Analysis of Memory Access Patterns for Large Language Model Inference

Max H. Fisher

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science and Applications

Dimitrios S. Nikolopoulos, Chair

Godmar V. Back

Huaicheng Li

April 30th, 2025

Blacksburg, Virginia

Keywords: NUMA, Page Placement, Memory Access Patterns, High Performance Computing, LLM Inference

Analysis of Memory Access Patterns for Large Language Model Inference

Max H. Fisher

(ABSTRACT)

The use of tiered heterogeneous memory systems in HPC workloads is growing in popularity as the increasing memory requirements for these workloads outpace the decline in the cost-per-gigabyte of fast DRAM; however, the Linux kernel has no intelligent strategy to manage these tiered memory systems. Because of this limitation, a great deal of research has been conducted to identify policies that make efficient use of these systems. Much of this prior research focuses on deep learning tasks, while only a few focus on inference for large models. The training and inference workloads for the same type of model are quite different: in training, the task is to continuously update the weights matrices with knowledge gained from each training datum, while in inference, the workload only reads from the weights. Training for neural networks also involves accesses in reverse order to what is used in inference, in a training technique called backpropagation. This thesis presents a memory access pattern heatmap tool that can track evolving access patterns through the lifetime of a workload. This tool is applied to llama.cpp, an LLM inference tool, to identify memory access patterns between remote and local NUMA nodes. The thesis then explores two basic NUMA page placement strategies, where all memory is bound to either the local or remote NUMA nodes to identify the impact of poor NUMA policies on performance and compares them to the default Linux strategy.

Analysis of Memory Access Patterns for Large Language Model Inference

Max H. Fisher

(GENERAL AUDIENCE ABSTRACT)

Scientific computing often involves running programs with very large memory footprints that might not fit in the available memory for the system on which they run. Because expanding the available memory on a system can be expensive, tiered memory systems, which provide the illusion of a fast and large memory by storing some data in regular small, expensive, and fast memory (DRAM) and the rest of the data in large, cheap, and slow memory (NVM), have been growing in popularity. However, effective use of such systems requires an intelligent strategy for moving data between tiers of memory. Past research has explored strategies that leverage data access patterns in programs like those that train machine learning models, but few explore leveraging the access patterns of running machine learning models. This thesis explores how llama.cpp, a program that runs large language models, makes accesses to data and how those patterns change as the program runs. It also explores how differing data placement policies impact performance. It compares how quickly llama.cpp runs when all data is allocated to a slow memory node to when all data is allocated to a fast memory node, reflecting how important these strategies are to fast performance.

Dedication

To Pearl, Liberty, and Laura

Contents

List of Figures viii

List of Tables x

1 Introduction 1

1.1 Background 1

1.2 Related Work 2

1.3 Motivation 3

1.4 Contributions 4

2 Review of Literature 7

2.1 Kernel-Space Systems 7

2.1.1 Transparent Page Placement 7

2.1.2 vTMM 8

2.1.3 Radiant 9

2.1.4 Application-Attuned Memory Management for Containerized Workloads 10

2.2 User-Space Systems with Online Analysis 11

2.2.1 CachedAttention 11

2.2.2 HeMem 12

2.2.3 DyNN Offload 13

2.3 User-Space Systems with Offline Analysis 14

2.3.1 AutoTM 14

2.3.2 CXL Strategies for GPU-based workloads 15

2.3.3 DeepMemoryDL 15

2.3.4 NoPFS 16

3 Methodology **17**

3.1 Environment 17

3.2 Heatmap Tool Design 17

3.3 numactl Testing 25

4 Results **27**

4.1 Heatmaps 27

4.2 numactl 43

4.2.1 STREAM Benchmarking Results 43

4.2.2 Impact of NUMA Strategies on LLM Inference 43

5 Discussion **50**

5.1 Proposed Strategy 51

6 Conclusions **53**

6.1 Future Work 53

Bibliography **55**

List of Figures

3.1 Diagram of the three layers in the heatmap tool. The implemented modules are highlighted in green, while potential future modules are also listed. . . . 18

4.1 Memory access pattern for llama.cpp during the inference phase and not including the initialization phase. Each granule is 16MB in size. 28

4.2 Memory access heatmap for an example matrix-vector multiplication 29

4.3 Memory access pattern for the larger working region of llama.cpp over its lifetime with 4MB granules. 31

4.4 Figure 4.3 heatmap with 540 columns. 32

4.5 Scatterplot of access counts to a snippet of the working region of llama.cpp. 32

4.6 Autocorrelation graph for the heatmap displayed in figure 4.3 33

4.7 From left to right: Memory access heat maps of the working region of llama.cpp for consecutive 10 second intervals during the inference phase. 34

4.8 Memory access pattern for llama.cpp simulating a CXL system and ignoring accesses to the cache. Each gramule is 4MB. 36

4.9 Autocorrelation function for the heatmap shown in 4.8. 37

4.10 Remote accesses made by llama.cpp for the same trial as Figure 4.8. 38

4.11 From left to right: Memory access heat maps of the working region of llama.cpp simulating a CXL system for consecutive 10-second intervals during the inference phase. 39

4.12 From left to right: Remote memory access heat maps of the working region of llama.cpp simulating a CXL system for the same consecutive 10-second intervals as shown in Figure 4.11. 41

4.13 From left to right: Remote memory access heat maps of the working region of llama.cpp simulating a CXL system for consecutive 1 second intervals. . . 42

List of Tables

3.1 User-Configurable Parameters for the Heatmap Tool 23

4.1 Results of STREAM benchmark of memory bandwidth for each NUMA strategy, reported in MB/s. Higher is better. 43

4.2 Average and standard deviation of real and system times in seconds over 10 warm-start trials of llama.cpp for each strategy. Lower is better. 44

4.3 perf mem report for llama.cpp running on the different node strategy with a warm start, sorted by the type of memory access. The overhead in this case represents the percent of samples in this category 45

4.4 perf mem report for llama.cpp running on the different node strategy with a cold start, sorted by the type of memory access. The overhead again represents the percent of samples in this category. 46

4.5 Average and standard deviation of real and system times in seconds over 10 cold-start trials of llama.cpp for each strategy. Lower is better. 46

4.6 perf mem report for llama.cpp running on the default memory strategy with a cold start, sorted by the type of memory access. The overhead represents the percent of samples in this category 48

4.7 perf mem report for llama.cpp running on the same node memory strategy with a cold start, sorted by the type of memory access. The overhead represents to percent of samples in this category 48

List of Abbreviations

CPU Central Processing Unit

CXL Compute Express Link

DL Deep Learning

DRAM Dynamic Random Access Memory

GPT Generative Pre-trained Transformer

GPU Graphical Processing Unit

LLM Large Language Model

NVM Non Volatile Memory

OS Operating System

PEBS Processor Event Based Sampling

PML Page Modification and Logging

TPP Transparent Page Placement

Chapter 1

Introduction

1.1 Background

The use of heterogeneous memory systems in high performance and scientific computing is growing as new systems and products come to market [29, 30]. These systems combine high-speed, low-capacity, expensive Dynamic Random Access Memory (DRAM) with low-speed, high-capacity Non Volatile Memory (NVM) to build a memory hierarchy that provides the illusion of a fast and large memory. Popular examples of these systems include Compute Express Link (CXL) [48] and Intel Optane DC [31]. These systems are becoming more important as the need for large-memory high performance computers for Deep Learning (DL) tasks grows. DL tasks have had increasingly large memory footprints that are outpacing the expansion in storage capacity for DRAM [36]. However, standards like CXL are still in their infancy, and only recently have CXL compliant modules come onto market. As such, there is limited support for heterogeneous memory hierarchies in the Linux Kernel [42].

In the Linux kernel, each tier of memory is represented by separate Non-Uniform Memory Access (NUMA) nodes, which are generally associated with a Central Processing Unit (CPU). On systems with multiple CPUs, each NUMA node represents the memory physically closer to the CPU, which allows the kernel to automatically allocate memory close to the requesting CPU. There is no official support for tiered memory systems other than allowing swapping between NUMA nodes [40]. By default, Linux will allocate pages to the NUMA

node closest to the requesting CPU. When that node is full, it will move on to further nodes and will not allocate on the closer node again until space frees up [36]. This naive strategy for the placement of pages in a tiered memory system has a substantial performance cost compared to a homogeneous memory system with a large and fast DRAM [29] [16]. In fact, a heterogeneous tiered memory system using a naive page placement strategy may perform similarly to a homogeneous memory system with only a large, slow tier, negating the benefit of a heterogeneous hierarchy [29]. Sub-optimal memory strategies not only reduce the performance of workloads, they also increase energy use. With Large Language Models (LLMs) being used in more latency critical workloads, the need for an optimal tiered memory management strategy is clear. To address this issue, the research community has proposed many strategies for managing tiered memory systems.

1.2 Related Work

Some strategies such as Transparent Page Placement (TPP) [36] use live analysis of memory use to decide when and how to migrate pages between remote and local memory tiers. However, TPP does not do any analysis of the workload or attempt to pre-fetch pages from remote memory before they are needed. Other strategies such as DeepMemoryDL [15] do use workload specific knowledge to develop pre-fetching strategies offline, but these strategies are specific to training machine learning models, not for inference. There is a dearth of strategies in the literature discussing pre-fetching strategies specific to LLM inference. Some strategies such as InstInfer [39] and Glinthawk [27] explore offloading some parts of the inference engine to a computational storage device; however, this is not a pre-fetching strategy as it offloads computation rather than making data available at local nodes when it is needed and keeping computation on-board. It also requires alteration to the workload code, which may not

always be possible. More recently, strategies that offload parts of the model to regular Solid State Drives (SSDs) have been explored [35]. CachedAttention [25], for example, enables the re-use of Key-Value caches used in a system serving multiple LLM clients at once. It does so by offloading the computed KV cache for each session to an SSD and then restoring that cache when the session's client returns to reduce computations. However, this is not generalized pre-fetching of memory pages over the course of an inference; it merely reduces the computational workload for an inference engine when swapping between sessions. None of these strategies explore an offline analysis to design a pre-fetching strategy for all pages in working memory during an inference. The strategies are explored in detail in Chapter 2 and vary by whether they run in kernel or user space, whether they analyze the workload or memory access pattern before execution or live, and the type of tool or workload that they support.

1.3 Motivation

While some strategies are general purpose, many interesting approaches leverage knowledge about specific workloads to find even more performance. Most such strategies focus on DL training; however, inference still has a large memory footprint, and few prior studies in the literature propose strategies specific to Machine Learning (ML) inference. The strategies that do exist operate at the software and model layer by altering the workload to make better use of limited DRAM. A system-level memory placement and pre-fetching strategy would complement them well by finding additional opportunities for performance improvement regardless of internal attention optimizations. ML inference is a task that is repeated many times over the lifetime of a model, so even small speedups can have substantial impacts on cloud infrastructure energy use and costs over time. Further, LLMs are being implemented

in time sensitive use cases such as fraud detection [47], increasing the need to performance improvements in LLM inference.

1.4 Contributions

To resolve this shortcoming in the literature, this thesis presents a memory heatmap tool that allows users to visualize remote and local memory accesses separately. Users can configure the heatmap to display either local accesses, remote accesses, or a combination of both to understand spatial access patterns–how regions of memory are accessed in relation to their neighboring regions. The tool can display heatmaps for individual periods during the workload execution by splitting execution into consecutive, equal-duration time slices and displaying a memory access heatmap representing all accesses made during an individual time slice, allowing users to track access patterns over time. The length of the time slices is user-configurable, and the tool can also be configured to display different types of accesses such as cache accesses, local RAM accesses, or remote RAM accesses. It can also display the heatmap image with varying dimensions based on the user's need; by default it generates a square image. These features allow the heatmap tool to be used to inform a potential workload-aware pre-fetching strategy. This idea is explored further in Chapter 5.1.

Second, this thesis conducts an analysis of the memory access patterns for LLM inference and proposes a strategy to exploit those patterns. This thesis first explores the costs of bad NUMA policies by testing various NUMA placement strategies, including the worst case strategy that forces all allocations to be to the remote NUMA node. In theory, this would result in markedly worse performance than other strategies, such as allowing the kernel to decide automatically where to place pages. In practice, however, this performance penalty was only observed when the inference workload was run for the first time after the system

restarted (i.e. a cold start). When run subsequently (i.e. a warm start), the performance penalty was minor. Both of these strategies performed worse than the ideal case, forcing all page allocations possible to occur on the local NUMA node. Counter-intuitively, allowing the kernel to decide where to place pages on its own performed worse than forcing all allocations onto the local node. This is explored further in Chapter 4.2.

Third, this thesis leverages the aforementioned heatmap tool to examine memory access patterns in a real world LLM CPU inference workload. Although most real-world inference workloads happen on GPUs, GPU systems can still make use of tiered memory systems, both in the form of CXL and also GPU-to-GPU remote memory accesses [18]. Even though the type of analysis presented here for CPU inference is not yet possible for GPU inference, this work can still be relevant to multi-GPU systems. Many models are too large to fit in a single GPU's on-board memory, so past research has explored tiered memory management in multi-GPU systems [18].

The analysis in this thesis finds a clear pattern both in spatial terms and temporally. There is a clear stride in memory accesses, which is shown by memory regions experiencing higher access counts occurring at relatively constant strides from each other. There is also a hotspot of accesses covering part of the memory region that clearly moves across the entire working set as time goes on. These patterns indicate there is room for a pre-fetching strategy to identify pages placed remotely before they are accessed and make them available at local nodes, reducing the time delay for the access. This is explored in detail in Chapter 4.1.

The contributions of this thesis are as follows: first, it presents a user-configurable heatmap tool to display memory access patterns both over the entire course of a process's execution and for different time periods within the process's execution. Second, it conducts an empirical study into the impacts of NUMA page placement policies, including the case where all pages are allocated remotely. Finally, it analyzes the memory access patterns of a real world LLM

inference workload and proposes an offline pre-fetching strategy that exploits those patterns.

Chapter 2

Review of Literature

Several strategies have been proposed in the literature to manage heterogeneous tiered memory systems. These can be grouped based on their status as a user space or kernel space system and by whether they conduct their analysis at runtime or offline prior to the workload starting.

2.1 Kernel-Space Systems

2.1.1 Transparent Page Placement

TPP [36] is an Operating System (OS) level strategy that dynamically identifies hot and cold pages using a combination of AutoNUMA [14] and Linux's least-recently-used page management system to migrate pages proactively, ensuring there is enough headroom at local NUMA nodes for new allocations. TPP will identify cold pages present at local memory nodes and offload them asynchronously once a high watermark is reached on the local node. This strategy limits the chance of a hot page being demoted in most circumstances. It uses AutoNUMA on CXL nodes to identify and migrate hot pages trapped in remote memory via a similar process. TPP is application agnostic and transparent to the programmer. It does not need to know the nature of the workload, nor does it require any changes to applications to run. It can be deployed globally with a kernel release. Although TPP does

not make guesses about where to place new pages as they are allocated, it does allow some applications that allocate caches to prioritize the CXL node for that cache while retaining the standard policy for other pages. TPP was evaluated on several workloads designed to mimic common workloads at Meta. On a workload designed to mimic the behavior of a Meta web server, 90% of memory accesses were served from the local node compared to just 22% for the default Linux strategy. TPP is among the state of the art for tiered memory management, although it is designed for general purpose use, and there may be more room for improvement if knowledge about the workload is leveraged.

2.1.2 vTMM

vTMM [44] is a tiered memory management system designed specifically for virtualization. It leverages Intel Page Modification and Logging (PML) to both assist in measuring page temperature and to migrate pages between tiers. Intel PML is a hardware-assisted technology that provides virtual memory tracking features for virtual machines. [21] It reduces the need for time-expensive traps needed for the hypervisor to manage a virtual machine's memory, making it a useful tool when designing a memory management system for virtual machines. vTMM scans page tables for the virtual machines and, with the help of PML, assigns a page temperature to each page and sorts them internally. vTMM then uses PML optimized page migration to swap pages between slower and faster tiers. This design allows it to minimize page access delays and traps and performs well even when multiple virtual machines are running on one host. Experimental analysis confirms the performance benefits. vTMM is outperforms AutoNUMA by over 200% using the graph500 [2] workload. vTMM solves the problem presented by some other systems that don't properly handle virtualization, and it presents an additional avenue for collecting real-time access pattern information that does not rely on changes to the host kernel.

2.1.3 Radiant

Radiant [33] is a tiered memory management policy that focuses specifically on the management of the page table. Systems with a large amount of memory available to them will also spend large amounts of memory on managing their memory subsystem, such as with page tables. The authors found that pages dedicated to managing the page table would sometimes end up in remote slow memory even when space was available in local fast memory. In addition, pages containing parts of a page table cannot be migrated between memory tiers, in contrast to data pages that can be migrated by an explicit NUMA policy or by the kernel itself. Even though the memory footprint of the page table is quite small relative to the total size of the system's memory, it would not be feasible to bind all page table pages to local memory tiers, because not all page table pages are frequently accessed. The strategy proposed in Radiant is based on the observation that three out of four page table pages accessed during a page table walk are in the upper levels of the page table tree, but these top levels make up only a small portion of the page table's memory footprint. Similarly, by actively monitoring page table access patterns, Radiant can chose to migrate last level page table pages between local and remote memory tiers. Radiant is implemented within the Linux kernel, and experimental analysis performed by the authors showed that it reduced the runtime of a set of synthetic and realistic workloads with large memory footprints by 20% on average. Radiant demonstrates the value of having different tiered memory management strategies for different types of pages and tasks.

2.1.4 Application-Attuned Memory Management for Containerized Workloads

Past research has focused on improving tiered memory management strategies for standard high performance computing workloads, where a single large task is allocated all of the resources on a machine; however, it is becoming more common to run high performance workloads in containerized settings. Arif et al. [17] sought to address this shortcoming in the literature. The challenge in implementing such a strategy is that co-located containerized workloads have differing memory access patterns, making a unified page placement and pre-fetching strategy untenable. Arif et al. propose an application-attuned strategy that is applied individually to containers. Each workload has access to API's to request allocation from specific tiers, and then the migration of these pages between tiers is managed transparently based on the observed hotness/coldness of those pages (based on a least recently used policy) and on information provided by the workload such as its latency sensitivity or lifetime. It will prefer not to migrate pages that the workload indicates are latency sensitive or have a short lifetime, even if those pages are found to be the least recently used. After implementing and integrating this strategy into the Linux kernel and into workload management systems such as SLURM [9], experimental analysis showed an increase in utilization of tiered memory and a reduction in workflow execution time by 35% when compared to the performance with default Linux tiered memory policies. This strategy is not fully transparent to the programmer, however, which may limit its applicability to workloads where altering the source code is unfeasible.

2.2 User-Space Systems with Online Analysis

These systems run in user space, but have some mechanism for identifying memory access patterns or page temperatures to make decisions based on dynamic runtime behavior.

2.2.1 CachedAttention

CachedAttention [25] is a strategy to reduce redundant computation of the Key-Value (KV) cache store for an LLM inference session. This store is used to reduce redundant computation of values for tokens in the context, which need to be re-used each for each generated token. It is created during the ”pre-filling” stage of the workload before inference begins. This cache is kept in high bandwidth memory accessible to the GPU (for GPU inference) but is only useful for that one session. In a turn-based LLM environment such as those used in OpenAI's ChatGPT [38], multiple users are served by the same inference engine. Each user submits a prompt–including the context of their conversation–to the inference engine which then generates a response. The engine then moves on to the next prompt, discarding the KV cache it generated for the previous client. Instead of discarding this KV cache, CachedAttention retains this store and offloads it to off-device storage for later use. CachedAttention intelligently pre-loads the KV cache for a returning user by integrating with the workload manager. To avoid straining the memory bandwidth between high bandwidth on-device memory and the SSD, it times the transfer to occur during the pre-filling stage for the new prompt. Experimental results showed an 87% decrease in delay before generating the first token and a 70% decrease in the cost of end-to-end inference. CachedAttention proposes an interesting strategy for offloading unneeded data until it is needed again, but it is not a strategy for managing the large memory footprint during inference itself as it only manages one piece of the memory footprint. It also requires alteration of the workload code which

may not be practical in all cases. This strategy is implemented at the software layer by altering the attention mechanism to support the cache. It could be complemented well by a system-level memory allocation and pre-fetching strategy, which would create opportunities for performance improvements regardless of internal attention optimizations.

2.2.2 HeMem

HeMem [40] is a tiered memory management scheme designed for big data applications. It leverages asynchronous access tracking and migrations to amortize the cost of these operations over time. HeMem also allows users to set memory management policies on a per-application basis. It runs in user space and does not require the installation of any kernel module or other alteration to the OS. To achieve this, instead of working within the kernel to track its page tables and NUMA data structures, HeMem intercepts certain memory-related system calls and registers itself with the kernel to manage page-faults that occur within the ranges it manages. It samples both processor event counters such as Processor Event Based Sampling (PEBS) to identify hot and cold pages and intercepted allocation calls to determine where pages should be placed when allocated. Migration and sampling are both handled asynchronously, and migration is offloaded from the CPU when possible using an I/OAT DMA engine. [34] Experimental results show that HeMem outperforms both Intel Optane DC Memory Mode [31] and Nimble [51], a kernel extension for tiered memory management, in common big data tasks where the hot set of memory fits into DRAM. However, in certain other cases, Intel Optane DC Memory Mode is more performant. While running in user space is convenient for applications where root access is not available, this design limits HeMem's ability to capture knowledge of all pages allocated, a weakness newer tiered memory management systems have addressed.

2.2.3 DyNN Offload

DyNN Offload [41] is a tensor migration strategy focusing on the management of GPU memory. It is designed specifically for dynamic neural networks [28], which allow the neural network structure to change across data samples, and are growing in popularity among the DL community. DyNN Offload runs a smaller pilot model that is used to make tensor pre-fetching decisions. This pilot model takes the input sample being given to the DyNN and predicts the execution of the operators for each execution block. This is then used by DyNN offload to pre-fetch the associated tensors into GPU memory. Any mis-predictions made by the pilot model can then be loaded on demand with a performance penalty, and the mistake can be noted and used to improve the pilot model training. The pilot model is originally trained based on static analysis of the DyNN training script. This process is transparent to the programmer. Experimental analysis showed that DyNN Offload outperforms the state-of-the-art Dynamic Tensor Re-Materialization [32] by 35%. It also allowed for the training of a model 8x larger than would be allowed with PyTorch alone, because DyNN offload makes it possible to work with less GPU memory, similar to how tiered memory systems allow for smaller local RAM. The management of GPU memory is a similar problem to the management tiered memory systems, in that GPU memory is smaller in size than overall system memory. A key difference, however, is that GPUs cannot directly access data stored in regular system memory, in contrast to tiered memory systems where accesses can be made to slow memory tiers for a performance penalty. In either case, DyNN Offload presents an interesting strategy for the migration of data, which may be applicable to tiered memory systems as well.

2.3 User-Space Systems with Offline Analysis

The set of tiered memory strategies that use offline analysis is most interesting to us. Systems using runtime analysis rely on information provided either by the kernel or by performance counters, which are not always portable between architectures. If the right performance counters are not available on the target system, then such strategies will not be applicable. Offline analysis is not always portable; however, it could be conducted on a per-enthronement basis to build an understanding of access patterns that are unique to the environment in which the workload runs

2.3.1 AutoTM

AutoTM [29] is an optimization tool designed to determine the ideal placement and fetching of pages for deep neural network learning operations. It is built on top of existing deep neural network training tool sets (specifically nGraph [22]) and does not require significant cooperation from the user. It is based on Integer Linear Programming [43] and takes advantage of the fact that many neural network computation graphs are static and contain no data-dependent control flow behavior. AutoTM automatically profiles the neural network provided to nGraph to determine ideal page placement strategies with the goal of reducing the execution time of the training with a constraint on the amount of DRAM used. Based on this analysis, it can also predict when to migrate pages between nodes based on pre-determined knowledge about the execution time of nodes in the computation graph and generate code that does so. It does this all during the nGraph compile time, with no additional analysis happening at runtime. The researchers were able to replace between 50% and 80% of their system's DRAM with persistent memory while observing only a 27.7% decrease in performance. This is compared to a 71.9% decrease they observed when making

the same change using the default first-touch NUMA page placement strategy. AutoTM's ease of use makes it more practical than some other systems that rely on cooperation from programmers to achieve their performance benefits, but is quite old now and nGraph has since been deprecated and merged into OpenVINO. [13] While no longer useful for real-world implementation, AutoTM still presents an interesting strategy to pre-calculate when pages should be migrated.

2.3.2 CXL Strategies for GPU-based workloads

Most of the strategies discussed focus on managing heterogeneous memory systems by controlling where pages are allocated and when they are migrated between tiers. Arif et al., however, propose a scheduling strategy for memory allocation in an attempt to minimize the overhead of data migrations [16]. Similar to DyNN Offload, this strategy focuses on GPU-based DL workloads; however, unlike DyNN Offload, it considers how heterogeneous memory systems such as CXL can be leveraged for this. The goal of this strategy is to determine an ideal scheduling of jobs within a workload management system such as SLURM such that there is sufficient memory at each tier and GPUs have available local fast DRAM to migrate data to during their workload. Experimental analysis showed a 65% decrease in overhead from data transfers. While this system is not designed around scheduling the movement of pages, its considerations of how all workloads in a shared system may compete for space in different tiers of memory is interesting and applicable.

2.3.3 DeepMemoryDL

DeepMemoryDL [15] is a page allocation and pre-fetching framework designed for DL workloads. It is integrated directly within TensorFlow [10] to operate transparently to the pro-

grammer. Since memory access patterns can vary between different phases of the same DL task, it analyzes the workload submitted to TensorFlow to automatically provide separate strategies at each phase of the training. DeepMemoryDL also creates and manages a fast, intermediate storage tier in CXL memory. This tier is used as a sort of fast scratch space for the cluster and assists in reducing the overhead of I/O stalls due to slower storage tiers. Compared to the default TensorFlow platform, a DL job using TensorFlow with DeepMemoryDL saw a performance improvement of up to 34% while seeing a performance improvement of up to 27% compared to CXL-based memory expansion strategies. DeepMemoryDL's transparent strategy is useful, but of course, it only works with TensorFlow.

2.3.4 NoPFS

NoPFS [23] is an I/O middle-ware designed specifically for distributed machine learning training to provide distributed caching policies that work across different datasets and storage hierarchies. Although NoPFS does not rely on a heterogeneous tiered memory, it is designed as a strategy to pre-fetch data before it is needed, making it relevant. It is not transparent to the user, but only requires a few lines of code in existing training scripts and no changes to DL frameworks. It uses clairvoyance [19] to analyze and predict exactly when a given DL job will access a data sample arbitrarily into the future. This allows it to automatically pre-fetch the data sample from storage on time. Experimental analysis showed that NoPFS was able to reduce the time lost waiting for I/O operations to complete and provided an end-to-end training time speedup of up to 5.4x on certain datasets. While NoPFS is also not a tiered memory management strategy, its use of clairvoyance to predict when a machine learning task would make an access is relevant to the strategy that will be proposed.

Chapter 3

Methodology

To identify patterns in memory accesses to exploit, the following process was implemented.

3.1 Environment

All testing was done on a single Dell R640 Power Edge server with two Intel Xeon Silver 4215 CPUs with eight cores each. The server had 348GB of fast RAM with an additional 1.5TB of Intel Optane DC Persistent Memory. It was configured with two NUMA nodes (one for each CPU socket), and the default Linux NUMA policy was in effect. The inference was run using llama.cpp [3] entirely using the CPUs. All inferences used Google's Gemma v2 [26] 27 billion parameter instruction tuned model. This model was chosen for no particular reason, but the size of the model was chosen such that the entire working set of memory would not fit in one NUMA node. The same model, prompt, and inference engine was used each time for consistency.

3.2 Heatmap Tool Design

The heatmap tool presented is designed in three interoperable layers which can be altered to match the needs of the analysis. These layers are represented in Figure 3.1. The first layer is the data capture layer, which is responsible for tracking memory accesses made by

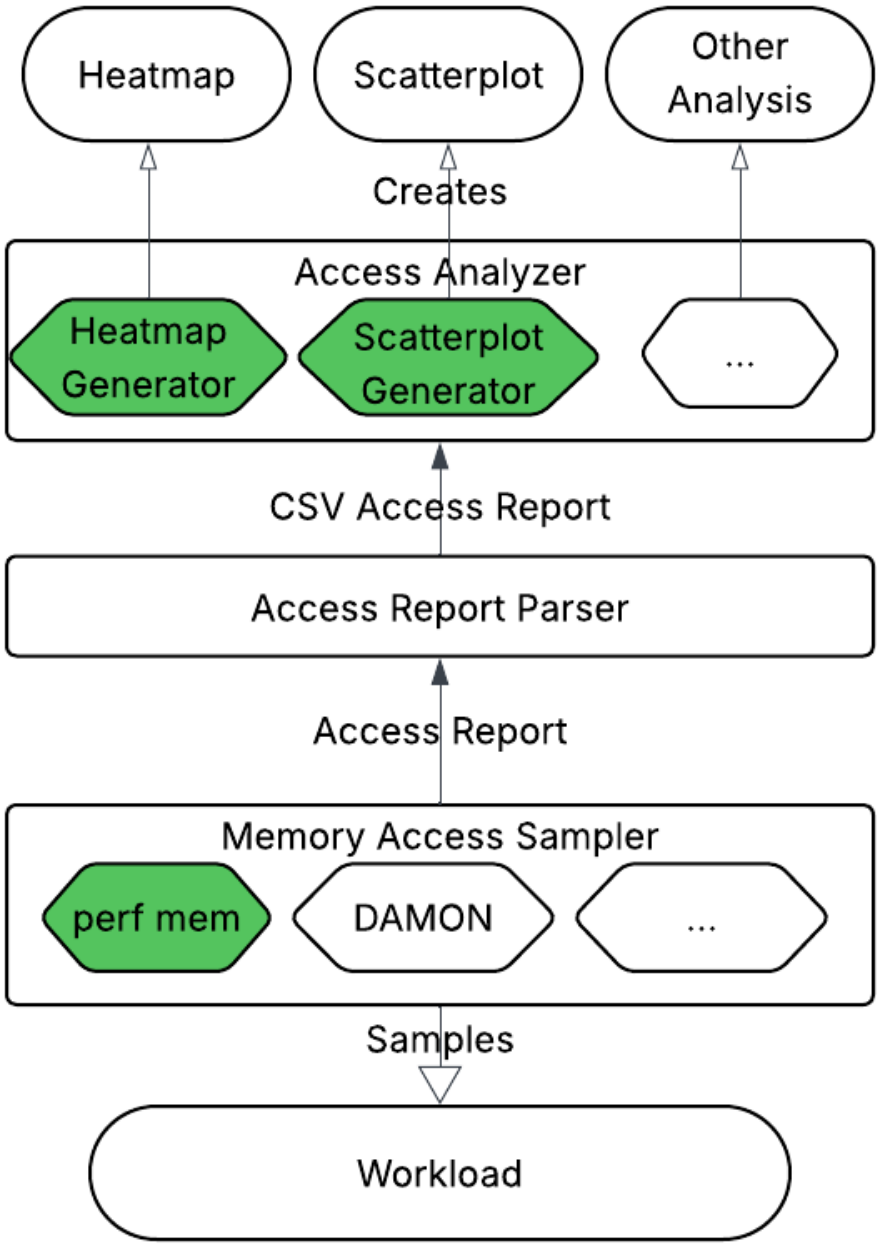

Figure 3.1: Diagram of the three layers in the heatmap tool. The implemented modules are highlighted in green, while potential future modules are also listed.

the process based on whether they were remote, local, or cache. Perf-Mem [6] was used to capture these memory access events. Perf-mem is part of the Perf [7] family of performance analysis tools and specifically records memory access events based on where in the memory hierarchy the address was stored (e.g. an L1 cache hit or a remote RAM hit). It can then report each of these accesses individually to be analyzed. It was selected for three main reasons: it was able to provide an easy to parse text document containing this report for the second layer to use; it easily reported to which tier of memory an access was made without requiring additional analysis; it was available on the machines used for testing. However it had significant drawbacks compared to other tools, such as Linux's Data Access MONitor (DAMON) [1].

Perf mem relies on hardware performance counters to track accesses, which means it is subject to potential sampling errors [52]. It is only compatible with the performance counters on Intel processors; memory events are not traceable on AMD EPYC processors with perf mem. Since the available GPU machines used AMD EPYC processors, analysis was confined to compute nodes with Intel processors, meaning the analysis could only be applied to CPU inference in this case. Most real world inference workloads happen on GPUs. Since perf uses performance counters in the CPUs to sample accesses, the GPUs could not be used to get an accurate understanding of accesses over time. GPUs have their own on-board memory, and their memory access patterns differ from CPU inference. However, many models are still too large to fit in on-board GPU memory, so several prior works discussed in Chapter 2 explore TMM strategies for GPU workloads; however, these needs will still differ from those of CPU workloads. Since most real-world LLM inference happens on GPUs, this restriction limits the universality of this analysis. Regardless, the broader pattern of how llama.cpp moves through its working memory should be informative. The reports of perf-mem are also not available during the execution of the process. This tool can then not be used for

live analysis, meaning any strategies relying on live analysis of memory accesses will have to use a different tool. In contrast, DAMON is useful for live analyses. Instead of leveraging hardware specific CPU performance counters to track memory accesses, DAMON relies on information from page table entries to count the number of accesses made to given pages. However, DAMON does not have an easy way to count accesses based on whether the access was made to a remote or local NUMA node as it is not NUMA aware. Finally, DAMON is not enabled by default on all systems and was not enabled on the system used in this testing.

Since perf-mem does not report the time of each access, recording time-based heatmaps requires a slightly different procedure. The PID of the workload process was captured and a new perf-mem recording was started and attached to the captured process ID. Perf-mem still needs a process to run, which is passed as a command line argument. In this case, perf-mem was told to run sleep [8] for the duration of the timing interval. For this analysis, time intervals of both one and 10 seconds were used, but this can be configured by the user. A while loop was used to keep track of whether the llama.cpp process was still running. As long as that process continued running, new perf-mem recordings were started and attached, each for the noted interval length. Once the llama.cpp inference was finished, reports of each of the recordings were generated and converted into text files. Depending on the length of the time interval, these files could be quite large, in some cases upwards of 50GB.

The second layer of the heatmap tool is responsible for parsing the accesses reported by the first layer. In the case of perf mem, this report is a large text file counting the number of accesses made to a certain virtual address and what level of the memory hierarchy (cache, local RAM, remote RAM) the access was mapped to. Since the file can be quite large, this layer parallelized parsing the report using an OpenMP script. To represent accesses, a methodology based on the work of Yoon et al. [53] was used. They presented a tool that tracked memory accesses in an embedded system and developed a model to identify access

patterns that indicated security violations. Their method for constructing a heatmap of memory accesses is used in this tool. The working set of memory was divided into large granules and the number of accesses to addresses within each granule was counted. The user is able to specify the size of the granules. The analysis script generated a csv file representing the number of remote and local accesses to each granule during each time interval. The user can also specify the start and endpoints of the tracked memory region, allowing them to "zoom in" on the regions of the virtual address space that are most interesting (such as the region associated with stack or heap memory). In this case, the parameters were set to cover the region of memory which spanned the largest size and had the most number of accesses from the process, which was observed to have belonged to BLAS [12].

One drawback to the current design is that Address Space Layout Randomization (ASLR) can interfere with the starting point of different memory regions. [45]. Although the access patterns within the region remains similar regardless of the ASLR impact, designing a strategy based around exact virtual addresses will not be feasible on systems with ASLR enabled. Instead, any potential pre-fetching strategy would have to discover the starting address for the regions it tracks once the process has started and base its strategy off patterns relative to this address. Since most real-world systems have ASLR enabled for security, this analysis kept ASLR enabled.

Once the granule size and the bounds of the tracked memory region were set, the script counted the number of accesses made to each granule and reported both the total number of accesses and the number of accesses made to remote NUMA nodes. The script at this stage can also be configured to include or exclude the number of cache accesses made to a given granule. Finally, this information is combined into a Comma Separated Value (CSV) file, which represents the number of accesses and remote accesses made to each granule, for every granule that had at least one access. For time-slice based heatmaps, the time slice

corresponding to each access count is also reported.

The third layer is responsible for taking the CSV output of the second layer and presenting it in a human-interpretable format. In this case, that means creating a heatmap image representing the memory accesses. To to this, a python script using matplotlib's imshow function was used [4]. This function has a limitation when creating very large heatmaps, as is common for this use case. The size of each granule must then be carefully chosen such that the image does not grow too large to be saved. The CSV file is taken and parsed into a sparse array where each index in the array represents the granule and each value in the array represents the number of accesses made. Since memory is a one-dimensional space, the data must be projected into two dimensions to display it as an image. This is done by reshaping the array of size n into a two-dimensional $N \times M$ matrix, where each row represents M consecutive elements from the array. The original array could then be constructed by concatenating the rows in order. The values of N and M are user configurable so long as the original array can fit in the matrix, but the default behavior is $N = M = \lceil \sqrt{n} \rceil$, which displays the region as a square, maximizing the amount of granules that can be displayed by the library.

This heatmap script in the third layer can be replaced by other analysis as needed. For example, this thesis also makes use of scatter plots as an additional means to analyze access patterns. Future work could train a machine learning model on the patterns observed as well. The multi-tiered modular approach makes it easy to alter individual components based on the source of the input or the needs of the output.

Table 3.1 shows the various parameters the user can set for each layer of the heatmap tool. Since different stages have different purposes, the nature of the configuration at each layer is different. The access sampler layer described above is the lowest layer. As noted above, this layer is not implemented here, but rather the results of the existing tool chains can be used

Layer	Parameter	Unit
Heatmap Generator	Shape of Heatmap	# of rows / columns
Scatterplot Generator	Range of Granules Displayed	Range of Granules
Access Analyzer	Granule Size	Bytes
	Start of Tracked Memory Region	Address
	End of Tracked Memory Region	Address
	Type of Access to Track	Local/Remote/etc.
perf mem	Time Slice Length	Seconds
	Events to Record	loads/stores
	Accesses to Record	eg. kernel/user

Table 3.1: User-Configurable Parameters for the Heatmap Tool

for analysis by the memory access heatmap tool. At the perf mem layer, users can specify the types of events to record, such as memory loads or memory stores, and the types of accesses to record, such as kernel or user accesses. For the purposes of this analysis, user-space loads were recorded. For the time-based analysis, this layer is also responsible for controlling the time slice length. Users can specify how long each time slice is in seconds.

The Access Analyzer is responsible for parsing the perf mem report and generating a CSV file. At this stage, users can specify the bounds of the memory region to track. As noted before, the interesting memory regions often move around due to Address Space Layout Randomization, so this needs to be set for every new recording made. Users can also specify at this stage if they are recording local or remote NUMA accesses. Finally, this stage is where users select the size of each granule, which is represented in bytes.

The final layer is responsible for displaying or analyzing the generated CSV file. Many different analyses could be conducted on the reported data, for example the heatmap or scatterplot used in this thesis. The exact parameters alterable by the user differ between each type of analysis. For example, the heatmap tool can be configured to display the image with varying dimensions, which is used in this analysis to identify the periodicity of the access pattern. The scatterplot is also used with user-configurable parameters for the range

of granules that are reported. This is helpful to zoom in further to see access counts for neighboring granules.

The tool can be invoked from the command line. Each step is executed individually so that the parameters can be set at each stage. The steps to use the heatmap tool are enumerated below:

1. The memory access sampler is run on the target workload. In this case, a perf mem report is recorded with the command line flags associated with the options the user wishes to configure as shown in Table 3.1.

2. The perf mem report is converted into text and provided to the access report parser. Here, the user can set the desired granule sizes and region bounds. For the first analysis, its often easier to capture the entire address space with large granules. This allows the user to identify the bounds of the stack and heap regions or identify other regions of memory the process is making many accesses to. Once the interesting regions are identified, the user can zoom in on them and use a lower granule size.

3. The user then selects which last level analysis they want to conduct. For the heatmap tool, this would be a heatmap generator. This generator takes the CSV generated by the parser and creates a heatmap from that data. The user can experiment with different dimensions of the heatmap to identify any strides visible in the pattern.

4. It is easy to re-use earlier steps of the analysis to experiment with different parameters. For example, if a user wants to change the granule size of the final heatmap, there is no need to re-run the perf mem step. The parser can be re-run with a different granule size using the already existing access report, and a new heatmap can be generated with the different granule size.

3.3 numactl Testing

In addition to looking for a pattern in memory accesses made by llama.cpp, it was important to understand how different page placement policies impacted performance. Two basic strategies were attempted to see how performance characteristics changed based on the placement of pages relative to the CPU requesting those pages. One strategy was to bind all threads and memory allocation to the same NUMA node; the other was to bind them to separate nodes. These were compared to a third baseline control trial where no memory node was specified but a CPU node remained specified for consistency in the number of threads available. To do this, the numactl [5] utility function was used. numactl provides command line flags to control both the CPUs the task was bound to as well as the NUMA memory nodes the kernel prioritizes. This tool makes it possible to override the default Linux NUMA page placement policy discussed in Chapter 1 and test how significant of an effect page placement policies have on performance. Naturally, we would hypothesize that the same-node strategy would significantly outperform the different-node strategy. It is also reasonable to expect the Linux default to perform at least as well or better than both these strategies.

To test the importance of NUMA policies and the potential negative impact a bad NUMA policy can have, the STREAM benchmark [37] was used to identify memory bandwidth and other performance characteristic differences between the strategies. Prior works [20, 50] have explored using the STREAM benchmark with heterogeneous memory systems and with CXL systems; however, these analyses often differ between specific hardware configurations. To explore the impacts of the differing NUMA policies on the specific hardware setup used for this investigation, the STREAM benchmark was run under the three NUMA strategies described, and the results are described in detail in Chapter 4.2.

Because the results of an LLM inference are based partially on random numbers and are non-deterministic, additional care must be taken to get meaningful results from this testing. Even the length of the output between trials can vary, which can have significant impacts on the workload runtime. To solve this problem, llama.cpp allows users to specify an inference temperature, which represents how responsive the model is to those random seeds, where a value of 0 would have deterministic results and a maximum number of tokens to predict. For the purposes of this test, the temperature of the inference was set to 0 and the maximum token count was set to 100 so that trials could be run more quickly. In addition, the thread count was set to eight for all cases, ensuring there were only as many threads as there were physical CPUs on each NUMA node.

Testing shown in Chapter 4.2 showed that the performance penalty on the different nodes strategy was significantly less than expected based on the STREAM benchmark results. Further analysis showed the difference was likely caused by llama.cpp benefiting from a warm start, meaning llama.cpp had recently been run with the same model. A previous execution of llama.cpp allowed the system to cache parts of the executable and model, improving startup times. To ensure that llama.cpp started from cold without the benefit of these caches, the system was restarted after each trial. The results from this experiment are also described in Chapter 4.2

Chapter 4

Results

The methodology discussed in Chapter 3 was implemented using the Dell R640 Power Edge server with two NUMA nodes as previously discussed. A series of heatmaps were captured and several interesting patterns were observed. The NUMA strategy comparisons were also tested on the STREAM benchmark and the real world llama.cpp inference workload with both warm and cold-start conditions. Their results are as follows:

4.1 Heatmaps

As expected, the heatmaps generated using the methodology discussed in Chapter 3.2 showed a pattern of accesses.

Figure 4.1 shows such a clean pattern. The lowest address is represented in the top left corner of the figure, and the highest in the bottom right. The figure can be read left to right, top to bottom to understand the access pattern over a single-dimension region of memory. Each row of the heatmap represents consecutive granules, and the original one-dimension array can be recreated by concatenating each row together in order. The heatmap is only displayed in two dimensions to better show large regions of memory.

In the figure, there is a clear periodicity in the accesses. The dimensions of the heatmap were selected such that the number of granules in each row was 135. The granule colors line

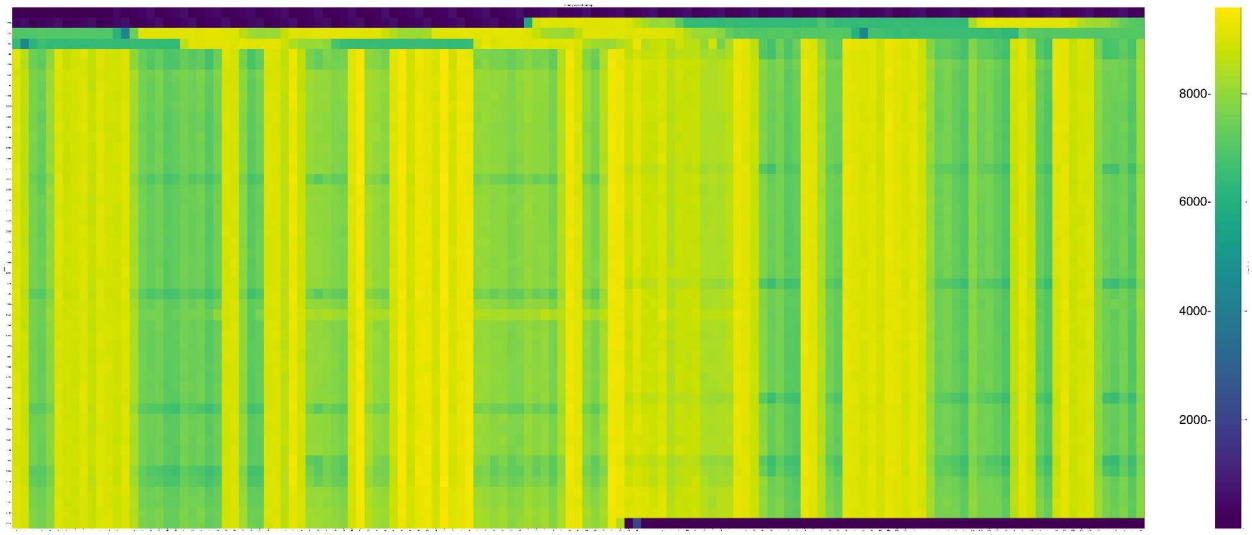

Figure 4.1: Memory access pattern for llama.cpp during the inference phase and not including the initialization phase. Each granule is 16MB in size.

up to demonstrate that there is a stride in accesses of approximately 135 granules.

In the case of Figure 4.1, the memory region pictured is approximately 100GB, and each granule is 16MB in size. This means that the stride is approximately 2GB in length. When viewing the perf output for this analysis, the region of memory pictured is said to belong to BLAS [12], indicating that this region is a series of matrices and their accesses are part of the matrix operations that underlie any neural network. The pattern displayed in the figure is consistent with such a matrix operation in that more accesses are made to the vector operand and product vector than the matrix operand in a matrix-vector multiplication.

To test this hypothesis, a test script representing a basic matrix-vector multiplication was written and analyzed using the memory heatmap tool. This script multiplies a 10,000 by 10,000 matrix of double precision floating point numbers by a 10,000 dimensional vector. The total size of these vectors and matrices is then over 800MB. The BLAS matrix-vector multiplication function is used because the same function is used by llama.cpp. The matrix-vector multiplication function was repeated several thousand times to ensure there were

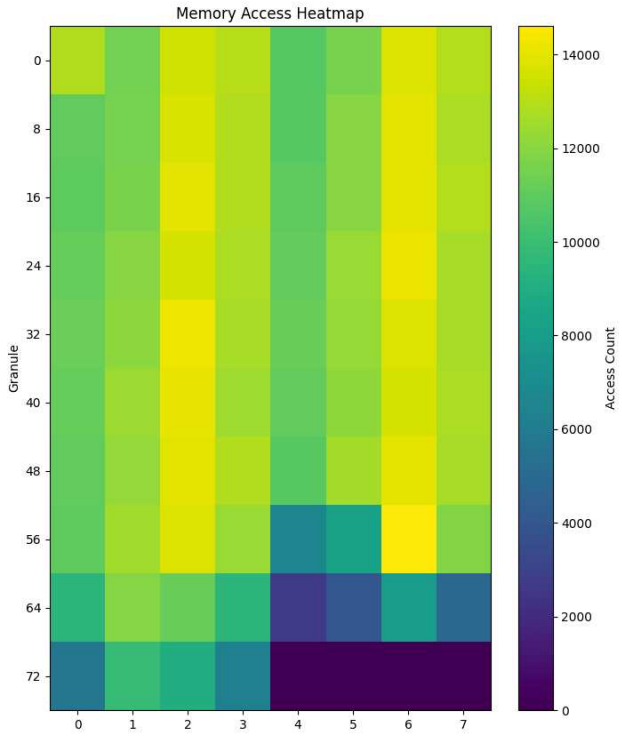

Figure 4.2: Memory access heatmap for an example matrix-vector multiplication

enough samples recorded by perf mem to accurately capture the spatial access patterns of matrix multiplication. Figure 4.2 shows the resultant heatmap of this analysis. The region of memory displayed represents the bulk of all accesses made by the process. Each granule is 1KiB in size, and the tracked memory region is only 80KiB in size. This is significantly smaller than the 800MB the combined matrices and vectors should take up; however, 80KiB is the size of the multiplier vector in the matrix-vector multiplication. It makes sense then that the vast majority of memory accesses would be made to this vector, as each call to the matrix-vector multiplication function would involve 10,000 accesses to this one vector, but only one access to each row vector of the multiplier matrix. Note also that only load events are tracked for this analysis. We would expect to see a similar result with the product vector if store events were recorded. However, recording both loads and stores simultaneously did not work when tried. This analysis focuses on load events because it is interested in ensuring

data is available to read when needed.

As we can see in Figure 4.2, there is a clear periodicity similar to what was seen in 4.1. The vertical stripes seen in that figure are present here, meaning the pattern observed is consistent with a matrix operation. The periodic variations in the number of accesses would then represent the underlying implementation of the BLAS library and how it moves through large vectors and matrices. In the case of Figure 4.2, the figure generated such that there are 8 columns, and the granules line up indicating that the periodicity in this case is eight. Since each granule is 1 KiB, the periodicity observed is 8 KiB. However, further testing revealed that the pattern observation held when the number of columns was reduced to four, meaning the actual periodicity is just 4KiB. However, this image is too narrow to be displayed legibly on the page; the eight column version still demonstrates the periodicity.

A similar pattern is shown in Figure 4.3. In this image, single granules with extremely high access counts that are a near constant offset from each other are visible. This is clearly the result of a series of accesses with constant strides. Each of these high access count granules is offset by 540 granules. To verify this, Figure 4.4 has a row width of 540 columns and shows a similar pattern to that seen in Figure 4.1. Although it is hard to see individual granules due to how wide this image is, it does confirm the hypothesis that the access pattern repeats every 540 granules. In both figures, each granule is 4MiB in size, meaning this stride is approximately 2GiB long. This granule size was chosen because, as discussed in Chapter 3.2, smaller granules would cause the script to generate an image too large for it to handle.

Figure 4.5 is a one-dimensional representation of the same data and demonstrates this stride even more clearly. While there appears to be an underlying pattern, there are also layers of granules that have access counts much higher than those around them. The upper layer has samples spaced approximately every 540 granules (2GiB) from each other. There is

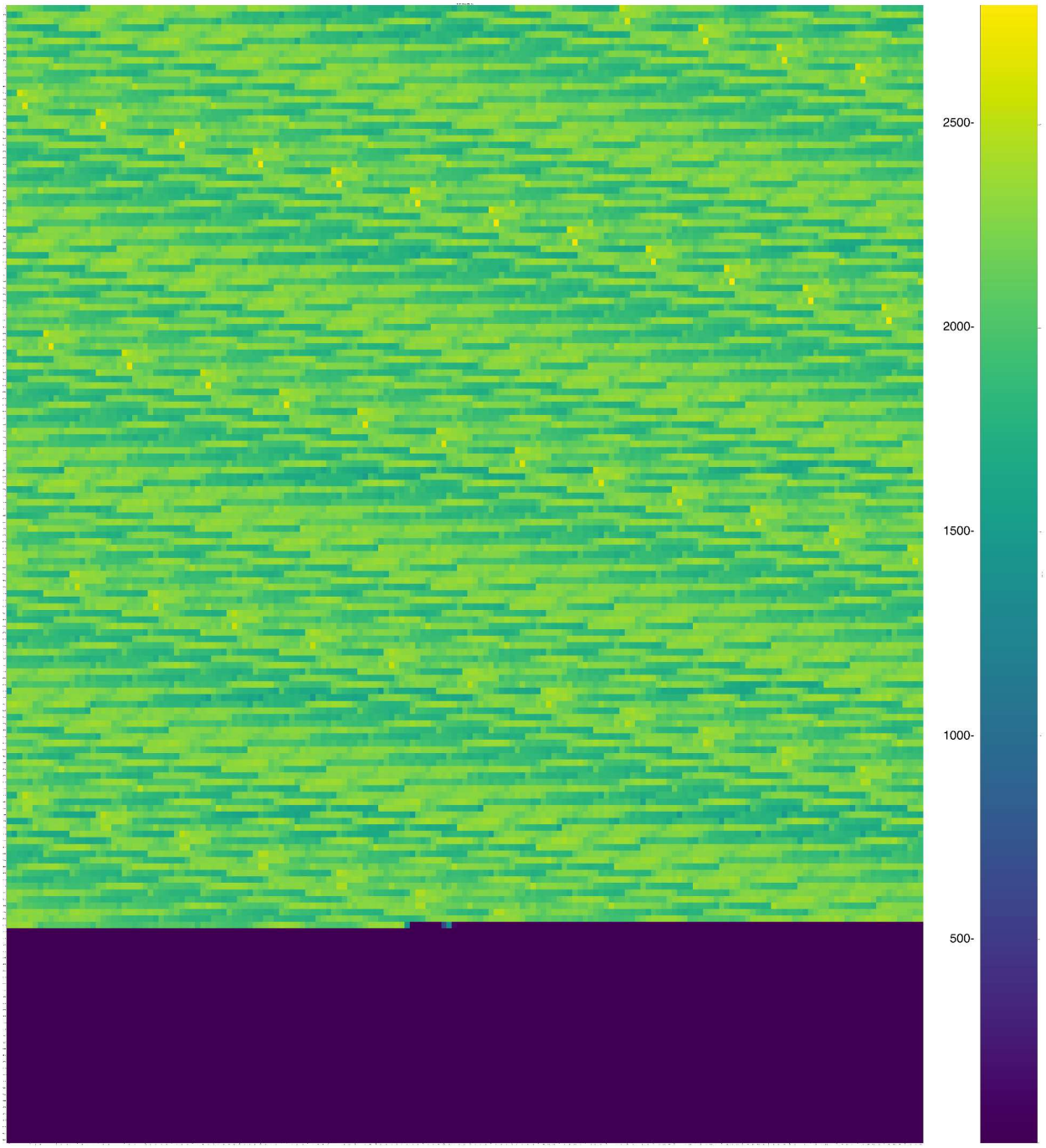

Figure 4.3: Memory access pattern for the larger working region of llama.cpp over its lifetime with 4MB granules.

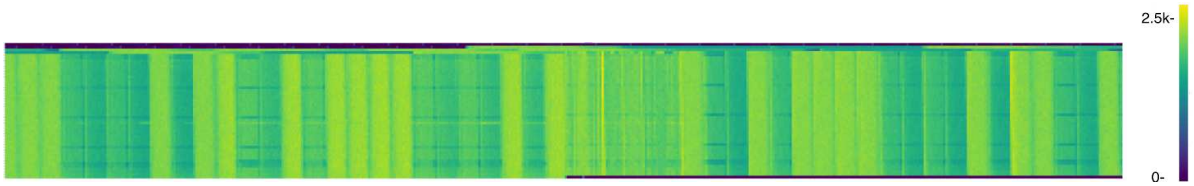

Figure 4.4: Figure 4.3 heatmap with 540 columns.

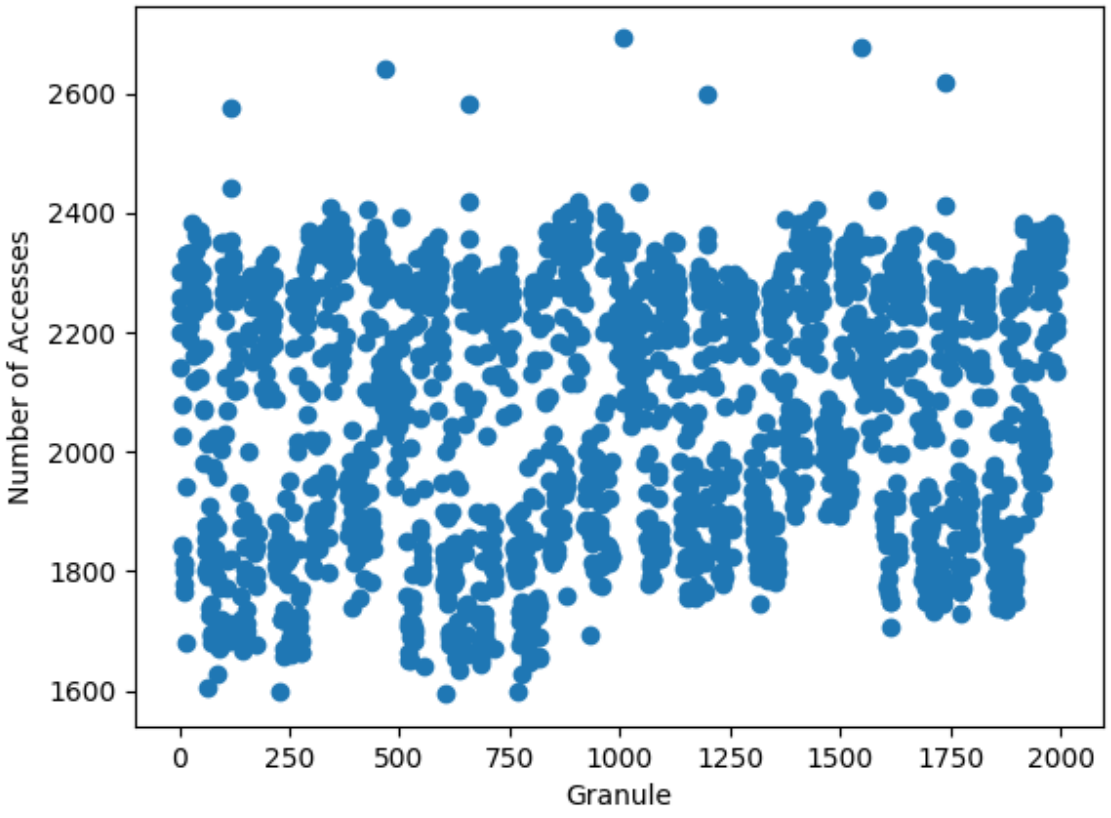

Figure 4.5: Scatterplot of access counts to a snippet of the working region of llama.cpp.

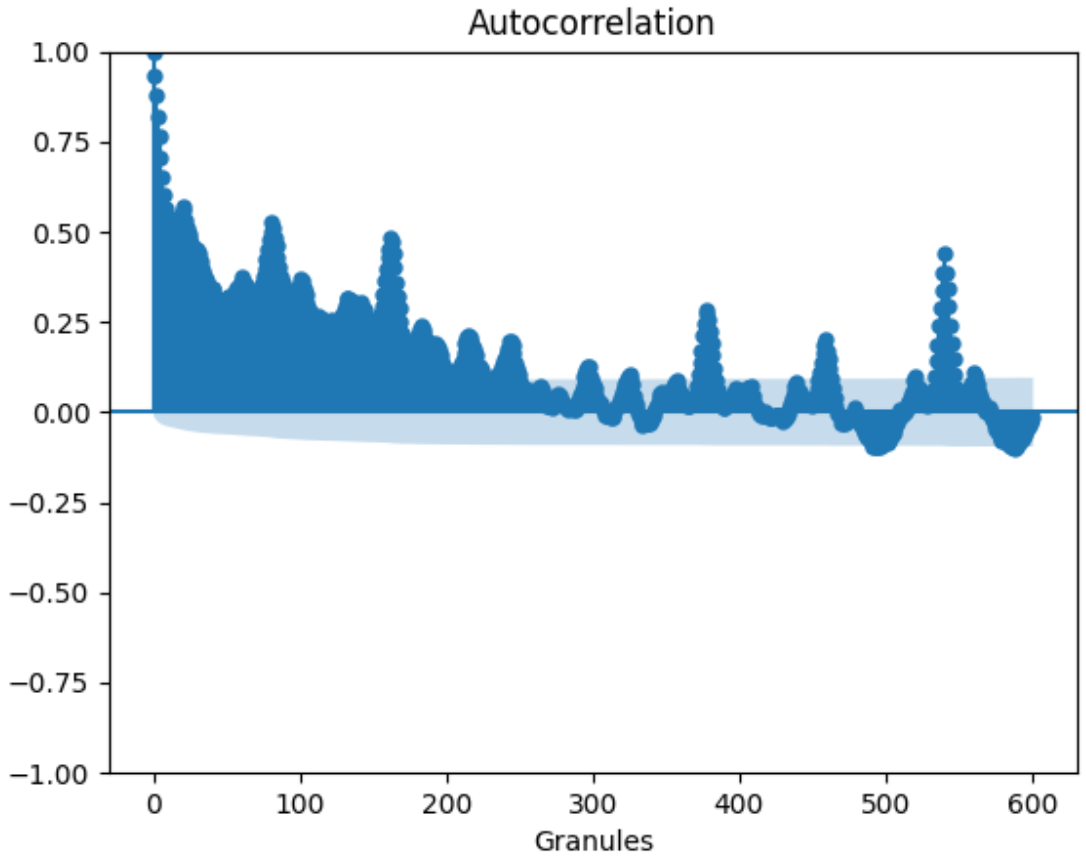

Figure 4.6: Autocorrelation graph for the heatmap displayed in figure 4.3

an additional lower layer, which appears to shadow the upper layer. This lower layer has a similar stride itself but appears to be offset by 180 granules (720 MiB) from the upper layer. There also appears to be an interesting pattern in the other granules as well. This is somewhat visible in the heatmaps, but there are waves of higher and lower access counts even within a 540 granule period.

As noted in section Chapter 3.2, the heatmap tool allows for other analyses to be applied to the access data, such as the scatterplot shown in Figure 4.5. One type of analysis that can be useful for this analysis is the autocorrelation function [24]. Autocorrelation analysis is generally used in time series data for machine learning. It measures the linear relationship

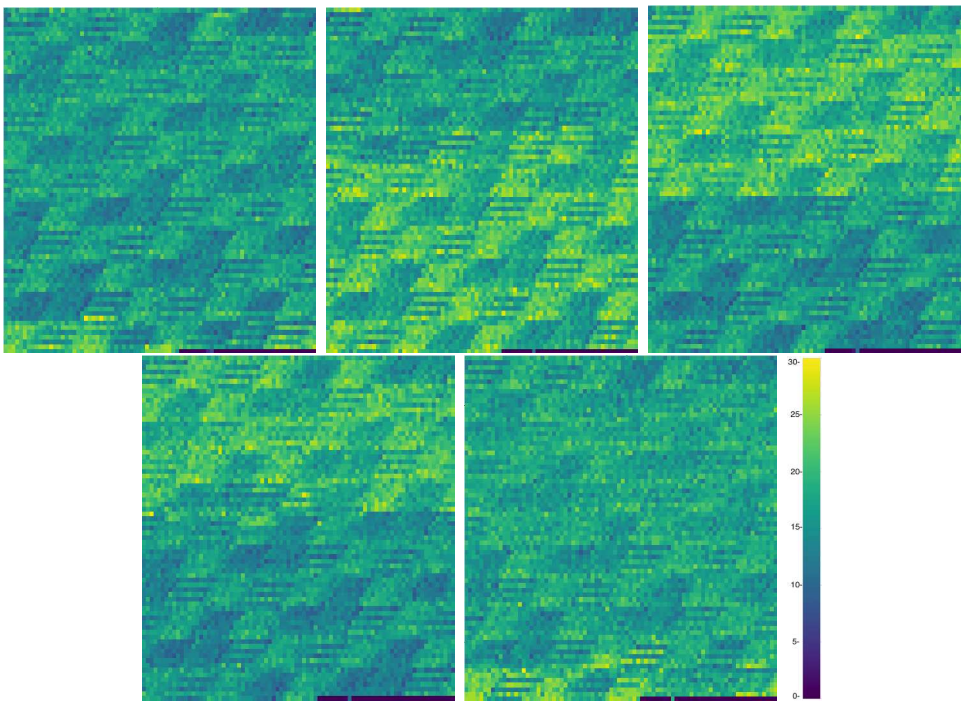

Figure 4.7: From left to right: Memory access heat maps of the working region of llama.cpp for consecutive 10 second intervals during the inference phase.

between a time-series function and lagged versions of itself. Its often used in machine learning contexts to identify periodicity in time-series data. We can treat our memory space as a time-series to identify recurring patterns in how parts of our memory space are accessed. Figure 4.6 shows the graph of the autocorrelation function for the same data used in 4.3. Spikes in the graph indicate that the underlying data repeats with that period; the taller the spike, the stronger the correlation for that period. As we can see in the figure, there is a large and prominent spike at 540 granules, confirming the periodicity observed in 4.4. However there are several other spikes indicating some patterns with smaller periodicity. Most of these are not as strongly correlated as the 540-granule period observed, but it could still be leveraged by a hypothetical pre-fetching model to identify pages that are likely to see more accesses.

The prior figures explored spatial access patterns in llama.cpp to identify the hottest regions of memory over the entire course of inference. However, a pre-fetching strategy will need

to understand when pages will be accessed, not just which pages will be accessed. Luckily, as discussed in Chapter 3.2, the heatmap tool supports capturing memory heatmaps for individual periods of time through the inference process. Figure 4.7 shows such time-sliced heatmaps. Read from left to right, the heatmaps reflect a similar pattern to that shown in Figure 4.3. Interestingly, nearly the entire memory region is accessed in every time slice. While access counts are obviously much lower in Figure 4.7—particularly because Figure 4.3 counts accesses made over all time slices, of which there are over 100—no memory region goes truly untouched. There is, however, a clear hot spot within working memory. With the time slices side by side, it is clear to see how this hot spot propagates through the region from high memory addresses to low memory addresses. The full size of this hot spot is quite large, as we see in the second and third time slices. This may make it hard to exploit the pattern if the hot spot is too large to store in local fast memory. In addition to the temporal access pattern, there is still a spatial access pattern similar to what was seen in Figure 4.3. Even for regions of memory outside the hot spot of accesses, there is a clear pattern of high access regions offset by some constant stride, similar to what was seen in the previous figures.

The prior analyses explored the memory access patterns for all accesses made by llama.cpp, including any accesses to the cache, local RAM, and remote RAM. This is useful for understanding how llama.cpp accesses memory and demonstrating the capabilities of the heatmap tool, but it is not sufficient for developing a pre-fetching strategy specific to llama.cpp. For this purpose, we need to track only the accesses to local and remote RAM that llama.cpp makes. We also need to simulate the environment that llama.cpp would experience on a CXL system. This is done by limiting llama.cpp's execution to one NUMA node of CPUs, allowing the remote NUMA node to represent the remote memory that would be present in a CXL system.

Figure 4.8 is a heatmap for llama.cpp run under those conditions. Just as seen in Figure 4.3,

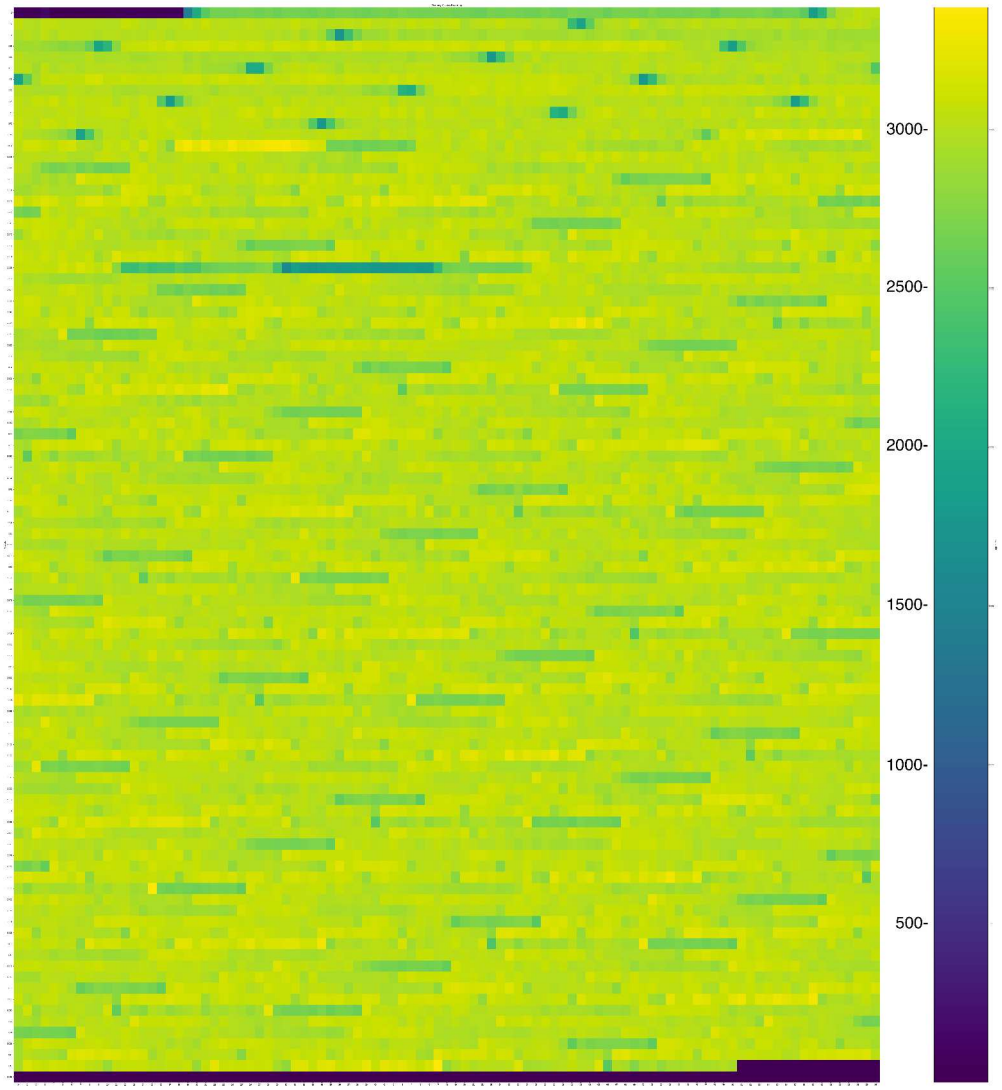

Figure 4.8: Memory access pattern for llama.cpp simulating a CXL system and ignoring accesses to the cache. Each gramule is 4MB.

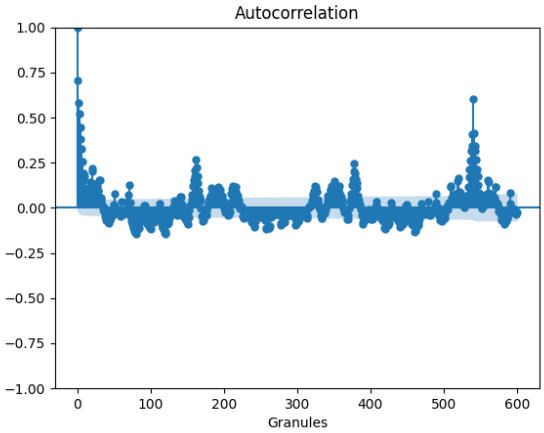

Figure 4.9: Autocorrelation function for the heatmap shown in 4.8.

each granule is 4MiB in size, and there is a periodicity of 540 granules. The periodicity is confirmed by an autocorrelation analysis shown in Figure 4.9. The autocorrelation shows strong periodicity at 540 granules with less intense periodicity at 190 and 390 granules. It is unsurprising that the stride in memory accesses is unchanged between this trial and that of Figure 4.3 given that the workload did not change. However, the relative intensity of the hot granules is noticeably less in this new heatmap. This is also unsurprising given that this heatmap does not include accesses made to the cache. We would expect that granules receiving the highest number of accesses would be present in cache most often, so a heatmap which discounts cache accesses would under-count accesses to these granules. Despite this, the spikes in the autocorrelation graph stand out more than in 4.6. In that figure, the tallest spike representing the 540 granule periodicity reaches only 0.5, while it reaches 0.6 for this new recording. A value of 1 would indicate that the access counts are perfectly repeating. In this case, the stronger autocorrelation indicates that the access pattern is more predictable when considering only accesses to remote and local RAM, which could be useful when designing a pre-fetching strategy based on these patterns.

Figure 4.10 shows the same trial as 4.8, but now only considering accesses made to remote

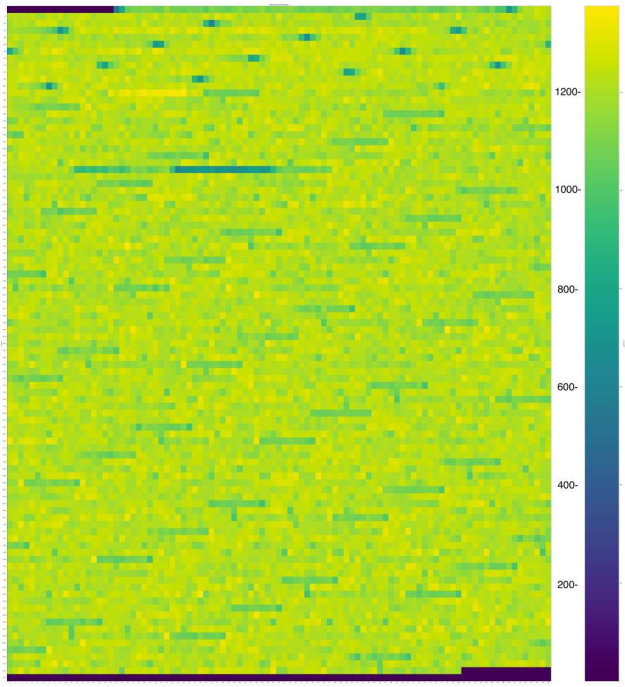

Figure 4.10: Remote accesses made by llama.cpp for the same trial as Figure 4.8.

NUMA nodes. The images could be overlaid as each heatmap shows the exact same granules in the exact same places on the heatmap. Notably, the remote access heatmap does not look different from Figure 4.8. In fact, while the remote access heatmap appears slightly noisier when placed side-by-side, the access count for each granule has a nearly perfectly linear correlation with the remote and local heatmap. Approximately 40% of all accesses made to a given granule were made to the remote NUMA node. [1] Since the same granules are relatively hot in both heatmaps, we can see that the default NUMA strategy is not accurately identifying hot pages and ensuring they are available in local memory before they are needed. This shows that there is room for improvement in the page placement policy, perhaps through a pre-fetching strategy that takes advantage of the knowledge we've gained about the access patterns for llama.cpp.

As noted earlier, a pre-fetching strategy would need knowledge about when accesses will be

[1] Note that the scale of the legend for Figure 4.10 is different from that of 4.8.

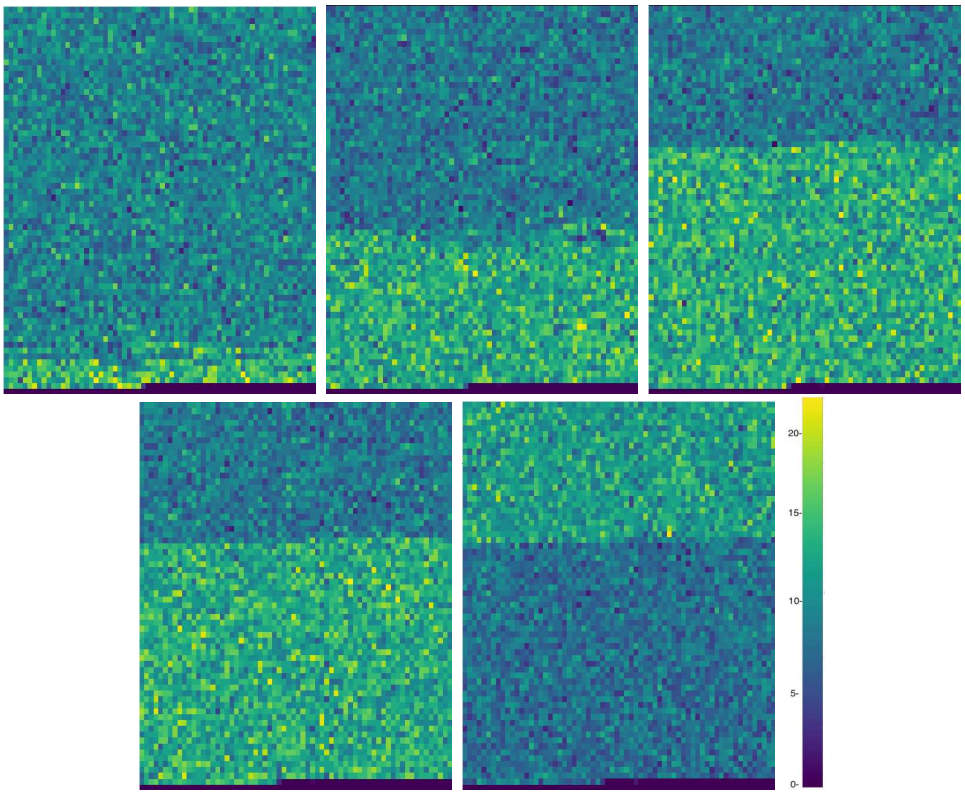

Figure 4.11: From left to right: Memory access heat maps of the working region of llama.cpp simulating a CXL system for consecutive 10-second intervals during the inference phase.

made in addition to an understanding of which pages will see the most frequent accesses. Figure 4.11 shows how the access pattern of the CXL simulation of llama.cpp changes over the course of 50 seconds in the middle of the inference process. Here, we can clearly see the same hotspot as shown in Figure 4.7. This hotspot seems to move throughout the tracked memory region, as seen before. Also noticeably missing from these heatmaps is the clear spatial pattern observed before. It is possible that the spatial pattern is less obvious in these recordings because of how few accesses are observed for each granule. In the heatmaps counting all accesses over the entire course of the inference, several thousand accesses were made to each granule. For these heatmaps, fewer than 20 accesses were made to each granule. These heatmaps may simply be more sensitive to noise in the sampling data. Another potential cause could be that these heatmaps ignore accesses made to the cache, while Figure 4.7 includes those accesses. Granules that see higher access counts are more likely to appear in the cache, as observed in Figure 4.8.

Figure 4.12 shows the remote access heatmaps for the same periods as Figure 4.12. Here, we see the same temporal access pattern we saw for the same period when tracking accesses to both remote and local NUMA nodes. This indicates that the default NUMA policy is not adapting to the access patterns made by llama.cpp, just as seen in Figure 4.10. Hot pages are frequently being trapped in remote nodes, increasing the average memory access time. While the temporal pattern remains visually obvious, the remote access heatmap does appear noisier than its counterpart, just as Figure 4.10 appeared noisier than its counterpart. Again, this can likely be explained by the fact that fewer accesses are being made to each granule. In this case, the ceiling of the range of accesses is 16 per granule, compared to 20 for the heatmap tracking both local and remote accesses. This is a noticeably larger proportion than seen in Figure 4.10. In that figure, roughly 40% of accesses were made to remote NUMA nodes, but for this case that number is 80%!. This proportion seems to

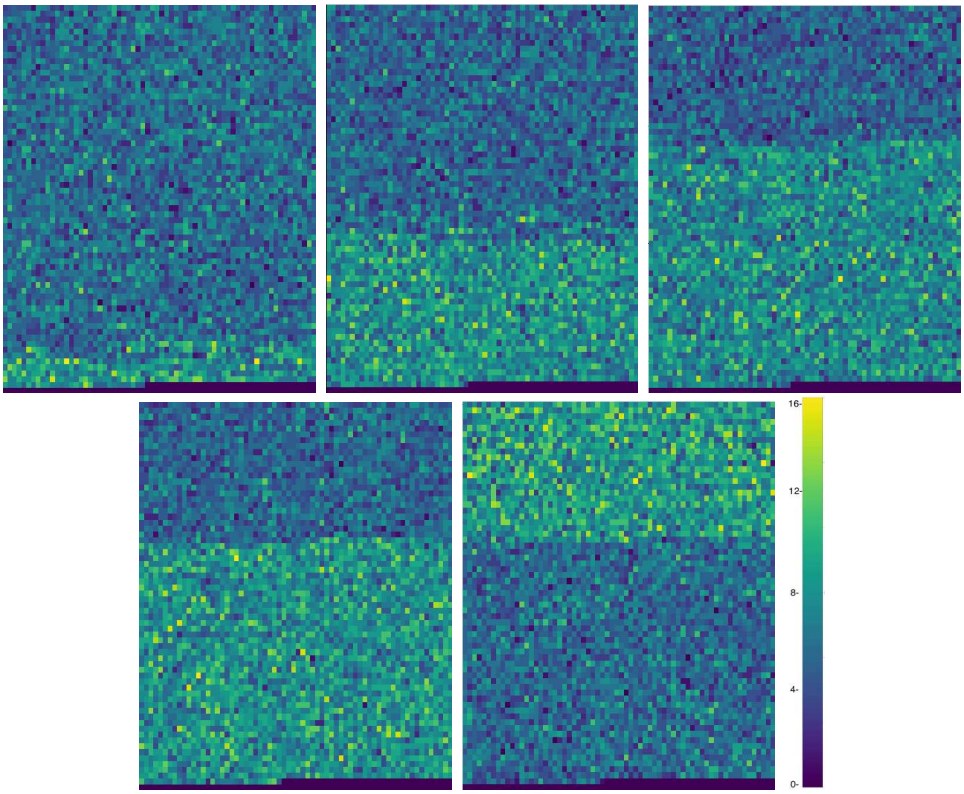

Figure 4.12: From left to right: Remote memory access heat maps of the working region of llama.cpp simulating a CXL system for the same consecutive 10-second intervals as shown in Figure 4.11.

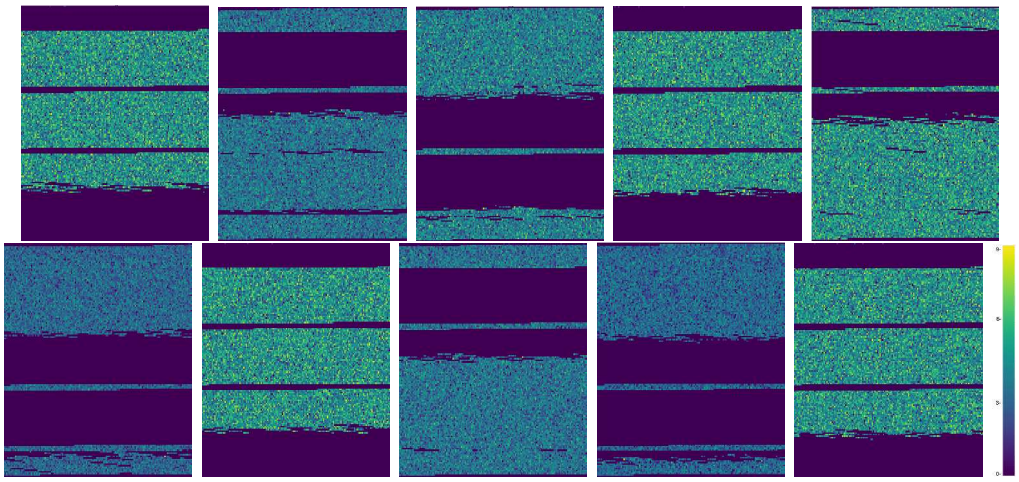

Figure 4.13: From left to right: Remote memory access heat maps of the working region of llama.cpp simulating a CXL system for consecutive 1 second intervals.

change over the course of the inference.

In an effort to better understand the temporal pattern shown in the previous figures, Figure 4.13 shows an additional set of time-sliced heat maps for llama.cpp, this time with 1 second time slices. Here we can see the hot zone much more clearly. Unlike in the previous timed heat maps, the entire working region of memory does not receive accesses during every time slice. Some parts are cold during some intervals, while they are hot in others. This fact could be leveraged to offload unneeded pages during the intervals they are not needed and to pre-fetch the pages that will be in use. We can also see that the periodicity of the movement of the hotspot is much faster than what was implied in Figure 4.11. This is not surprising. The inferencer predicts each new token much faster than every 10 seconds, so we would expect to better be able to predict the accesses with a shorter time granules. Finding the balance between short enough time slices and ensuring there is enough time to pre-fetch and evict pages will be important for further research into pre-fetching strategies.

Strategy	Copy	Scale	Triad	Add
No Mem-bind	47,171.1	49,180.0	53,784.7	53,778.4
Same Nodes	47,062.7	49,144.0	53,780.6	53,802.0
Different Nodes	24,571.7	20,493.6	24,020.5	25,478.2

Table 4.1: Results of STREAM benchmark of memory bandwidth for each NUMA strategy, reported in MB/s. Higher is better.

4.2 numactl

4.2.1 STREAM Benchmarking Results

The results of the stream benchmark analysis described in Chapter 3.3 are provided in Table 4.1. Unsurprisingly, the different node strategy incurs a significant bandwidth penalty. The different node strategy performs 48% worse overall than the same node strategy. Also unsurprisingly, there was minimal difference in bandwidth between the various operations. This same observation was made in [20], where Bergstrom observed that the primary constraint on these operations is the memory bandwidth, not the CPU performance. Although this observation is similar to those of previous works, it is still valuable to note the specific performance benchmarks observed on this machine to assist with reproducibility.

4.2.2 Impact of NUMA Strategies on LLM Inference

The process discussed in Chapter 3.3 was applied for 30 trials; 10 trials where only a CPU node binding was specified, 10 trials where the CPU was bound to same node as memory, and 10 trials where the CPU was bound to a different node than memory. llama.cpp was set with a temperature of zero on eight threads and was limited to predict only the first 100 tokens.

Table 4.2 shows the result of the experiment. The mean and standard deviation of the real

Strategy	real_time	real_stdev	sys_time	sys_stdev
No Mem-bind	266.4257	0.215	17.4554	0.363
Same nodes	264.5671	0.092	7.4554	0.031
Different nodes	266.0762	0.098	7.995	0.036

Table 4.2: Average and standard deviation of real and system times in seconds over 10 warm-start trials of llama.cpp for each strategy. Lower is better.

and sys times reported by the time [11] utility are reported in seconds. Based on these results, several interesting observations can be made.

First, while pushing all allocations to the remote NUMA node has a negative impact on performance, this impact is not nearly as much as we would expect based on the results from the STREAM analysis. This difference may be small, but an independent T-test for means was performed and yielded a p-value of 7.09×10^{-12}, which demonstrates that this difference is statistically significant. Further analysis is needed to discover why the performance impact was so much smaller than one would expect.

Second, the performance of the workload is slightly worse when not specifying a NUMA node for allocations and instead letting the kernel manage this with its default policy. This difference is small, but a T-test yielded a p-value of 0.0004. This is statistically significant, albeit not as strong as the difference between the two other allocation policies. It is not clear why this would be. In all cases, the threads were bound to the CPU from node 0. The only difference between each trial group was whether and where the memory binding was set. How is it that the default strategy can perform worse than forcing all memory allocations to go to the remote node?

Third, there is a significant difference between the system times for the default CPU-bind-only strategy compared to the system times for the two other strategies. This may be related to the previous observation of the default strategy being slower than the others. It is worth noting that while real and user times are still slower, they are not as significantly slower as

Memory Access Type	Overhead (%)	Number of Samples
LFB or LFB hit	47.18	4086333
Remote RAM hit	33.62	1648952
L2 or L2 hit	9.47	1643316
Local RAM or RAM hit	8.81	723280
L1 or L1 hit	0.48	84443
L3 or L3 hit	0.43	51652
L3 miss	0.01	1181

Table 4.3: perf mem report for llama.cpp running on the different node strategy with a warm start, sorted by the type of memory access. The overhead in this case represents the percent of samples in this category

the system time. This may indicate that some work that happens in user time for the other trials happens in system time for this case, but it is not clear what that work would be.

To ascertain why the overall performance impact was so much less than expected, perf mem was used to understand the types of accesses being made by llama.cpp when it was run with the CPU and memory placement being bound to separate nodes. Table 4.3 shows the result of this report. It is clear that a plurality of accesses are made from the Line Fill Buffer (LFB), with remote accesses coming in second. These dwarf the number of accesses made to the local NUMA node. There are a variety of reasons some accesses are made to the local NUMA node even though the strategy attempts to push all allocations onto the remote NUMA node. The likely culprit in this case is shared libraries or kernel accesses which were not subject the the NUMA policy.

Further testing revealed that llama.cpp performed significantly worse under the different node strategy when it was run for the first time after the system started compared to subsequent runs with the same parameters. We call the first run case a ”cold start” and subsequent runs ”warm starts”. When it first begins, llama.cpp executes a dry run of the model with a blank prompt to load the model into memory before beginning its inference. This ensures that the model is resident in memory and relevant caches before inference

Memory Access Type	Overhead (%)	Samples
LFB or LFB hit	45.57	8350451
Remote RAM hit	41.65	4618820
L2 or L2 hit	9.47	2978983
L1 or L1 hit	2.45	1562044
L3 or L3 hit	0.79	444874
Local RAM or RAM hit	0.05	20165
L3 miss	0.01	2977

Table 4.4: perf mem report for llama.cpp running on the different node strategy with a cold start, sorted by the type of memory access. The overhead again represents the percent of samples in this category.

Strategy	real_time	real_stdev	sys_time	sys_stdev
No Mem-bind	292.0157	4.956	77.5864	0.284
Same nodes	285.1775	6.885	64.6070	1.025
Different nodes	445.0564	5.743	83.9309	1.200

Table 4.5: Average and standard deviation of real and system times in seconds over 10 cold-start trials of llama.cpp for each strategy. Lower is better.

begins. However, this takes much longer when it is run for the first time compared to subsequent executions. A perf report for this cold start condition is shown in Table 4.4. Here, the percentage of accesses made to local memory decrease from 8.81% to just 0.05% of all samples. At the same time, the percentage of accesses made to remote memory increases from 33.62% to 41.65%. This difference is noticeable in the runtime performance for llama.cpp with a cold start.

Table 4.5 shows the result of the same trial described in Chapter 3.3 with a cold start. This effect was achieved by restarting the system between each inference. This ensured that any migration or replication the OS conducted of remote pages is reset between each run, getting a more accurate representation of the performance penalties of remote accesses.

The first observation is the significant slowdown for the different node strategy. In the warm start trials, the slowdown in average real time for the different node strategy compared

to the same node strategy was 0.05%. This is counterintuitive compared to the STREAM benchmarking results shown in Table 4.1, which shows that the bandwidth of remote memory is approximately 52% that of local memory. In the cold start results, however, the average real time for llama.cpp running with the different node strategy is roughly 36% slower than the real time for llama.cpp running with the same node strategy. This is not quite the 2x slowdown observed in the STREAM report, but this is not surprising given that a plurality of accesses are still made to the LFB in both the warm and cold start conditions, as shown in tables 4.3 and 4.4. Still, this is a significant performance penalty that is much more in line with the hypothesis that forcing all allocations onto the remote NUMA node would result in a significant performance penalty.

The second observation is the noticeable increase in variance for all observed trials. In the warm start trials, variance was quite small. The runtime differences between strategies were comparatively small, but since the observed run times for each strategy were packed so close together, it was possible to show that the change in strategy had a statistically significant impact. With the cold start strategies, variance is much higher. This difference may be caused by the workload growing more sensitive to smaller changes in where pages are placed and the latency to access them. In the warm start strategy, something outside of the NUMA policy is causing the number of remote accesses to be reduced either via caching or migration. With a cold start, there is no opportunity for this to happen, so these small differences add up over the course of the inference and make the run time vary slightly more. Still, the difference between the same and different node strategies is much more significant in the cold start trials; even with the much higher variance, a T-test shows the result is statistically significant, with $p = 3.48 \times 10^{-21}$.

Third, the observation from the warm start strategy that forcing all allocations to happen on a local node outperforms the default Linux strategy continues to hold in the cold start

Memory Access Type	Overhead (%)	Samples
LFB or LFB hit	47.02	3319573
Local RAM hit	39.29	1678523
L2 or L2 hit	12.63	1878408
L1 or L1 hit	0.52	77612
L3 or L3 hit	0.50	44660
L3 miss	0.03	2300
Remote RAM hit	0.01	1304

Table 4.6: perf mem report for llama.cpp running on the default memory strategy with a cold start, sorted by the type of memory access. The overhead represents the percent of samples in this category

Memory Access Type	Overhead (%)	Samples
LFB or LFB hit	44.07	3319573
Local RAM hit	34.36	1678523
L2 or L2 hit	11.15	1878408
L1 or L1 hit	8.33	77612
L3 or L3 hit	1.96	44660
Remote RAM hit	0.09	2300
L3 miss	0.04	1304

Table 4.7: perf mem report for llama.cpp running on the same node memory strategy with a cold start, sorted by the type of memory access. The overhead represents to percent of samples in this category

strategy. While the difference is not as stark as that between the same and different node strategies, a T-test shows the result remains statistically significant, with $p = 0.02$. This is not as strong as what was observed in the warm start trials, but nevertheless it shows a slight decrease in performance. To identify potential causes of this observation, a standard perf analysis was conducted on the workload to identify the overhead of various functions in both user and kernel space. In this case, we observed that the down_read_trylock function within the kernel has an overhead of 0.12% when no binding is specified for memory allocations, but an overhead of $<<$ 0.1% when the same node strategy is used. Although 0.12% does not account for the entire difference in system times observed, it does show a potential cause or part of this observation.

To determine where this change in performance may be coming from, a perf mem analysis was conducted again. This time, the same-node strategy was compared to the default strategy. Table 4.7 shows the perf mem report for the default strategy, and Table 4.6 shows the report for the same node strategy. As expected, the difference between these two reports is quite small, which is unsurprising given that the performance difference for the two strategies is also quite small. However, the one item that stands out is the significant increase in the number of L1 cache hits. in the default strategy, only 0.52% of samples were accesses that were found in the L1 cache, but the same node strategy resulted in 8.83% of accesses being made to the L1 cache. This was coupled with a decrease in the number of accesses to the LFB, local RAM, and L2 cache, which indicates that accesses to slower tiers of the memory hierarchy were reduced in favor of more accesses to the fast L1 cache tier. It is unclear why altering the NUMA policy on this system would have such a profound effect on the cache efficiency of the workload.

Chapter 5

Discussion

The experimental results show two primary conclusions. First: Chapter 4.2 demonstrates the impact of a poor NUMA policy on the time-cost of an application. When more accesses are pushed to remote NUMA nodes unnecessarily, there is a negative impact on the time cost of the application. This impact is much greater for some tasks when they are run for the first time after system startup compared to subsequent executions. While much of the research into tiered memory management policies focuses on improvements in the training process, a given model is only trained once. A model is run, however, many times over its useful lifetime. Even small improvements to the amount of time needed to execute an inference could have substantial impact on the overall cost of hosting a model. Counterintuitively, we observed that there was a slight performance penalty for the default page placement strategy compared to specifying that memory allocations should be placed on the local NUMA node. One would expect that the default policy, even it is not ideal, would outperform other basic strategies. Further analysis revealed that a likely cause of this performance difference is an increase in the number of memory accesses made to the L1 cache layer, improving the average memory access time for this strategy.

Second: Chapter 4.1 shows that there is a consistent pattern in the memory accesses made by llama.cpp. Based on the underlying architecture for LLMs proposed in Vaswani et al. [49], a similar pattern should exist in other runtime environments as well. LLMs infer by running a series of matrix operations—the BLAS memory accesses seen in Chapter 4.1—moving

between different large matrices stored in memory. Figure 4.7 shows what appears to be a constant stride in accesses made. An analysis of how the access patterns change over time is shown in Figure 4.11 and Figure 4.13. These figures show a hot zone that migrates across the memory region. The heatmaps remote node access heatmaps show that many hot pages are trapped in remote NUMA nodes, even when the time-slice heatmaps show that the hot granules are predictable. Knowing ahead of time which pages were likely to be accessed provides an avenue to reduce memory access times by loading pages into fast local memory before the access is made.

5.1 Proposed Strategy

Based on the results shown in Chapter 4, it is clear that there is a consistent access pattern in GPT inference that could be exploited to reduce the number of accesses made to remote NUMA nodes. While this analysis was only applied to CPU inference, an inference conducted on a GPU should have a similar pattern in its accesses, but more granular page access tracking would not be available on those systems because GPUs do not usually provide that type of performance counter tracking. In theory, other common workloads should have access patterns of their own that can be exploited. Most of the existing literature [29, 40, 44] focuses on page placement and pre-fetching strategies based on live runtime analysis of access patterns. However, this often limits the workloads to which they can apply. A potential strategy is to apply the same analysis from Chapter 4 to a target workload. This workload should be a frequently repeated task with a large working memory that would benefit from a tiered memory system. If this analysis shows that there is an exploitable access pattern, a pre-fetching strategy can be devised and implemented ahead of time. This strategy would not rely on live analysis of hot and cold pages; rather, like some other strategies [15, 23, 29],

it would pre-calculate when to migrate pages based on the earlier analysis. Such a system would be transparent to the programmer, making it useful in situations where it is infeasible to make changes to the source code. It would not be totally transparent to the end user, but if designed well it would be simple for a user to run the analysis and a tool could automatically generate the strategy based on the analysis.

For such a strategy to be implemented for an LLM inference workload, as was explored in Chapter 4, it would require an ML platform team to run a system with heterogeneous tiered memory. The platform team would conduct the analysis described in Chapter 3.2 to identify the access patterns and remote access patterns unique to their system and environment. Once those patterns are identified, the team could install an automatic pre-fetcher in a user-space tool that would run the target workload and would be responsible for setting NUMA policies with the kernel and automatically migrating pages on behalf of the workload. This design would require administrator privileges, but should not require alteration of the kernel. Other designs for this tool could be explored, placing the tool in either kernel or user space or even having page migration be part of the workload's executable. Such systems could leverage kernel tools like DAMON [1] to provide page temperature tracking. Such a live analysis of memory access patterns could be a workaround for the lack of availability of performance counters on GPU systems noted earlier.

Chapter 6

Conclusions

With the rise of heterogeneous tiered memory systems, a need has arisen to find a way to manage the placement and migration of pages between local fast memory and remote slow memory. Several strategies have been proposed in the literature to handle this; however, few are general purpose. At the same time, the Linux kernel's default NUMA page placement policy is lacking. Past research shows there is a great deal of performance to be gained from better managing page placement and migration compared to the default Linux policy. There is a potential benefit to a page placement and pre-fetching strategy that relies on prior analysis of memory access patterns for the workload to identify exploitable patterns. Analysis showed that there is additional performance to be gained by exploiting the memory access patterns displayed in GPT inference.

6.1 Future Work

Further research is necessary to design such a strategy and determine its efficacy. There are several avenues that this research shows are in need of additional exploration. Exploring methods to track memory accesses at runtime through tools such as DAMON would be a good place to start. This would allow the proposed strategy to be applied to scenarios where there is a predictable pattern but the exact pattern is not known before execution. Mixture of Experts models [46], for example, use an inference engine to generate a response

which would likely have a pattern similar to that observed here. However, there are several different models used and the exact routing of which of those models are used is not known until execution begins. A live analysis would be able to recognize which regions of memory will receive a larger share of memory accesses early on in the execution and alter its pre-fetching strategy based on this. Such a strategy could combine the offline analysis presented here with the live analysis from DAMON by first developing a model of the access patterns for a variety of potential paths the workload can take. Live analysis from DAMON can then be used to identify which path the workload has taken at runtime, and the pre-fetching strategy can be updated based on this knowledge.

The same analysis should be conducted on a GPU inference workload as well. As noted in Chapter 3.2, it was not possible to record the memory access pattern for GPU workloads on the systems used for this thesis. However, most real-world inference workloads happen on GPUs, so further research should explore any potential differences in access patterns between GPU and CPU inference.

Further analysis can also explore developing a machine learning model to predict accesses and model both the temporal and spatial patterns observed.

The observation that the default page placement strategy performs worse than the same node strategy is also interesting and is worth further exploration with a more advanced toolset.

Bibliography

[1] DAMON: Data Access MONitor — The Linux Kernel documentation, . URL https://www.kernel.org/doc/html/v5.17/vm/damon/index.html.

[2] Graph 500, . URL http://graph500.org/.

[3] llama.cpp, . URL https://github.com/ggml-org/llama.cpp?tab=readme-ov-file.

[4] matplotlib.pyplot.imshow — Matplotlib 3.10.1 documentation, . URL https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.imshow.html.

[5] numactl(8) – Linux Manual Page, . URL https://www.man7.org/linux/man-pages/man8/numactl.8.html.

[6] perf-mem(1) – Linux manual page, . URL https://www.man7.org/linux/man-pages/man1/perf-mem.1.html.

[7] perf: Linux profiling with performance counters, . URL https://perfwiki.github.io/main/.

[8] sleep(1) – Linux Manual Page, . URL https://www.man7.org/linux/man-pages/man1/sleep.1.html.

[9] Slurm Workload Manager - Documentation, . URL https://slurm.schedmd.com/.

[10] TensorFlow, . URL https://www.tensorflow.org/.

[11] time(1) - Linux manual page, . URL https://www.man7.org/linux/man-pages/man1/time.1.html.

[12] Basic Linear Algebra Subprograms, 2025. URL https://www.netlib.org/blas/.

[13] OpenVINO, 2025. URL https://docs.openvino.ai/2025/index.html.

[14] Andrea Arcangeli. AutoNUMA, May 2012. URL https://mirrors.edge.kernel.org/pub/linux/kernel/people/andrea/autonuma/autonuma_bench-20120530.pdf.

[15] Moiz Arif, Kevin Assogba, M. Mustafa Rafique, and Sudharshan Vazhkudai. Exploiting CXL-based Memory for Distributed Deep Learning. In *Proceedings of the 51st International Conference on Parallel Processing*, pages 1–11, Bordeaux France, August 2022. ACM. ISBN 978-1-4503-9733-9. doi: 10.1145/3545008.3545054. URL https://dl.acm.org/doi/10.1145/3545008.3545054.

[16] Moiz Arif, Avinash Maurya, and M. Mustafa Rafique. Accelerating Performance of GPU-based Workloads Using CXL. In *Proceedings of the 13th Workshop on AI and Scientific Computing at Scale using Flexible Computing*, pages 27–31, Orlando FL USA, August 2023. ACM. ISBN 9798400701665. doi: 10.1145/3589013.3596678. URL https://dl.acm.org/doi/10.1145/3589013.3596678.

[17] Moiz Arif, Avinash Maurya, M. Mustafa Rafique, Dimitrios S. Nikolopoulos, and Ali R. Butt. Application-Attuned Memory Management for Containerized HPC Workflows. In *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 114–127, San Francisco, CA, USA, May 2024. IEEE. ISBN 9798350387117. doi: 10.1109/IPDPS57955.2024.00019. URL https://ieeexplore.ieee.org/document/10579205/.

[18] Trinayan Baruah, Yifan Sun, Ali Tolga Dinçer, Saiful A. Mojumder, José L. Abellán, Yash Ukidave, Ajay Joshi, Norman Rubin, John Kim, and David Kaeli. Griffin: Hardware-Software Support for Efficient Page Migration in Multi-GPU Systems.

In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 596–609, February 2020. doi: 10.1109/HPCA47549.2020.00055. URL https://ieeexplore.ieee.org/document/9065453/authors. ISSN: 2378-203X.

[19] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966. ISSN 0018-8670. doi: 10.1147/sj.52.0078. URL https://ieeexplore.ieee.org/document/5388441/?arnumber=5388441. Conference Name: IBM Systems Journal.

[20] Lars Bergstrom. Measuring NUMA effects with the STREAM benchmark, March 2011. URL http://arxiv.org/abs/1103.3225. arXiv:1103.3225 [cs].

[21] Stella Bitchebe, Djob Mvondo, Alain Tchana, Laurent Réveillère, and Noël De Palma. Intel Page Modification Logging, a hardware virtualization feature: study and improvement for virtual machine working set estimation, January 2020. URL http://arxiv.org/abs/2001.09991. arXiv:2001.09991 [cs].

[22] Scott Cyphers, Arjun K. Bansal, Anahita Bhiwandiwalla, Jayaram Bobba, Matthew Brookhart, Avijit Chakraborty, Will Constable, Christian Convey, Leona Cook, Omar Kanawi, Robert Kimball, Jason Knight, Nikolay Korovaiko, Varun Kumar, Yixing Lao, Christopher R. Lishka, Jaikrishnan Menon, Jennifer Myers, Sandeep Aswath Narayana, Adam Procter, and Tristan J. Webb. Intel nGraph: An Intermediate Representation, Compiler, and Executor for Deep Learning, January 2018. URL http://arxiv.org/abs/1801.08058. arXiv:1801.08058 [cs].

[23] Nikoli Dryden, Roman Böhringer, Tal Ben-Nun, and Torsten Hoefler. Clairvoyant Prefetching for Distributed Machine Learning I/O, 2021. URL https://arxiv.org/abs/2101.08734. Version Number: 2.

[24] João Henrique F. Flores, Paulo Martins Engel, and Rafael C. Pinto. Autocorrelation and partial autocorrelation functions to improve neural networks models on univariate time series forecasting. In *The 2012 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, June 2012. doi: 10.1109/IJCNN.2012.6252470. URL https://ieeexplore.ieee.org/abstract/document/6252470. ISSN: 2161-4407.

[25] Bin Gao, Zhuomin He, Puru Sharma, Qingxuan Kang, Djordje Jevdjic, Junbo Deng, Xingkun Yang, Zhou Yu, and Pengfei Zuo. Cost-Efficient Large Language Model Serving for Multi-turn Conversations with CachedAttention, June 2024. URL http://arxiv.org/abs/2403.19708. arXiv:2403.19708 [cs].

[26] Gemma Team. Gemma, 2024. URL https://www.kaggle.com/m/3301.

[27] Pouya Hamadanian and Sadjad Fouladi. Glinthawk: A Two-Tiered Architecture for Offline LLM Inference, February 2025. URL http://arxiv.org/abs/2501.11779. arXiv:2501.11779 [cs].

[28] Yizeng Han, Gao Huang, Shiji Song, Le Yang, Honghui Wang, and Yulin Wang. Dynamic Neural Networks: A Survey, December 2021. URL http://arxiv.org/abs/2102.04906. arXiv:2102.04906 [cs].

[29] Mark Hildebrand, Jawad Khan, Sanjeev Trika, Jason Lowe-Power, and Venkatesh Akella. AutoTM: Automatic Tensor Movement in Heterogeneous Memory Systems using Integer Linear Programming. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 875–890, Lausanne Switzerland, March 2020. ACM. ISBN 978-1-4503-7102-5. doi: 10.1145/3373376.3378465. URL https://dl.acm.org/doi/10.1145/3373376.3378465.

[30] Takahiro Hirofuchi and Ryousei Takano. The Preliminary Evaluation of a Hypervisor-based Virtualization Mechanism for Intel Optane DC Persistent Memory Module, 2019. URL https://arxiv.org/abs/1907.12014. Version Number: 1.

[31] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module, 2019. URL https://arxiv.org/abs/1903.05714. Version Number: 3.

[32] Marisa Kirisame, Steven Lyubomirsky, Altan Haan, Jennifer Brennan, Mike He, Jared Roesch, Tianqi Chen, and Zachary Tatlock. Dynamic Tensor Rematerialization, March 2021. URL http://arxiv.org/abs/2006.09616. arXiv:2006.09616 [cs].

[33] Sandeep Kumar, Aravinda Prasad, Smruti R. Sarangi, and Sreenivas Subramoney. Radiant: efficient page table management for tiered memory systems. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management*, pages 66–79, Virtual Canada, June 2021. ACM. ISBN 978-1-4503-8448-3. doi: 10.1145/3459898.3463907. URL https://dl.acm.org/doi/10.1145/3459898.3463907.

[34] Thai Le, Steven Briscoe, and Jonathan Stern. Fast memcpy with SPDK and Intel® I/OAT DMA Engine, April 2017. URL https://www.intel.com/content/www/us/en/developer/articles/technical/fast-memcpy-using-spdk-and-ioat-dma-engine.html.

[35] Baolin Li, Yankai Jiang, Vijay Gadepally, and Devesh Tiwari. LLM Inference Serving: Survey of Recent Advances and Opportunities, July 2024. URL http://arxiv.org/abs/2407.12391. arXiv:2407.12391 [cs].

[36] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia,

and Prakash Chauhan. 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*, pages 742–755, Vancouver BC Canada, March 2023. ACM. ISBN 978-1-4503-9918-0. doi: 10.1145/3582016.3582063. URL https://dl.acm.org/doi/10.1145/3582016.3582063.

[37] John D McCalpin. The STREAM Benchmark The STREAM Benchmark.

[38] OpenAI. OpenAI GPT-4.5 System CArd. 2025.

[39] Xiurui Pan, Endian Li, Qiao Li, Shengwen Liang, Yizhou Shan, Ke Zhou, Yingwei Luo, Xiaolin Wang, and Jie Zhang. InstInfer: In-Storage Attention Offloading for Cost-Effective Long-Context LLM Inference, September 2024. URL http://arxiv.org/abs/2409.04992. arXiv:2409.04992 [cs].

[40] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 392–407, Virtual Event Germany, October 2021. ACM. ISBN 978-1-4503-8709-5. doi: 10.1145/3477132.3483550. URL https://dl.acm.org/doi/10.1145/3477132.3483550.

[41] Jie Ren, Dong Xu, Shuangyan Yang, Jiacheng Zhao, Zhicheng Li, Christian Navasca, Chenxi Wang, Harry Xu, and Dong Li. Enabling Large Dynamic Neural Network Training with Learning-based Memory Management. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 788–802, Edinburgh, United Kingdom, March 2024. IEEE. ISBN 9798350393132. doi: 10.1109/HPCA57654.2024.00066. URL https://ieeexplore.ieee.org/document/10476398/.

[42] Steve Scargall. Breaking Memory Barriers: CXL's Game-Changing Impact on AI/ML. URL https://computeexpresslink.org/wp-content/uploads/2024/12/CXL-Breaking-Memory-Barriers-Webinar.pdf.

[43] Alexander Schrijver. *Theory of linear and integer programming*. Wiley-Interscience series in discrete mathematics and optimization. Wiley, Chichester Weinheim, nachdr. edition, 2011. ISBN 978-0-471-98232-6.

[44] Sai Sha, Chuandong Li, Yingwei Luo, Xiaolin Wang, and Zhenlin Wang. vTMM: Tiered Memory Management for Virtual Machines. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 283–297, Rome Italy, May 2023. ACM. ISBN 978-1-4503-9487-1. doi: 10.1145/3552326.3587449. URL https://dl.acm.org/doi/10.1145/3552326.3587449.

[45] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307, Washington DC USA, October 2004. ACM. ISBN 978-1-58113-961-7. doi: 10.1145/1030083.1030124. URL https://dl.acm.org/doi/10.1145/1030083.1030124.

[46] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer, January 2017. URL http://arxiv.org/abs/1701.06538. arXiv:1701.06538 [cs].

[47] Gurjot Singh, Prabhjot Singh, and Maninder Singh. Advanced Real-Time Fraud Detection Using RAG-Based LLMs, January 2025. URL http://arxiv.org/abs/2501.15290. arXiv:2501.15290 [cs].

[48] S. Van Doren. Abstract - HOTI 2019: Compute Express Link. In *2019 IEEE Symposium on High-Performance Interconnects (HOTI)*, pages 18–18, Santa Clara, CA, USA, August 2019. IEEE. ISBN 978-1-72815-525-8. doi: 10.1109/HOTI.2019.00017. URL https://ieeexplore.ieee.org/document/9070313/.

[49] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention Is All You Need, August 2023. URL http://arxiv.org/abs/1706.03762. arXiv:1706.03762 [cs].

[50] Jianbo Wu, Jie Liu, Gokcen Kestor, Roberto Gioiosa, Dong Li, and Andres Marquez. Performance Study of CXL Memory Topology. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '24, pages 172–177, New York, NY, USA, December 2024. Association for Computing Machinery. ISBN 9798400710919. doi: 10.1145/3695794.3695809. URL https://dl.acm.org/doi/10.1145/3695794.3695809.

[51] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Nimble Page Management for Tiered Memory Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 331–345, Providence RI USA, April 2019. ACM. ISBN 978-1-4503-6240-5. doi: 10.1145/3297858.3304024. URL https://dl.acm.org/doi/10.1145/3297858.3304024.

[52] Jifei Yi, Benchao Dong, Mingkai Dong, and Haibo Chen. On the precision of precise event based sampling. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 98–105, Tsukuba Japan, August 2020. ACM. ISBN 978-1-4503-8069-0. doi: 10.1145/3409963.3410490. URL https://dl.acm.org/doi/10.1145/3409963.3410490.

[53] Man-Ki Yoon, Sibin Mohan, Jaesik Choi, and Lui Sha. Memory heat map: anomaly

detection in real-time embedded systems using memory behavior. In *Proceedings of the 52nd Annual Design Automation Conference*, pages 1–6, San Francisco California, June 2015. ACM. ISBN 978-1-4503-3520-1. doi: 10.1145/2744769.2744869. URL https://dl.acm.org/doi/10.1145/2744769.2744869.