 3**	
Cenário	**Contexto**	`KV+Attn`	`Attn+FFN`	`KV+Attn`	`Attn+FFN`
C5 (Controle)	∼128	**−39,2%**	−66,0%	**−23,5%**	−63,6%
C3 (Conversação)	∼2048	**−46,8%**	−57,7%	**−39,0%**	−58,1%
C2 (Geração)	∼4224	**−49,0%**	−55,9%	**−42,7%**	−55,2%
C1 (Sumarização)	∼4224	−53,2%	**−48,1%**	−49,3%	**−48,0%**
C4 (Estresse)	∼8000	−53,2%	**−49,4%**	−54,5%	**−40,0%**

modelos. Primeiro, a configuração `Tudo` é sistematicamente a mais penalizadora. Segundo, `KV+FFN` ocupa a segunda faixa de maior degradação, de modo que qualquer combinação que desloque a KV-Cache junto à FFN concentra as maiores perdas, com a única exceção do cenário de controle C5, onde a KV-Cache é desprezível.

A disputa pelo ponto ótimo dá-se entre as duas combinações remanescentes, `KV+Attn` e `Attn+FFN`, e o melhor candidato depende do contexto (Tabela 13). Sob contexto curto a moderado (C5, C3 e C2), `KV+Attn` é consistentemente menos penalizadora: no LLaMA 3 em C5, por exemplo, degrada apenas −23,5%, contra −63,6% de `Attn+FFN`. O motivo é que com a KV-Cache desprezível, deslocá-la para a Optane custa quase nada, e manter a FFN em DRAM evita o custo fixo de reler seus pesos volumosos.

A inversão consuma-se em C1: a partir de ∼4224 tokens com prompt longo, `Attn+FFN` torna-se preferível porque preservar a KV-Cache em DRAM evita o gargalo que, sob contexto extenso, domina o custo de qualquer combinação que a inclua. Sob C4 (∼8000 tokens), a vantagem de `Attn+FFN` consolida-se e, no LLaMA 3, atinge sua maior amplitude (−40,0% contra −54,5%; ∼14,5 pp).

Tabela 14 – Estimativa do tamanho de cada bloco de componentes em memória. Os pesos (Atenção e FFN) são estimados sob `Q4_K_M` (≈4,5 bits/parâmetro); a KV-Cache é estimada em FP16 para os cenários experimentais (batch = 1). O total do modelo quantizado (`.gguf`) inclui também *embeddings* e demais tensores auxiliares. As estimativas condizem com o que foi verificado empiricamente durante a experimentação.

Componente	**LLaMA 2 7B**	**LLaMA 3 8B**
Pesos de Atenção (Q, K, V, O)	≈1,15 GB	≈0,72 GB
Pesos da FFN (gate, up, down)	≈2,32 GB	≈3,02 GB
KV-Cache (FP16, batch = 1):		
C5 (∼128 tokens)	≈64 MB	≈16 MB
C3 (∼2 048 tokens)	≈1,0 GB	≈0,25 GB
C2/C1 (∼4 224 tokens)	≈2,1 GB	≈0,52 GB
C4 (∼8 000 tokens)	≈2,1 GB	≈0,98 GB

5.4.3 Síntese

O resultado central da QP4 é, portanto, que não existe uma combinação universalmente dominante: a escolha de menor degradação muda com o regime de contexto, espelhando, no nível combinado, a hierarquia isolada estabelecida na QP1. A regra operacional não é a escolha de uma combinação fixa, mas uma política condicionada ao contexto que preserva em DRAM o componente que constitui o gargalo daquele regime: a KV-Cache quando o contexto é longo, os pesos da FFN quando é curto.

5.5 Ocupação de memória dos componentes

As seções anteriores caracterizaram o eixo empírico de desempenho. Esta seção caracteriza o segundo eixo do problema bi-objetivo: o uso de DRAM. Diferentemente da vazão, que exige medição, a ocupação de memória de cada componente é *estrutural*, já que decorre diretamente dos hiperparâmetros arquiteturais e pode ser estimada *a priori*. Quantificá-la é o que permitirá, na Seção 5.6, confrontar economia de DRAM e degradação de vazão em um único critério de decisão.

A Tabela 14 estima o tamanho de cada bloco de componentes a partir dos hiperparâmetros arquiteturais dos modelos (Tabela 6, Capítulo 4), considerando a quantização `Q4_K_M` (≈4,5 bits/parâmetro) para os pesos estáticos e FP16 (16 bits) para a KV-Cache[1].

A tabela expõe três observações sobre as magnitudes dos blocos:

1. **A FFN é o maior bloco estático em ambos os modelos**, representando aproximadamente o dobro dos pesos de atenção no LLaMA 2 e mais de quatro vezes no LLaMA 3. Em

[1] O `llama.cpp` armazena a KV-Cache em FP16 por padrão na versão utilizada nos experimentos. Formatos quantizados de KV-Cache (e.g., Q8_0, Q4_0) estão disponíveis em versões mais recentes, mas não foram empregados neste trabalho.

termos de pesos, deslocar a FFN para a Optane é a ação de maior impacto sobre a economia de DRAM, mas é também a de maior custo de vazão sob contexto curto (QP1);

2. **A KV-Cache é desprezível sob contexto curto, mas rivaliza com os pesos sob contexto longo.** No LLaMA 2 em C4, a KV-Cache (≈2,1 GB) aproxima-se da FFN (≈2,32 GB) sem, contudo, superá-la, configurando um regime em que estado dinâmico e pesos estáticos têm magnitudes comparáveis. No LLaMA 3, o GQA reduz a KV-Cache, mantendo-a abaixo de 1 GB mesmo em C4, uma vantagem arquitetural que amplia a margem para mantê-la em DRAM;

3. **O GQA redistribui o peso relativo dos blocos.** No LLaMA 3, a Atenção pesa apenas ≈0,72 GB, de modo que deslocá-la para a Optane libera pouca DRAM; contudo, seu custo de vazão também é baixo (QP2). No LLaMA 2, os pesos de atenção (≈1,15 GB) são mais volumosos e seu deslocamento oferece economia mais significativa, porém com custo de vazão proporcionalmente maior.

A Tabela 15 cruza essas estimativas com as configurações combinadas, quantificando a DRAM liberada e a degradação correspondente para cada combinação nos cenários extremos. A leitura conjunta das duas metades da tabela ilustra o dilema central da estratificação. Em C5, a KV-Cache libera menos de 100 MB, tornando seu deslocamento praticamente inútil em termos de DRAM; a FFN é o alvo mais produtivo, mas custa ∼56–59% de vazão. Em C4, o quadro se inverte quanto ao custo, mas não quanto à economia: a KV-Cache, antes gratuita, passa a custar ∼46% (LLaMA 2) a ∼53% (LLaMA 3) de vazão, ao passo que a FFN se torna comparativamente mais barata (∼36–39%). Crucialmente, esse encarecimento da KV-Cache não é compensado por uma economia de DRAM superior: no LLaMA 2, a KV-Cache em C4 libera ∼2,1 GB, valor inferior aos ∼2,3 GB da FFN; no LLaMA 3, o GQA reduz a KV a apenas ∼1 GB, contra ∼3 GB da FFN. Em ambos os modelos, portanto, sob contexto longo a FFN domina a KV-Cache como alvo de estratificação: libera DRAM igual ou maior a um custo de vazão estritamente menor. A KV-Cache, que sob contexto curto era o deslocamento mais seguro, torna-se sob contexto extenso a escolha menos atraente em ambas as arquiteturas.

5.6 Síntese e resposta à questão principal

As Seções 5.1–5.4 caracterizaram o eixo de desempenho e a Seção 5.5 o eixo de uso de DRAM, ambos isoladamente. Esta seção reúne os dois e responde à questão principal. Para tanto, formaliza primeiro o compromisso bi-objetivo por meio da fronteira de Pareto (Seção 5.6.1); consolida os padrões resultantes em uma visão integrada (Seção 5.6.2); traduz os achados em diretrizes prescritivas (Seção 5.6.3); e, por fim, responde à pergunta central da dissertação (Seção 5.6.4).

Tabela 15 – DRAM liberada e degradação de decode TPS (%) por configuração combinada, nos cenários extremos C5 (contexto curto) e C4 (contexto longo). A coluna "DRAM lib." soma os componentes deslocados para a Optane; para a KV-Cache, usa-se o valor do cenário correspondente.

	LLaMA 2 7B		**LLaMA 3 8B**	
Configuração	**DRAM lib.**	**Degr. (%)**	**DRAM lib.**	**Degr. (%)**
Cenário C5 (∼128 tokens)				
KV	0,06 GB	−3,9	0,02 GB	−2,3
Attn	1,15 GB	−37,9	0,72 GB	−22,0
FFN	2,32 GB	−56,2	3,02 GB	−59,1
KV+Attn	1,21 GB	−39,2	0,74 GB	−23,5
Attn+FFN	3,47 GB	−66,0	3,74 GB	−63,6
KV+FFN	2,38 GB	−56,9	3,04 GB	−59,8
Tudo	3,53 GB	−66,7	3,76 GB	−65,9
Cenário C4 (∼8 000 tokens)				
KV	2,1 GB	−45,6	0,98 GB	−52,7
Attn	1,15 GB	−21,5	0,72 GB	−5,5
FFN	2,32 GB	−39,2	3,02 GB	−36,4
KV+Attn	3,25 GB	−53,2	1,70 GB	−54,5
Attn+FFN	3,47 GB	−49,4	3,74 GB	−40,0
KV+FFN	4,42 GB	−59,5	4,00 GB	−61,8
Tudo	5,57 GB	−64,6	4,72 GB	−65,5

5.6.1 Compromissos entre DRAM e vazão

As Seções 5.1–5.4 e 5.5 trataram a degradação de vazão e a economia de DRAM como dimensões separadas. A decisão de alocação, contudo, é intrinsecamente bi-objetivo: deseja-se simultaneamente maximizar a DRAM liberada e a vazão de *decode* (equivalentemente, minimizar a degradação). Esta subseção formaliza esse compromisso por meio do conceito de *eficiência de Pareto*, convertendo a leitura qualitativa da Tabela 15 em um critério de seleção.

Diz-se que uma configuração A *domina* uma configuração B quando A não é pior em nenhum dos dois objetivos e é estritamente melhor em ao menos um:

$$A \succ B \iff \big(\mathrm{DRAM}_A \geq \mathrm{DRAM}_B \wedge \mathrm{TPS}_A > \mathrm{TPS}_B\big) \vee \big(\mathrm{DRAM}_A > \mathrm{DRAM}_B \wedge \mathrm{TPS}_A \geq \mathrm{TPS}_B\big). \tag{5.1}$$

A *fronteira de Pareto* é o conjunto das configurações não dominadas por nenhuma outra: são as escolhas para as quais não se pode liberar mais DRAM sem sacrificar vazão, nem ganhar vazão sem devolver DRAM. Configurações dominadas são estritamente inferiores e podem ser descartadas *a priori*, pois existe sempre uma alternativa que entrega mais memória por igual ou menor custo.

A dominância pura, entretanto, é insensível à utilidade prática do ponto: uma configuração pode ser não dominada apenas por ocupar uma faixa de DRAM em que nenhuma outra opera, ainda que sua vazão seja baixa demais para uso real. Para evitar que a fronteira recomende

Tabela 16 – Pertinência à fronteira de Pareto por configuração e cenário, em ambos os modelos. ✓: Pareto-eficiente; ×: dominada (existe configuração com mais DRAM liberada e vazão igual ou maior); ∅: inviável (decode TPS $< 3{,}0$ tok/s). DRAM liberada conforme Tabela 15; vazão conforme Tabelas 9 e 10.

	LLaMA 2 (MHA)					**LLaMA 3 (GQA)**				
Config.	C5	C3	C2	C1	C4	C5	C3	C2	C1	C4
`KV`	✓	×	✓	×	×	✓	×	×	×	∅
`Attn`	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
`FFN`	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
`KV+Attn`	✓	✓	✓	×	×	✓	✓	✓	×	∅
`KV+FFN`	✓	×	✓	✓	✓	✓	×	×	∅	∅
`Attn+FFN`	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
`Tudo`	✓	✓	✓	∅	∅	✓	✓	✓	∅	∅

configurações inviáveis, introduzimos um *limiar de aceitabilidade* sobre a vazão de *decode*. Adotamos TPS $\geq 3{,}0$ tokens/s como piso mínimo, abaixo do qual a geração é considerada inaceitavelmente lenta para uso interativo[2]. Configurações abaixo desse piso são excluídas da fronteira por inviabilidade, independentemente de serem ou não dominadas.

A Tabela 16 aplica os dois critérios, dominância de Pareto (Eq. 5.1) e corte de aceitabilidade, a cada par (modelo × cenário), classificando cada configuração como *eficiente* (✓), *dominada* (×) ou *inviável* (∅, abaixo do piso de vazão). A Figura 7 representa graficamente as dez fronteiras, uma por par (modelo × cenário), no plano (DRAM liberada, *decode* TPS). A linha tracejada horizontal marca o piso de aceitabilidade; os pontos sobre a fronteira são ligados por um traço contínuo, e os dominados aparecem em tom esmaecido.

A leitura da Tabela 16 revela três padrões que conferem caracter quantitativo à política sensível ao contexto derivada das QP1–QP4:

1. **Os componentes isolados `Attn` e `FFN` são Pareto-eficientes em todos os dez casos.** Nenhuma configuração combinada os domina: ou liberam DRAM a um custo de vazão que nenhuma combinação iguala, ou ocupam o joelho da fronteira. A preferência por um único componente na Optane não é, portanto, uma heurística, mas uma consequência da dominância: deslocar um segundo componente quase sempre move o ponto para baixo (menor TPS) sem ganho proporcional de DRAM que o mantenha não dominado;

2. **O corte de aceitabilidade elimina sistematicamente as configurações que deslocam a KV-Cache sob contexto extenso.** Sob contexto longo, `Tudo` torna-se inviável (∅) em ambos os modelos (C1 e C4), e no LLaMA 3 o corte remove também `KV+FFN`, `KV+Attn`

[2] A literatura de sistemas de inferência situa o piso de vazão de *decode* na velocidade de leitura humana confortável, da ordem de 5 tokens/s ($\approx$ 250 palavras/min a $\sim$ 1,3 token/palavra) (ZHONG et al., 2024). Adotamos 3 tokens/s como piso tolerável, em que o texto surge mais devagar que a leitura, mas ainda acompanhável, e não como alvo de conforto, escolha conservadora coerente com a ausência de aceleradores no ambiente avaliado.

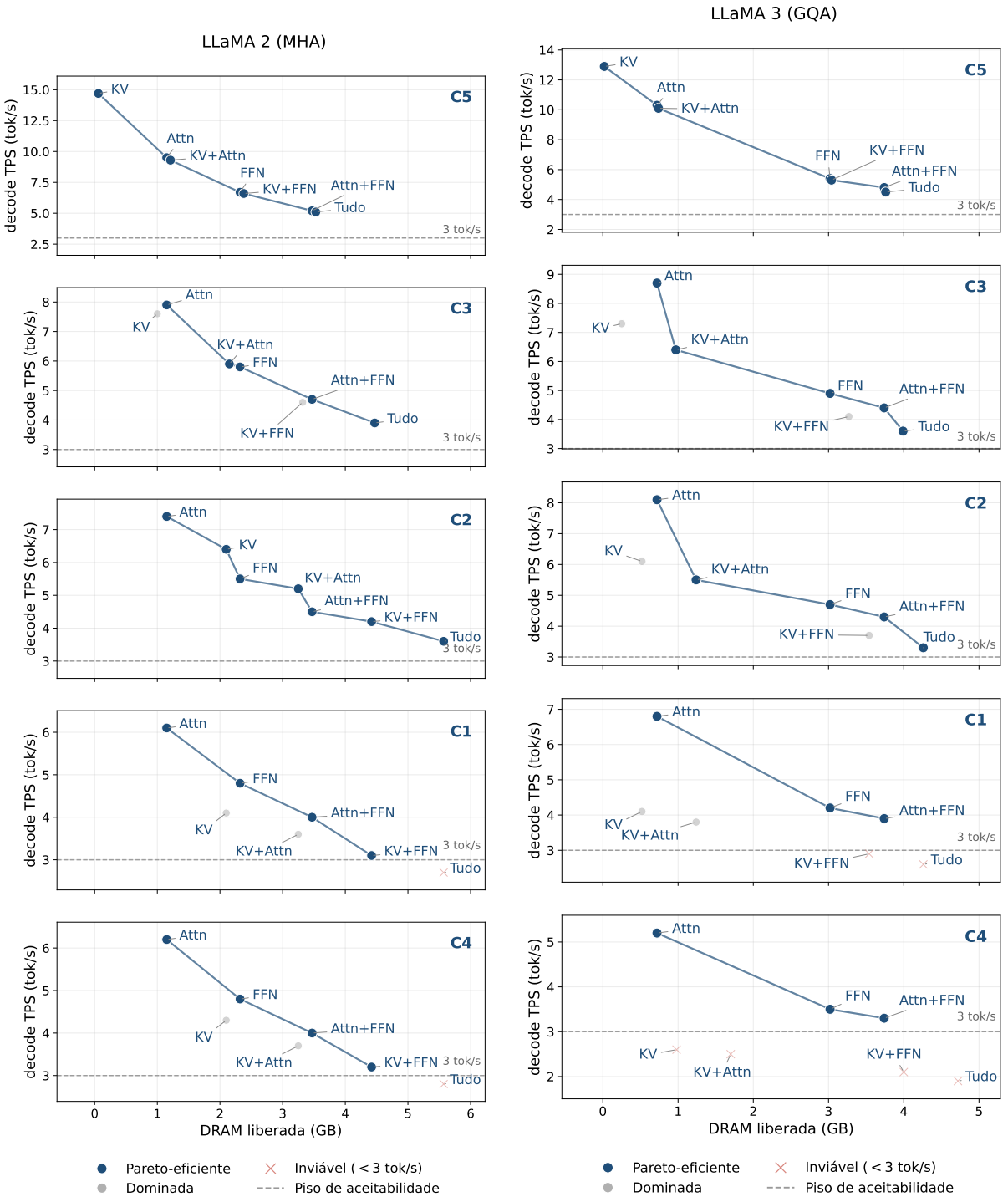


Figura 7 – Fronteira de Pareto por cenário: LLaMA 2 (MHA, esq.) e LLaMA 3 (GQA, dir.). Pontos cheios e ligados pertencem à fronteira; pontos esmaecidos são dominados. A linha tracejada é o piso de aceitabilidade (3,0 tok/s); configurações abaixo dela são inviáveis.

e a própria `KV` isolada em C4. O critério físico de vazão mínima reproduz, sem apelar à magnitude da degradação, a recomendação de poupar a KV-Cache quando o contexto é longo;

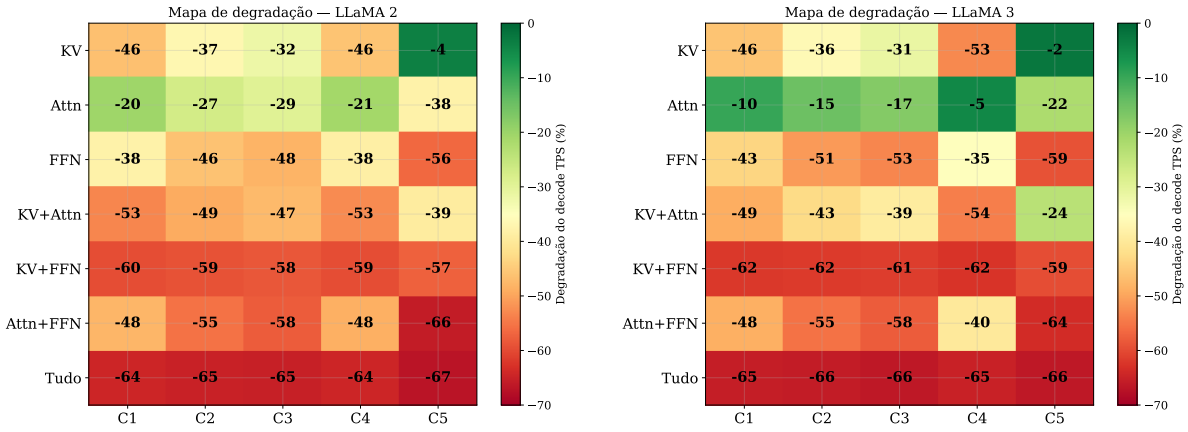
3. **Sob contexto longo, deslocar a KV-Cache é Pareto-dominado em *ambas* as arquiteturas, por razões opostas, mas convergentes.** No LLaMA 3 em C4, apenas três configurações sobrevivem ao corte (`Attn`, `FFN`, `Attn+FFN`), todas preservando a KV-Cache em DRAM: a KV-Cache enxuta do GQA libera pouca DRAM ($\sim$1 GB) e custa vazão suficiente para cair abaixo do piso, de modo que qualquer configuração que a inclua é simultaneamente pouco produtiva e inviável. No LLaMA 2, embora a KV-Cache em C4 seja maior ($\sim$2,1 GB), ela ainda libera *menos* DRAM que a FFN ($\sim$2,3 GB) a um custo de vazão estritamente superior; em consequência, a FFN *domina* a KV-Cache, derrubando `KV` e `KV+Attn` da fronteira (C1 e C4), enquanto `Attn` e `Attn+FFN` permanecem eficientes. Os dois modelos chegam, assim, ao mesmo veredito sob estresse: a fronteira útil contrai-se para configurações que mantêm a KV-Cache em DRAM, seja porque deslocá-la é inviável (GQA), seja porque é dominado (MHA).

A fronteira de Pareto, portanto, não introduz uma recomendação nova, mas corrobora a política sensível ao contexto derivada das QP1–QP4: descarta de saída tudo o que for dominado ou inviável, deixando ao praticante apenas o conjunto de configurações racionais para cada regime. Para consolidar esses padrões em uma única leitura visual e expor, lado a lado, o efeito da arquitetura sobre todo o espaço de decisão, a próxima subseção recorre a mapas de calor da degradação.

5.6.2 Visão integrada: mapas de calor

Estabelecidos o eixo de desempenho (QP1–QP4), o eixo de DRAM (Seção 5.5) e seu cruzamento formal (Seção 5.6.1), a Figura 8 consolida em uma única leitura a degradação relativa do *decode* TPS de todas as configurações (isoladas e combinadas) em todos os cenários, com um mapa por modelo. Diferente das análises anteriores, que isolavam um eixo ou um par de configurações, os mapas expõem simultaneamente todo o espaço de decisão, permitindo que os padrões já demonstrados sejam reconhecidos de relance e, sobretudo, contrastados entre as duas arquiteturas.

Em ambos os mapas, as linhas `KV+FFN` e `Tudo` concentram os tons mais escuros, confirmando visualmente o que a Pareto já formalizara: qualquer combinação que desloque simultaneamente a KV-Cache e a FFN ocupa a faixa de maior degradação, independentemente do cenário ou da arquitetura. No extremo oposto, a combinação mais clara muda com o contexto: `KV+Attn` domina nas colunas de contexto curto (C5, C3, C2), enquanto `Attn+FFN` domina nas de contexto longo (C1, C4).



(a) LLaMA 2 (MHA) (b) LLaMA 3 (GQA)

Figura 8 – Mapas de calor da degradação relativa do *decode* TPS (%) por configuração (linhas) e cenário (colunas). Tons mais escuros indicam maior degradação. A comparação entre os painéis evidencia o efeito do GQA sobre o espaço de decisão de alocação: no LLaMA 3 (b), as configurações que preservam a KV-Cache em DRAM exibem células sistematicamente mais claras do que no LLaMA 2 (a), sobretudo sob contexto longo (C1, C4).

A comparação entre os mapas (a) e (b) revela ainda o efeito do GQA sobre o espaço de decisão:

1. **A linha `Attn` isolada é mais clara no LLaMA 3 do que no LLaMA 2**, refletindo a redução de pesos de projeção do GQA. Esse clareamento propaga-se para as combinações que incluem a Atenção (`KV+Attn` e `Attn+FFN`), que exibem degradação menor do que suas contrapartes no mapa (a);

2. **O gradiente horizontal (C5 → C4) na linha `KV` é mais acentuado no LLaMA 3**: a KV-Cache parte de um tom muito claro em C5 e atinge seu tom mais escuro em C4, evidenciando que, sob GQA, a KV-Cache é simultaneamente muito barata sob contexto curto e o gargalo dominante sob contexto longo. No LLaMA 2, esse gradiente é menos pronunciado porque o custo dos pesos de atenção (maior em MHA) já eleva o patamar de degradação mesmo sob contexto curto.

5.6.3 Diretrizes de estratificação

Os resultados consolidados convergem para um conjunto enxuto de diretrizes prescritivas. Elas não são heurísticas independentes, mas leituras operacionais da fronteira viável: cada diretriz traduz, em linguagem prática, uma propriedade já estabelecida formalmente.

D1 – A política é sensível ao contexto, não fixa. Não há configuração universalmente ótima: preserve em DRAM o componente que é gargalo do regime, a FFN sob contexto curto, a

KV-Cache sob contexto longo. Toda decisão parte da estimativa do contexto operacional esperado;

D2 – Prefira deslocar um único componente. `Attn` e `FFN` isolados são Pareto-eficientes em todos os dez pares (Tab. 16); deslocar um segundo componente reduz vazão sem ganho proporcional de DRAM. Combine apenas quando a meta de memória exceder o que um componente isolado oferece;

D3 – Desloque a KV-Cache sob contexto curto; preserve-a sob contexto longo. Curto: seu deslocamento custa $< 4\%$ de vazão, mas libera pouca DRAM (se a meta é agressiva, mire a FFN). Longo: deslocá-la é Pareto-dominado (LLaMA 2) ou inviável (LLaMA 3);

D4 – Nunca combine KV-Cache com FFN, nem desloque tudo sob contexto longo. `KV+FFN` e `Tudo` concentram as maiores degradações e caem abaixo do piso (< 3 tok/s) em contexto extenso;

D5 – Trate a arquitetura como variável de primeira ordem. Sob GQA, os pesos de atenção tornam-se candidatos quase gratuitos, enquanto a KV-Cache enxuta é ao mesmo tempo pouco produtiva e crítica de preservar. A mesma política gera fronteiras viáveis distintas em MHA e GQA.

5.6.4 Resposta à questão principal

Retomamos, à luz dos resultados consolidados, a questão central da dissertação: *"Como a alocação seletiva de componentes internos de um LLM (pesos de atenção, pesos da FFN e KV-Cache) entre DRAM local e camadas de memória de maior capacidade e maior latência afeta o compromisso entre desempenho e uso de DRAM na inferência em ambientes sem aceleradores?"*

A evidência empírica sustenta uma resposta em três níveis. Primeiro, **a alocação seletiva estabelece um compromisso explícito e quantificável entre DRAM liberada e vazão de *decode***, e esse compromisso *não* é uniforme: a posição de cada componente no plano (DRAM, vazão) depende do regime de contexto, da fase de inferência e da arquitetura. Não existe, portanto, uma alocação universalmente ótima, existe uma fronteira de configurações eficientes que se reconfigura com o regime operacional (Seção 5.6.1).

Segundo, **o compromisso é governado por um único mecanismo subjacente**: o volume de bytes relido por passo de geração. Pesos (Atenção, FFN) impõem um custo fixo, independente do contexto; o estado (KV-Cache) impõe um custo crescente, proporcional ao contexto acessado. Essa assimetria explica a rotação da hierarquia (QP1, QP2), o paradoxo do GQA, em que reduzir a KV-Cache beneficia a estratificação dos *pesos*, não a do estado (QP3), e a dominância da FFN sobre a KV-Cache sob contexto longo (QP4, Pareto).

Terceiro, **a estratificação seletiva é uma estratégia viável e útil em ambientes sem aceleradores, desde que sensível ao contexto**. Sob contexto curto, ela libera memória a custo

desprezível; sob contexto longo, ainda oferece configurações eficientes (tipicamente preservando a KV-Cache em DRAM), embora a fronteira viável se contraia e parte do espaço de configurações torne-se inaceitavelmente lenta. A contribuição central deste trabalho é, assim, menos a recomendação de uma configuração específica e mais a demonstração de que a alocação seletiva deve ser tratada como um problema de otimização bi-objetivo sensível ao regime, para o qual a fronteira de Pareto fornece o critério de decisão e as diretrizes D1–D5 fornecem um guia prático.

6 Conclusão

Esta dissertação investigou a alocação seletiva de componentes internos de LLMs (pesos de atenção, pesos da FFN e KV-Cache) entre DRAM e uma camada de memória de maior capacidade e maior latência em ambientes de CPU, visando equilibrar desempenho de inferência e uso de DRAM. A análise empírica revelou que o gargalo da estratificação é governado por um único mecanismo: o volume de bytes relidos por passo de geração, que determina qual componente se torna o principal limitador em função do regime de operação. Sob contextos curtos, a maior sensibilidade recai sobre os pesos da FFN; sob contextos longos, a KV-Cache passa a dominar.

Os resultados demonstraram que não existe um componente universalmente mais tolerante à estratificação (QP1): a hierarquia de custos rotaciona ao longo do gradiente de contexto. A sensibilidade da alocação é governada pela leitura média de estado por passo de geração, e não pelo componente ou pelo tamanho final do contexto isoladamente (QP2). Diferenças arquiteturais são de primeira ordem (QP3): o GQA não torna o modelo uniformemente mais tolerante, mas *redistribui* a tolerância, barateando a estratificação dos pesos de atenção e, relativamente, encarecendo a da KV-Cache. Por fim, nenhuma combinação de componentes é universalmente dominante (QP4): a política de menor custo é condicionada ao contexto, preservando em DRAM o componente que constitui o gargalo de cada regime.

A formalização do problema como uma fronteira de Pareto com restrição de vazão levou à derivação de diretrizes práticas que orientam escolhas de alocação compatíveis com o regime de operação e a arquitetura do modelo (MHA *vs.* GQA).

6.1 Limitações

Como todo estudo empírico, esta pesquisa foi conduzida sob um conjunto de delimitações deliberadas, adotadas para assegurar o rigor e a reprodutibilidade dos resultados. Longe de invalidar as conclusões, tais recortes definem com precisão o domínio em que as diretrizes propostas são diretamente aplicáveis e apontam extensões naturais para trabalhos posteriores.

No plano do hardware, a validação experimental concentrou-se no Intel Optane DC, tecnologia que, embora descontinuada comercialmente, permanece um representante paradigmático de memória persistente endereçável por byte e oferece um banco de provas maduro e bem caracterizado para o estudo de estratificação. A generalização para dispositivos baseados em CXL apoia-se na equivalência qualitativa do modelo de custo (acesso byte a byte a uma camada de maior capacidade e maior latência) mas ainda não foi verificada empiricamente. Como o mecanismo governante identificado (o volume de bytes relidos por passo) independe da

tecnologia física subjacente, há indícios fortes de que as diretrizes se transfiram, o que constitui uma hipótese a ser confirmada.

No plano do modelo e da carga de trabalho, o escopo concentrou-se em modelos de 7–8 bilhões de parâmetros, na geração autorregressiva token a token, com $batch = 1$, KV-Cache em FP16 e pesos quantizados em Q4_K_M. Essa configuração corresponde a um cenário de implantação realista e de amplo interesse prático (inferência de baixo custo em infraestrutura sem aceleradores), mas restringe a extrapolação imediata para modelos de maior escala, regimes de *batch* elevado, arquiteturas de atenção distintas ou outras estratégias de compressão de estado. Ainda assim, por decorrerem de um princípio único e mensurável, as conclusões oferecem uma base analítica que transcende os pontos específicos avaliados.

6.2 Trabalhos Futuros

As delimitações acima abrem um conjunto de direções de pesquisa promissoras, que estendem tanto a validação quanto o alcance analítico deste trabalho.

- Validação em hardware emergente: A verificação direta das diretrizes em plataformas com memória CXL real permitiria confirmar a hipótese de equivalência de custo e quantificar eventuais desvios introduzidos pela latência e pela banda específicas dessa interconexão. Uma caracterização comparativa entre Optane e CXL consolidaria o modelo de custo como agnóstico à tecnologia física;

- Ampliação do espaço de modelos e arquiteturas: Estender a caracterização a modelos maiores (dezenas a centenas de bilhões de parâmetros) e a variantes de atenção adicionais, como *Multi-Query Attention* e mecanismos de atenção esparsa ou linear, verificaria se o *paradoxo de redistribuição* observado entre MHA e GQA se generaliza, e se o ponto de rotação da hierarquia de custos se desloca de forma previsível com a razão entre pesos e estado;

- Alocação dinâmica ciente do regime: Enquanto este trabalho assumiu uma alocação estática, a dependência do gargalo em relação ao regime de operação sugere ganhos com políticas *dinâmicas*, capazes de migrar componentes entre camadas ao longo da inferência, por exemplo, promovendo a KV-Cache à DRAM à medida que o contexto cresce e o custo de releitura se torna dominante. O desafio central está em manter o custo de migração inferior ao benefício obtido.

Referências

AINSLIE, J. et al. GQA: Training generalized multi-query transformer models from multi-head checkpoints. In: *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Singapore: Association for Computational Linguistics, 2023. p. 4895–4901. Citado 3 vezes nas páginas 25, 27 e 36.

ALIZADEH, K. et al. LLM in a flash: Efficient large language model inference with limited memory. In: *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL), Volume 1: Long Papers*. [S.l.: s.n.], 2024. p. 12562–12584. ArXiv:2312.11514. Citado 3 vezes nas páginas 38, 39 e 41.

BALDASSIN, A. et al. Persistent memory: A survey of programming support and implementations. *ACM Comput. Surv.*, v. 54, n. 7, jul. 2021. ISSN 0360-0300. Citado na página 30.

BERGER, D. S. et al. Cxl in cloud practice: Practical lessons for incrementally scaling deployment. *IEEE Transactions on Computers*, v. 75, n. 4, p. 1234–1246, 2026. Citado na página 22.

CANTALUPO, C. et al. *memkind: An Extensible Heap Memory Manager for Heterogeneous Memory Platforms and Mixed Memory Policies*. [S.l.], 2015. OSTI 1245908. Citado na página 34.

CORBET, J. Autonuma: the other approach to numa scheduling. *LWN. net*, 2012. Citado na página 33.

DETTMERS, T. et al. LLM.int8(): 8-bit matrix multiplication for transformers at scale. *Advances in Neural Information Processing Systems*, v. 35, 2022. Citado na página 20.

DU, H. et al. FlexInfer: Breaking memory constraint via flexible and efficient offloading for on-device LLM inference. In: *Proceedings of the 5th Workshop on Machine Learning and Systems (EuroMLSys)*. [S.l.]: ACM, 2025. p. 56–65. ArXiv:2503.03777. Citado 3 vezes nas páginas 38, 39 e 41.

FANG, Y. et al. Accelerating LLM inference via dynamic KV cache placement in heterogeneous memory system. *arXiv preprint arXiv:2508.13231*, 2025. Citado 3 vezes nas páginas 39, 40 e 41.

FRANTAR, E. et al. OPTQ: Accurate quantization for generative pre-trained transformers. In: *Proceedings of the 11th International Conference on Learning Representations (ICLR)*. [S.l.: s.n.], 2023. Originally titled GPTQ on arXiv (arXiv:2210.17323). Renamed to OPTQ at camera-ready. Citado 2 vezes nas páginas 28 e 35.

GALVIN, P. B. et al. Operating system concepts. In: ____. 10th. ed. [S.l.]: John Wiley & Sons, 2018. p. 436–440, 795–803. Citado na página 33.

GANJIHAL, S. R. Predictive multi-tier memory management for KV cache in large-scale GPU inference. *arXiv preprint arXiv:2604.26968*, 2025. Citado 2 vezes nas páginas 36 e 37.

GERGANOV, G. et al. *llama.cpp*. 2023. <https://github.com/ggerganov/llama.cpp>. Accessed: 2025. Citado 4 vezes nas páginas 28, 35, 38 e 41.

GEVA, M. et al. Transformer feed-forward layers are key-value memories. In: *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Online and Punta Cana, Dominican Republic: Association for Computational Linguistics, 2021. p. 5484–5495. Citado na página 25.

GHOSE, S. et al. Processing-in-memory: A workload-driven perspective. *IBM Journal of Research and Development*, v. 63, n. 6, p. 3:1–3:19, 2019. Citado na página 32.

GRATTAFIORI, A. et al. The Llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024. Citado 4 vezes nas páginas 19, 21, 44 e 45.

GU, Y. et al. PIM is all you need: A CXL-enabled GPU-free system for large language model inference. In: *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. [S.l.: s.n.], 2025. ArXiv:2502.07578. Citado 2 vezes nas páginas 38 e 41.

HENNESSY, J. L.; PATTERSON, D. A. *Computer Architecture, Sixth Edition: A Quantitative Approach*. 6th. ed. [S.l.]: Morgan Kaufmann Publishers Inc., 2017. ISBN 0128119055. Citado na página 29.

HILDEBRAND, M. et al. A case against hardware managed dram caches for nvram based systems. In: *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. [S.l.: s.n.], 2021. p. 194–204. Citado na página 30.

JIANG, X. et al. NEO: Saving GPU memory crisis with CPU offloading for online LLM inference. In: *Proceedings of Machine Learning and Systems (MLSys)*. [S.l.: s.n.], 2025. Citado na página 20.

LASCH, R. et al. Cost modelling for optimal data placement in heterogeneous main memory. *Proceedings of the VLDB Endowment*, v. 15, n. 11, p. 2867–2880, 2022. Citado na página 21.

LEE, D. et al. Simultaneous Multi-Layer Access: Improving 3D-Stacked Memory Bandwidth at Low Cost. *ACM Transactions on Architecture and Code Optimization*, v. 12, n. 4, p. 49:1–49:29, 2016. Citado na página 32.

LEE, K. et al. Improving key-value cache performance with heterogeneous memory tiering: A case study of compute-express-link-based memory expansion. *IEEE Micro*, v. 45, n. 2, p. 102–113, 2025. Citado na página 33.

LEE, S.; KIM, S.-h. et al. A 1ynm 1.25V 8Gb, 16Gb/s/pin GDDR6-based Accelerator-in-Memory supporting 1TFLOPS MAC Operation and Various Activation Functions. In: *IEEE International Solid-State Circuits Conference (ISSCC)*. [S.l.: s.n.], 2022. p. 1–3. Citado na página 32.

LEE, T. et al. Memtis: Efficient memory tiering with dynamic page classification and page size determination. In: *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP '23)*. [S.l.]: ACM, 2023. Citado 2 vezes nas páginas 20 e 32.

LEVIATHAN, Y.; KALMAN, M.; MATIAS, Y. Fast inference from transformers via speculative decoding. In: *Proceedings of the 40th International Conference on Machine Learning (ICML)*. [S.l.: s.n.], 2023. (Proceedings of Machine Learning Research, v. 202), p. 19274–19286. Citado na página 19.

LEWIS, P. et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. In: *Proceedings of the 34th International Conference on Neural Information Processing Systems*. [S.l.: s.n.], 2020. (NIPS '20). ISBN 9781713829546. Citado na página 20.

LI, H. et al. Pond: Cxl-based memory pooling systems for cloud platforms. In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. [S.l.: s.n.], 2023. (ASPLOS 2023), p. 574–587. ISBN 9781450399166. Citado na página 22.

LIN, J. et al. KVDrive: A holistic multi-tier KV cache management system for long-context LLM inference. *arXiv preprint arXiv:2605.18071*, 2025. Citado 3 vezes nas páginas 36, 37 e 41.

LIN, J. et al. AWQ: Activation-aware weight quantization for on-device LLM compression and acceleration. In: *Proceedings of the 7th Conference on Machine Learning and Systems (MLSys)*. [S.l.: s.n.], 2024. Best Paper Award. arXiv:2306.00978. Citado 2 vezes nas páginas 28 e 35.

LIN, M. et al. HybridGen: Efficient LLM generative inference via CPU–GPU hybrid computing. *arXiv preprint arXiv:2604.18529*, 2025. Citado 2 vezes nas páginas 36 e 37.

LIN, S.; GUO, Z.; LIN, J. DAK: Direct-access-enabled GPU memory offloading with optimal efficiency for LLM inference. *arXiv preprint arXiv:2604.26074*, 2025. Citado 2 vezes nas páginas 36 e 37.

LIU, L. et al. PAM: Processing across memory hierarchy for efficient KV-centric LLM serving system. *arXiv preprint arXiv:2602.11521*, 2025. Citado 2 vezes nas páginas 39 e 40.

LUO, C. et al. HeadInfer: Memory-efficient LLM inference by head-wise offloading. *arXiv preprint arXiv:2502.12574*, 2025. Citado 2 vezes nas páginas 19 e 28.

MARUF, H. A. et al. Tpp: Transparent page placement for cxl-enabled tiered-memory. In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. [S.l.: s.n.], 2023. p. 742–755. ISBN 9781450399180. Citado na página 33.

MIAO, X. et al. Towards efficient generative large language model serving: A survey from algorithms to systems. *arXiv preprint arXiv:2312.15234*, 2024. Citado na página 19.

MUTLU, O. et al. A modern primer on processing in memory. *Emerging Computing: From Devices to Systems*, Springer, 2021. Also available as arXiv:2012.03112. Citado na página 32.

NA, S. et al. FlexInfer: Flexible LLM inference with CPU computations. In: *Proceedings of the 8th Conference on Machine Learning and Systems (MLSys)*. [S.l.: s.n.], 2025. Citado 5 vezes nas páginas 20, 22, 36, 37 e 41.

RADFORD, A. et al. *Language Models are Unsupervised Multitask Learners*. [S.l.], 2019. Citado na página 19.

REN, Z. et al. An i/o characterizing study of offloading llm models and kv caches to nvme ssd. In: *Proceedings of the 5th Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems*. [S.l.: s.n.], 2025. (CHEOPS '25), p. 23–33. ISBN 9798400715297. Citado na página 21.

SCARGALL, S. *Programming Persistent Memory - A Comprehensive Guide for Developers*. 1st. ed. [S.l.]: Apress, 2020. ISBN 978-1-4842-4932-1. Citado 3 vezes nas páginas 11, 30 e 34.

SHARMA, D. D.; BLANKENSHIP, R.; BERGER, D. An introduction to the compute express link (cxl) interconnect. *ACM Comput. Surv.*, v. 56, n. 11, jul. 2024. ISSN 0360-0300. Disponível em: <https://doi.org/10.1145/3669900>. Citado 2 vezes nas páginas 20 e 31.

SHAZEER, N. Fast transformer decoding: One write-head is all you need. *arXiv preprint arXiv:1911.02150*, 2019. Citado na página 36.

SHAZEER, N. GLU variants improve transformer. *arXiv preprint arXiv:2002.05202*, 2020. Citado 2 vezes nas páginas 27 e 29.

SHENG, Y. et al. FlexGen: High-throughput generative inference of large language models with a single GPU. In: *Proceedings of the 40th International Conference on Machine Learning (ICML)*. [S.l.: s.n.], 2023. (PMLR, v. 202), p. 31094–31116. Citado 2 vezes nas páginas 36 e 41.

SONG, K. et al. HybridTier: An adaptive and lightweight CXL-memory tiering system. In: *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '25)*. [S.l.]: ACM, 2025. Citado na página 20.

STOJKOVIC, J. et al. DynamoLLM: Designing LLM inference clusters for performance and energy efficiency. In: *Proceedings of the 2025 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. [S.l.: s.n.], 2025. Citado na página 22.

SU, J. et al. RoFormer: Enhanced transformer with rotary position embedding. *Neurocomputing*, v. 568, p. 127063, 2024. Citado na página 29.

TANG, Y. et al. Exploring CXL-based KV cache storage for LLM serving. In: *NeurIPS 2024 Workshop on Machine Learning for Systems*. [S.l.: s.n.], 2024. Citado 2 vezes nas páginas 39 e 40.

TOUVRON, H. et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023. Citado 3 vezes nas páginas 19, 21 e 29.

TOUVRON, H. et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023. Citado 2 vezes nas páginas 21 e 44.

VASWANI, A. et al. Attention is all you need. In: *Advances in Neural Information Processing Systems*. [S.l.: s.n.], 2017. v. 30. Citado 5 vezes nas páginas 11, 19, 25, 26 e 27.

WANG, X. et al. Exploring and evaluating real-world CXL: Use cases and system adoption. In: *Proceedings of the International Symposium on Memory Systems (MemSys)*. [S.l.: s.n.], 2024. Citado na página 31.

WU, J. et al. Performance study of CXL memory topology. In: *Proceedings of the International Symposium on Memory Systems (MemSys)*. [S.l.: s.n.], 2024. Citado na página 31.

WULF, W. A.; MCKEE, S. A. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH Computer Architecture News*, v. 23, n. 1, p. 20–24, 1995. Citado na página 29.

XIAO, G. et al. Efficient streaming language models with attention sinks. In: *Proceedings of the 12th International Conference on Learning Representations (ICLR)*. [S.l.: s.n.], 2024. ArXiv:2309.17453. Citado na página 36.

YANG, J. et al. An empirical guide to the behavior and use of scalable persistent memory. In: *18th USENIX Conference on File and Storage Technologies (FAST '20)*. Santa Clara, CA: USENIX Association, 2020. p. 169–182. ISBN 978-1-939133-12-0. Citado 4 vezes nas páginas 13, 20, 30 e 31.

YANG, X. et al. Beluga: A CXL-based memory architecture for scalable and efficient LLM KVCache management. *arXiv preprint arXiv:2511.20172*, 2025. Citado 3 vezes nas páginas 39, 40 e 41.

YUAN, Z. et al. LLM inference unveiled: Survey and roofline model insights. *arXiv preprint arXiv:2402.16363*, 2024. Citado na página 19.

ZHANG, B.; SENNRICH, R. Root mean square layer normalization. In: *Advances in Neural Information Processing Systems (NeurIPS)*. [S.l.: s.n.], 2019. v. 32. Citado na página 29.

ZHANG, Z. et al. H_2O: Heavy-hitter oracle for efficient generative inference of large language models. In: *Advances in Neural Information Processing Systems 36 (NeurIPS)*. [S.l.: s.n.], 2023. Citado na página 36.

ZHENG, X. et al. Adaptive multi-objective tiered storage configuration for KV cache in LLM service. *arXiv preprint arXiv:2603.08739*, 2025. Citado 2 vezes nas páginas 36 e 37.

ZHONG, Y. et al. DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving. In: *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. [S.l.: s.n.], 2024. p. 193–210. Citado na página 66.

DADOS CURRICULARES

IDENTIFICAÇÃO
Nome: Luísa Leona Cattai
Nacionalidade: Brasileira
Nome em citações bibliográficas: CATTAI, L. L.
Currículo Lattes: <http://lattes.cnpq.br/4872969672275994>
ORCID: <https://orcid.org/0009-0009-3270-9448>

FORMAÇÃO ACADÊMICA
2024–2026: Mestrado em Ciência da Computação
Universidade Estadual Paulista "Júlio de Mesquita Filho" (UNESP), Câmpus de Rio Claro.

2019–2023: Bacharelado em Ciências da Computação (ou seu curso)
Universidade Estadual Paulista "Júlio de Mesquita Filho" (UNESP), Câmpus de Rio Claro.

PRODUÇÃO BIBLIOGRÁFICA
CATTAI, Luísa; BALDASSIN, Alexandro; DANTAS, Allberson. Otimização de Inferência em LLMs na CPU: Análise do Cenário Atual. In: ESCOLA REGIONAL DE ALTO DESEMPENHO DE SÃO PAULO (ERAD-SP), 16. , 2025, São José do Rio Preto/SP. Anais [...]. Porto Alegre: Sociedade Brasileira de Computação, 2025 . p. 78-81. DOI: https://doi.org/10.5753/eradsp.2025.9731.

CATTAI, Luísa; BALDASSIN, Alexandro. Investigando a Efetividade de LLMs na Otimização de Código Paralelo. In: ESCOLA REGIONAL DE ALTO DESEMPENHO DE SÃO PAULO (ERAD-SP), 15. , 2024, Rio Claro/SP. Anais [...]. Porto Alegre: Sociedade Brasileira de Computação, 2024 . p. 1-4. DOI: https://doi.org/10.5753/eradsp.2024.239595.

PARTICIPAÇÃO EM EVENTOS CIENTÍFICOS
XV ESCOLA REGIONAL DE ALTO DESEMPENHO DE SÃO PAULO, 2024, Rio Claro. Investigando a Efetividade de LLMs na Otimização de Código Paralelo. 2024. (Congresso).