Visualization of Data Movements and Accesses

Seminar Thesis

Til Mohr

Chair for High Performance Computing, IT Center,
RWTH Aachen, Seffenter Weg 23,
52074 Aachen, Germany
Supervisor: Isa Thärigen, M.Sc.

This is the abstract. It is a short summary of the thesis contents (100 to 150 words).

Keywords: data locality, data movement, data access, optimization, visualization

1 Introduction

- Increasing Processor-Memory Speed Gap (2.1) and increasing requirements (2.2) result in a large increase in the affect of data movement costs and their resulting bottlenecks. While breakthroughs in hardware research can improve this issue, software engineers can also try to mitigate these issues by improving data locality (2.4) through software.

- Increasing complexity of programs makes it difficult to create a mental model of the data movement of a program. This makes it challenging for experts and impossible for domain researchers to optimize such a program.

2 Memory-Related Performance Problems

As modern computing systems evolve, the demand for increased computational power and memory resources has become more prevalent. This demand is driven by the increasing complexity of applications and the need to process larger amounts of data. In this section, we will explore the challenges and performance problems that result from the ever-growing requirements for memory and computational resources. We begin by discussing the processor-memory performance gap and its implications in Section 2.1, followed by a brief examination of the increasing computational and memory requirements of modern applications (Section 2.2). The processor-memory performance gap and the increasing computational and memory requirements combined result in a need to tackle high data transfer costs and bottlenecks (Section 2.3). Finally, we will define the concept of data locality in Section 2.4, which solutions consider to the aforementioned problems.

2.1 Processor-Memory Performance Gap

It is well known, that the performance of CPUs doubles roughly every two years, a phenomenon resulting from Moore's law. Similarly, memory technology has also been progressing exponentially, however, at a slower pace [1–4]. Since the difference between two exponential functions is also exponential, this gap will expand rapidly. This concept is known as the processor-memory performance gap. Figure 1 illustrates this trend in improvements in computational and memory performance, measured by floating point operations and memory operations per second, respectively.

The increasing processor-memory performance gap becomes a critical problem when considering data access times. Take the equation for the average memory access time:

$$t_{avg} = p \cdot t_c + (1 - p) \cdot t_m \tag{1}$$

Here, $p \in [0, 1)$ denotes the probability of a cache hit (Section 2.3). As at least one instruction has to be fetched from memory, at least one cache miss per application is guaranteed, thus $p < 1$. t_c and t_m denote the times to access data from a cache and the main memory, respectively [5, 6]. These times

[1] Data was acquired from a collection of STREAM benchmark results on `https://www.cs.virginia.edu/stream/`.

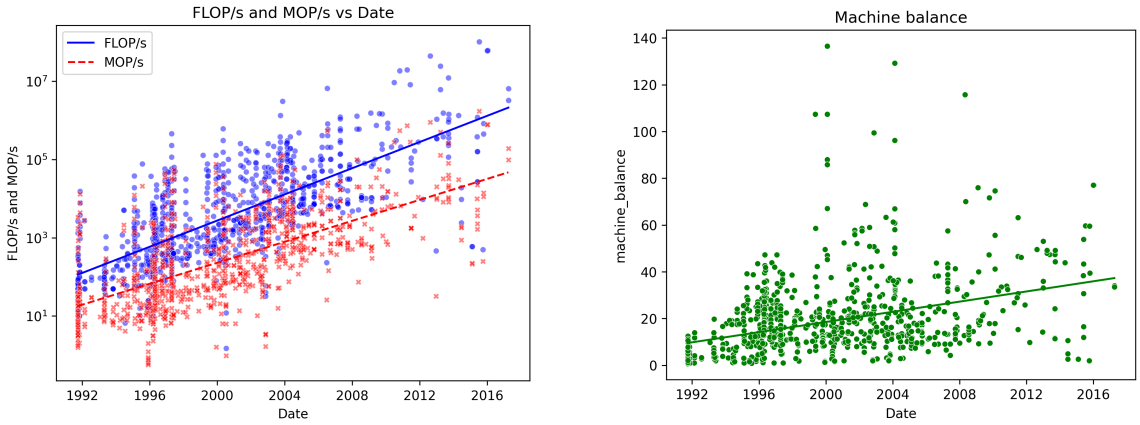

Figure 1: Illustration of the expanding Processor-Memory Gap. The left graph charts the progression of FLOPs and MOPs on a logarithmic scale across various computing platforms, with the FLOPs trendline demonstrating a steeper ascent, indicative of the widening gap. The right figure depicts the development of the machine balance score for these platforms.[1]

measure the performance of the cache and the main memory as a combination of memory latency and bandwidth (Section 2.3). Without loss of generality, consider these times as the corresponding number of clock cycles.

As a result of the increasing processor-memory performance gap, t_m (and to a lesser extent t_c) will be increasing exponentially, taking more and more clock cycles to access the same amount of data - clock cycles that could be used to perform calculations. As a result, the overall system performance will be increasingly determined by memory performance. At some point, CPUs would be able to execute code faster than we can feed them with instructions and data. For this reason, the processor-memory performance gap is also known as the memory wall problem [3, 5, 6].

To quantify the processor-memory performance gap, the notion of machine balance has been introduced [4, 7]:

$$balance = \frac{peak\ FLOP/s}{sustained\ MOP/s} \tag{2}$$

This metric, also depicted in Figure 1, is a measure of how well a system is balanced between computational and memory performance. A balance of 1 indicates a perfectly balanced system, whereas $balance \ll 1$ or $balance \gg 1$ indicates a system that is entirely compute or memory-bound, respectively [4, 7].

2.2 Computation and Memory Requirements

Different applications have different requirements for system resources. There exist some programs that have a larger computational demand, thus benefiting from a higher machine balance (Section 2.1) [7]. However, the memory wall problem states that it will be increasingly more difficult for such applications to exploit further advances in computational performance, as for any application the memory performance will grow to be the limiting factor [3, 5].

Furthermore, we notice that computational as well as memory requirements are increasing rapidly. A prime example of this is the field of artificial intelligence systems, which currently sees exponential growth in the number of parameters used [8]. Hence, for any application, regardless of its computational or memory demands, significant strides must be made in enhancing both the processing and memory capabilities. This ensures that the constraints imposed by the memory wall problem do not inhibit the potential performance of these applications.

2.3 Data Transfer Costs and Bottlenecks

Memory Latency Memory latency pertains to the time delay between a request to access data from the main memory and the start of the execution of this operation. Increased memory latencies, measured in clock cycles, lead to the processor waiting longer for data, significantly tightening the performance bottleneck. This latency challenge can adversely impact the execution of applications, and its reduction is often a complex task [2].

Memory Bandwidth Memory bandwidth denotes the volume for data transfer to or from memory

per unit of time. A bottleneck arises when the bandwidth is insufficient to handle the required data transfer volume, causing the processor to wait for data [2].

Cache Misses To alleviate the impact of memory latencies, a multi-tiered memory hierarchy has been implemented in modern computing systems. This hierarchy includes the use of caches, which are smaller, faster, and more expensive memory modules placed between the CPU and main memory. When the processor needs to access data, it first checks if the data is already in the cache, a situation known as a cache hit. However, if the data is not in the cache, the processor has to retrieve it from the slower main memory, a process known as a cache miss [1, 2, 6]. When a cache miss occurs, an entire block of memory known as a cache line is loaded into the cache. The cache line includes the requested data and some adjacent memory locations. However, this process of retrieving data from the main memory takes considerably more time due to the higher latency of the main memory. Therefore, cache management, handled by software, is vital to maintain optimal performance. Improper management can lead to an increase in cache misses, thereby significantly degrading the system's performance.

2.4 Data Locality

Data locality is a key concept in enhancing memory performance and therefore reducing the implications of the processor-memory performance gap. It refers to the tendency of a processor to access the same set of memory locations, or closely stored memory locations, repetitively over a short period. This concept capitalizes on the multi-level memory hierarchy of modern computers: By improving data locality, one reduces the number of cache misses (Section 2.3), i.e., decreases p in Equation 1, improving overall system performance [9, 10].

There are two main types of data locality: temporal and spatial locality. Temporal locality involves reusing the same data within a relatively small duration. This means that if a memory location is accessed, it is probable that the same location will be accessed again soon. Spatial locality, on the other hand, refers to the use of data items stored in proximity. In other words, if a memory location is accessed, memory locations nearby will likely be accessed shortly [11].

Data layout plays a significant role in the realization of data locality, as it can substantially influence the memory access patterns and hence, the underlying performance of a program. A thoughtful arrangement of data in memory can encourage both temporal and spatial locality, thus reducing cache misses and enhancing the overall system performance [11].

To illustrate, consider a two-dimensional array laid out in memory, where elements in the same row are stored in consecutive memory locations. If an application iterates through this array row by row, it benefits from spatial locality, as loading one element of the matrix also loads the few next elements in the row into the cache due to loading of entire cache lines (Section 2.3), thus reducing the number of cache misses. On the contrary, if the application were to traverse the array column by column, it would not benefit from spatial locality due to the dispersed memory locations of elements in the same column, leading to a higher rate of cache misses and reduced performance.

While this example demonstrates a simple scenario, the reality is often more complex, especially for larger and more intricate applications. Understanding the data access patterns of an application is key to deciding the best data layout, and this often requires an intimate knowledge of the program's structure. Moreover, optimizing data locality can be quite challenging due to the diversity of hardware architectures. The same program can exhibit different data locality characteristics on different hardware due to variations in the memory hierarchy (such as cache sizes and levels, memory bandwidth, and latency).

This paper will provide an overview of an approach to optimize data locality with the help of visualizations.

3 Data Gathering and Visualization Approaches

The goal is to make it easily accessible and possible for everyone (experts and domain researchers) to understand a programs' data movements and fix issues. To do this, data has to be gathered automatically (3.1,3.2,3.3) and then visualized in an understandable manner (3.4).

In this section, we will discuss the different approaches to gathering data for visualizing data movements in a program.

3.1 Dynamic Analysis

Run program and gather data while it is running, using Hardware Counters, Profiling, or Tracing. [12–16]

Advantages:

- No need for parameterization already is compiled for specific hardware
- Can be used in combination with actual data even more accurate information

Disadvantages:

- Running a whole program is expensive (time and cost)
- Difficult to isolate and analyze specific parts of the program
- Very coarse time granularity can not measure very short time intervals (hardware counters are not precise enough in aspects of being updated / read)

3.2 Static Analysis

Analyze the program statically using a compiler [17–23]

Advantages:

- Can be used to analyze specific parts of the program
- Fast and cheap No need to run program

Disadvantages:

- Needs to be parameterized for specific hardware (and often not accurate enough might miss some details)
- Can not be used in combination with actual data

3.3 Simulation

Simulate the program on a simulator [17, 24–26]

Advantages: **TODO**

- Can be used to analyze specific parts of the program
- In between Static and Dynamic Analysis in terms of precision and speed and cost

Disadvantages: **TODO**

3.4 Visualization Techniques

Very brief overview of different visualizations (Colored Graphs, Heatmaps, etc.)

4 Memory Access Visualization Tools

Take ~3 papers, briefly present how they work (which data gathering technique and what visualization), and what results they have shown.

4.1 MemAxes: Visualization and Analytics for Characterizing Complex Memory Performance Behaviors

[12]

4.2 Abstract Visualization of Runtime Memory Behavior

[25]

4.3 Boosting Performance Optimization with Interactive Data Movement Visualization

[17]

4.4 Comparison

5 Conclusions

Conclusion

Future Work:

- Data Gathering and Visualizations can always be improved
- But we can use the gathered data to automatically optimize programs [22] Even less work for the programmer (who might be just a domain researcher)
- Deep Learning is also being experimented with to automatically optimize programs at compile time [27]

References

[1] Danijela Efnusheva, Ana Cholakoska, and Aristotel Tentov. "A survey of different approaches for overcoming the processor-memory bottleneck". In: *International Journal of Computer Science and Information Technology* 9.2 (2017), pp. 151–163.

[2] Philip Machanick. "Approaches to addressing the memory wall". In: *School of IT and Electrical Engineering, University of Queensland* (2002).

[3] Sally A McKee. "Reflections on the memory wall". In: *Proceedings of the 1st conference on Computing frontiers*. 2004, p. 162.

[4] John D McCalpin. "A survey of memory bandwidth and machine balance in current high performance computers". In: *Newsletter of the IEEE Technical Committee on Computer Architecture (TCCA)(December 1995)* (1997).

[5] Wm A Wulf and Sally A McKee. "Hitting the memory wall: Implications of the obvious". In: *ACM SIGARCH computer architecture news* 23.1 (1995), pp. 20–24.

[6] Nihar R Mahapatra and Balakrishna V Venkatrao. "The processor-memory bottleneck: problems and solutions." In: *XRDS* 5.3es (1999), 2–es.

[7] John David McCalpin. "Memory bandwidth and system balance in hpc systems". In: *UT Faculty/Researcher Works* (2016).

[8] Our World in Data. *Artificial Intelligence Parameter Count*. `https://ourworldindata.org/grapher/artificial-intelligence-parameter-count`. [Online; accessed May 12th, 2023].

[9] Didem Unat et al. "Trends in data locality abstractions for HPC systems". In: *IEEE Transactions on Parallel and Distributed Systems* 28.10 (2017), pp. 3007–3020.

[10] Adrian Tate et al. *Programming abstractions for data locality*. Tech. rep. Office of Scientific and Technical Information (OSTI), 2014.

[11] Karim Esseghir. *Improving data locality for caches*. Rice University, 1993.

[12] Alfredo Giménez et al. "Memaxes: Visualization and analytics for characterizing complex memory performance behaviors". In: *IEEE transactions on visualization and computer graphics* 24.7 (2017), pp. 2180–2193.

[13] Sergio Moreta and Alexandru Telea. "Visualizing dynamic memory allocations". In: *2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*. IEEE. 2007, pp. 31–38.

[14] Abhinav Bhatele et al. "Novel views of performance data to analyze large-scale adaptive applications". In: *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE. 2012, pp. 1–11.

[15] Kathryn S McKinley and Olivier Temam. "Quantifying loop nest locality using SPEC'95 and the perfect benchmarks". In: *ACM Transactions on Computer Systems (TOCS)* 17.4 (1999), pp. 288–336.

[16] Laksono Adhianto et al. "HPCToolkit: Tools for performance analysis of optimized parallel programs". In: *Concurrency and Computation: Practice and Experience* 22.6 (2010), pp. 685–701.

[17] Philipp Schaad, Tal Ben-Nun, and Torsten Hoefler. "Boosting performance optimization with interactive data movement visualization". In: *arXiv preprint arXiv:2207.07433* (2022).

[18] Philipp Schaad. "Boosting Performance Engineering with Visual Interactive Optimization and Analysis". MA thesis. ETH Zurich, 2021.

[19] Tal Ben-Nun et al. "Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2019, pp. 1–14.

[20] Stan Matwin and Tomasz Pietrzykowski. "Prograph: a preliminary report". In: *Computer Languages* 10.2 (1985), pp. 91–126.

[21] Jeffrey Kodosky. "LabVIEW". In: *Proceedings of the ACM on Programming Languages* 4.HOPL (2020), pp. 1–54.

[22] Alexandru Calotoiu et al. "Lifting C semantics for dataflow optimization". In: *Proceedings of the 36th ACM International Conference on Supercomputing*. 2022, pp. 1–13.

[23] Tal Ben-Nun et al. "Bridging Control-Centric and Data-Centric Optimization". In: *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization*. 2023, pp. 173–185.

[24] Julian Hammer et al. "Kerncraft: A tool for analytic performance modeling of loop kernels". In: *Tools for High Performance Computing 2016: Proceedings of the 10th International Workshop on Parallel Tools for High Performance Computing, October 2016, Stuttgart, Germany*. Springer. 2017, pp. 1–22.

[25] ANM Imroz Choudhury and Paul Rosen. "Abstract visualization of runtime memory behavior". In: *2011 6th International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*. IEEE. 2011, pp. 1–8.

[26] Roman Iakymchuk and Paolo Bientinesi. "Modeling performance through memory-stalls". In: *ACM SIGMETRICS Performance Evaluation Review* 40.2 (2012), pp. 86–91.

[27] Chris Cummins et al. "Programl: A graph-based program representation for data flow analysis and compiler optimizations". In: *International Conference on Machine Learning*. PMLR. 2021, pp. 2244–2253.