Exploring Fine-grained Memory Analysis for PIM Offloading

Thomas M. Papka
Loyola University Chicago
Chicago, Illinois, USA
tpapka@luc.edu

Nanda Velugoti (advisor)
Illinois Institute of Technology
Chicago, Illinois, USA
nvelugoti@hawk.iit.edu

Kyle C. Hale (advisor)
Illinois Institute of Technology
Chicago, Illinois, USA
khale1@iit.edu

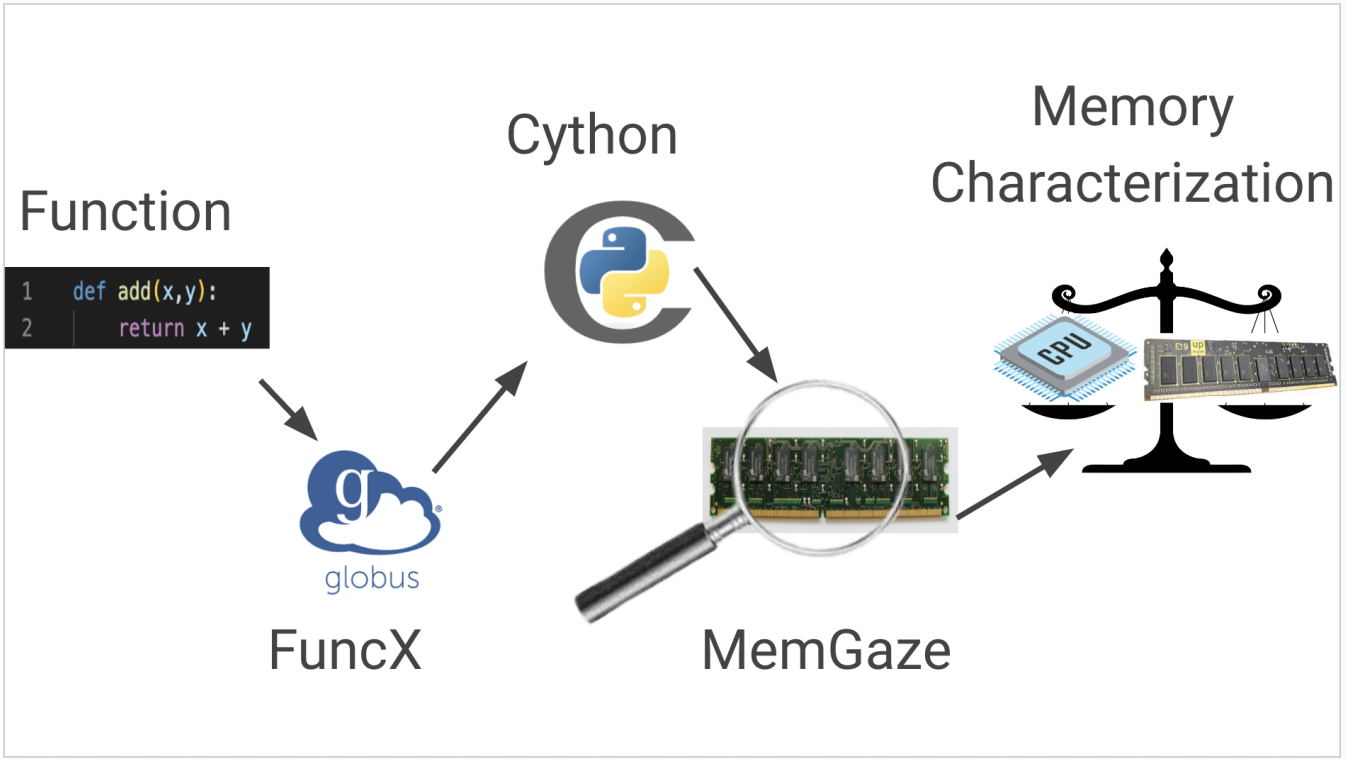


Figure 1: Memory analysis pipeline for better understanding when to offload processing from CPU to PIM device.

Abstract

In modern computing, a challenge is the data bottleneck between the CPU and RAM. This issue arises because the CPU can process data faster than it can be accessed from RAM; this is worsened by the fact that large amounts of RAM are less accessible than a powerful CPU. Furthermore, RAM's high cost creates a need for a cost-effective solution. Processing in Memory (PIM) offers a potential remedy by reducing data movement, thus alleviating bottlenecks. To optimize the use of this new hardware, developers need to identify when to offload their programs to a PIM device. To address this need, we have developed a solution that enables developers to run Python programs through our pipeline, highlighting the memory-intensive parts of their code.

CCS Concepts

• **Computer systems organization** → **Architectures**.

Keywords

processing in memory(PIM), data bottleneck, memory offloading, memory-intensive computing, memgaze

ACM Reference Format:
Thomas M. Papka, Nanda Velugoti (advisor), and Kyle C. Hale (advisor). 2024. Exploring Fine-grained Memory Analysis for PIM Offloading. In *Proceedings of ACM Student Research Competition (SC '24).* ACM, New York, NY, USA, 3 pages. https://doi.org/XXXXXXX.XXXXXXX

SC '24, November 17 – 22, 2024, Atlanta, GA

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

In computational science, a common bottleneck occurs when applications become memory-bound, limiting performance by memory

access speed rather than CPU speed. This issue arises when the CPU waits for data transfer, leading to idle CPU cycles. Programs that frequently access large memory-stored data, such as scientific simulations or data analysis, are particularly prone to being memory-bound.

Memory-bound systems are characterized by high memory access latency, limited cache utilization, and bandwidth constraints. Effective CPU cache use can alleviate memory bottlenecks, but cache misses occur when a program's working set exceeds the cache size or has an irregular access pattern, leading to additional trips to slower main memory. Performance is constrained if the memory subsystem cannot supply data to the CPU as fast as it needs.

2 Background

Processing in Memory (PIM) has emerged as a promising solution to the limitations of memory-bound systems. Traditional architectures struggle with data transfer inefficiencies between the CPU and memory, leading to bottlenecks that degrade performance. PIM enables computation within the memory, reducing the time and energy spent on data movement.

Mutlu et al. [3] provide an overview of PIM technology, examining theoretical and practical aspects. PIM minimizes the dependency on CPU-memory data transfer by processing data within the memory itself, alleviating the bottleneck effect in high-performance applications. Chen et al. [1] introduces a software framework to integrate PIM capabilities into existing systems, enabling efficient data processing without substantial codebase modifications. While increasing RAM capacity could help alleviate the bottleneck, it's often not practical or cost-effective for researchers.

3 Approach

We use ***MemGaze***, a tool designed to gather memory usage traces with minimal overhead [2]. MemGaze leverages modern processors' capabilities to record memory access sequences with precision. Our process is as follows:

(1) Application (function) is written in Python
(2) We implement the function in Globus Compute FaaS (under development)
(3) Using Cython, we convert the function to C for use with MemGaze
(4) We execute the MemGaze-enabled function to collect trace files
(5) Analyze the trace to generate cluster analysis graphs

We augment MemGaze analysis with tools to convert trace files into clustered graphs (see Figure 2 - 3) that show memory usage patterns, helping developers identify memory sections of their Python function for potential offloading to PIM. This comprehensive approach enables developers to gain insights into memory behavior and optimize applications for better performance and resource utilization.

After collecting trace data, we perform cluster analysis to visualize memory access patterns. The process begins by analyzing trace text and JSON files, detailing memory access points of our functions. Each vertex in the resulting graph represents a memory access from the executed function, grouped into communities based on access patterns. Darker shades represent communities with more frequent interactions, providing a clear overview of how different parts of the function interact within the memory.

4 Results

To generate meaningful graphs, the trace text files must be filtered to isolate the original Python code and exclude additional Cython-generated code. A custom Python script lets the user focus on the original Python functions. This filtering is crucial for producing manageable results, enabling programmers to concentrate on the functions of interest.

The resulting graphs illustrate a vector addition and a breadth-first search (BFS) example, showcasing the contrast between filtered and unfiltered cluster analysis (see Figures 2 and 3). The unfiltered graphs include excessive details about Cython functions, while the filtered graphs provide a clearer view of memory access patterns in the original functions. By cross-referencing with the JSON file, the most memory-intensive functions can be identified, streamlining the optimization process.

5 Conclusion and Future Work

This project successfully streamlines the process of identifying memory usage in applications, aiding in determining whether portions should be offloaded to a PIM device. This advancement provides programmers with tools to make informed decisions about PIM utilization. Future work could focus on fully automating the process and enhancing graph analysis capabilities, improving the system's utility. Integration into a Function-as-a-Service (FaaS) platform like Globus Compute could also be explored, ensuring any programmer can analyze Python programs' memory usage efficiently.

6 Acknowledgments

This work was funded by the Research Experiences for Undergraduates (REU) program, with support from the National Science Foundation grant OAC-2150500. I want to thank my REU advisors, Nanda Velugoti and Kyle Hale, for their invaluable guidance and support throughout this project. I also wish to thank my fellow REU students for collaborating during this summer's research experience.

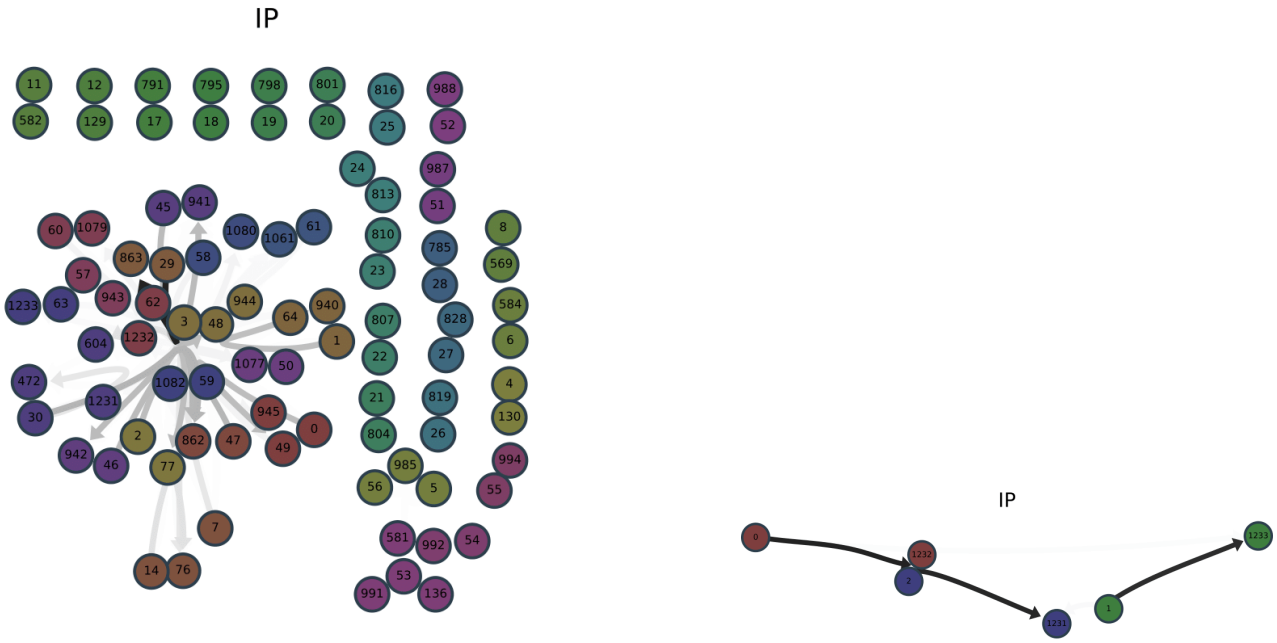
References

[1] Jinfan Chen, Juan Gómez-Luna, Izzat El Hajj, Yuxin Guo, and Onur Mutlu. 2023. Simplepim: A Software Framework for Productive and Efficient Processing-in-Memory. In *2023 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 99–111.

[2] Ozgur O Kilic, Nathan R Tallent, Yasodha Suriyakumar, Chenhao Xie, Andrés Marquez, and Stephane Eranian. 2022. MemGaze: Rapid and Effective Load-Level Memory Trace Analysis. In *2022 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 484–495.

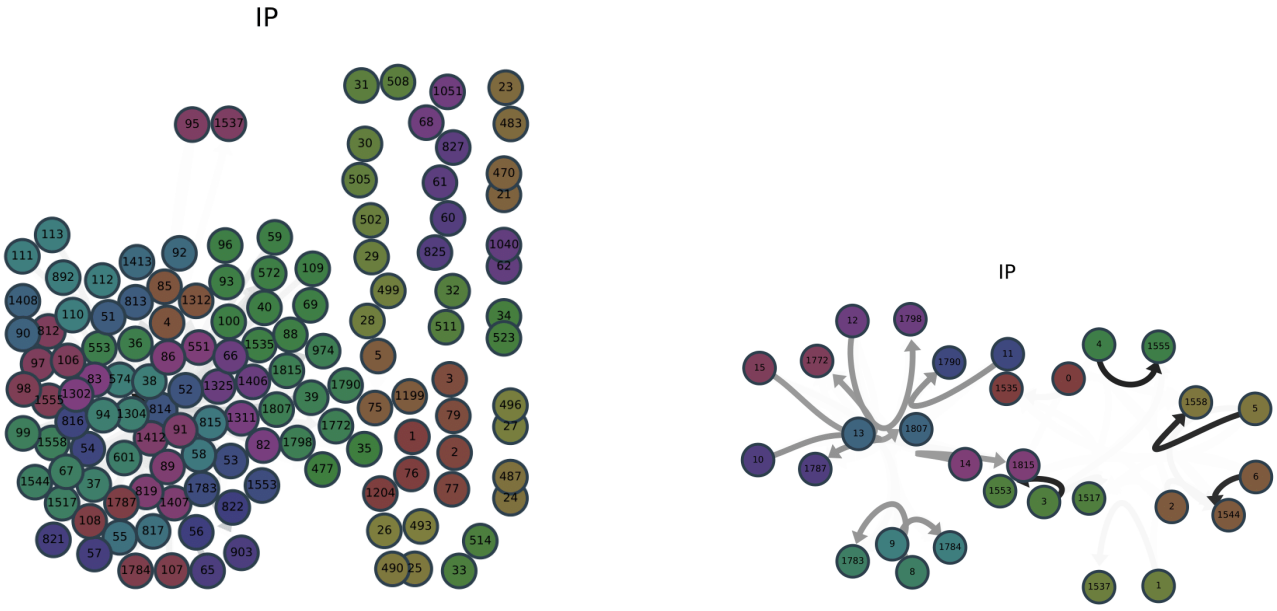
[3] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. 2022. A Modern Primer on Processing in Memory. In *Emerging computing: from devices to systems: looking beyond Moore and Von Neumann*. Springer, 171–243.

Received 09 August 2024



(a) Unfiltered Result (b) Filtered Result

Figure 2: Results for *Vector Addition* function, where the unfiltered has all data collected by MemGaze, and filtered results contain only data of interest to the vector addition function.



(a) Unfiltered Result (b) Filtered Result

Figure 3: Results for *Breadth First Search* function, where the unfiltered has all data collected by MemGaze, and filtered results contain only data of interest to the vector addition function.