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/`.

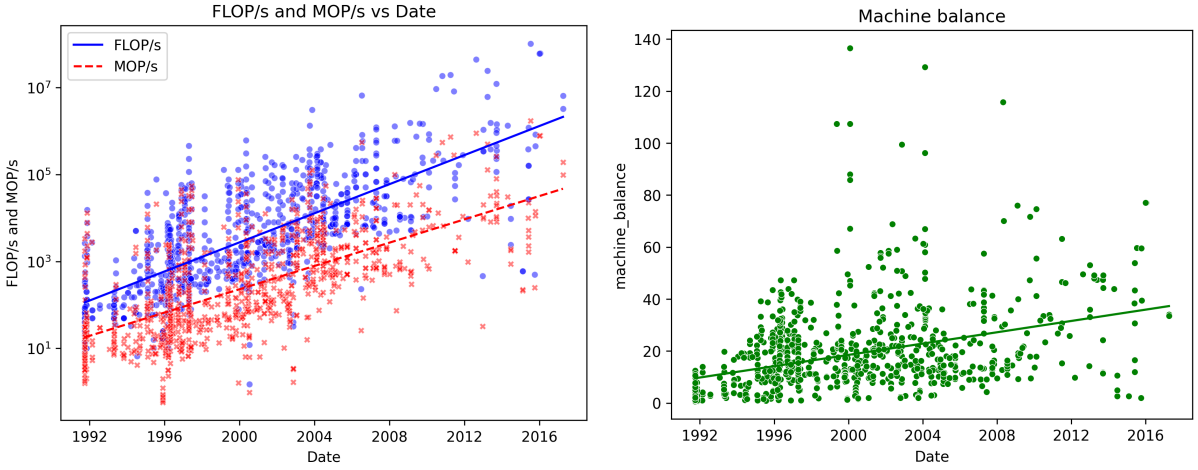

Figure 1: Illustration of the expanding Processor-Memory Gap. The left graph charts the progression of FLOPs and MOPs on a logarithmic scale across various computing platforms, with the FLOPs trendline demonstrating a steeper ascent, indicative of the widening gap. The right figure depicts the development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

In the pursuit of optimizing a program's data locality, implementing visual aids to represent data movements and data layouts can be particularly helpful. This approach enables quick and effective identification of data-related issues, their comprehension, and ultimately, their resolution. This method empowers not only program optimization experts but also domain researchers to effortlessly optimize their programs.

To enable such effective visualization, however, it's essential to first collect information regarding data locality. Several studies have explored this area, leading to the identification of three primary strategies: Dynamic Analysis, Static Analysis, and Simulation. These strategies, which we'll delve into in Sections 3.1, 3.2, and 3.3, each bring their unique benefits and drawbacks. Furthermore, it's important to note that some techniques used for gathering data locality information may not be confined to just one of these three fundamental categories, and could instead exhibit characteristics of multiple approaches.

Once this data locality information is gathered, it needs to be presented in a user-friendly manner. There exists a wide variety of visualization techniques that can fulfill this requirement, some of which we will detail in Section 3.4.

3.1 Dynamic Analysis

Dynamic analysis of a program's data locality involves executing the program while concurrently gathering memory-related data and statistics. A straightforward approach to dynamic analysis is to run the program, extract hardware counters such as cache misses, and analyze them. This approach, however, does not provide a holistic view of the program's behavior because it lacks contextual information.

To gain a deeper understanding of program performance beyond general statistics, dynamic analysis uses more nuanced techniques like profiling, statistical sampling, and tracing [12–16].

Profiling involves periodically interrupting the program's execution to capture both hardware-derived attributes and context-related information [13, 14, 16]. Profiling techniques analyze the program's call stack and program counter to provide specific details such as the current line of code being executed, the symbol, and, for arrays, the accessed index. This information facilitates the derivation of deeper metrics, such as the number of cache misses per array [16]. Profiling typically focuses on memory-related events, but the constant interruption can increase runtime overhead. Hence, a trade-off between the granularity and the quality of the measurements is necessary.

Statistical sampling is an alternative approach related to profiling. It captures the program's state at fixed time intervals rather than event-triggered interruptions. The advantage of statistical sampling is that it avoids frequent interruption of the program's execution, reducing the runtime overhead. However, high-quality measurements require sufficiently high sampling rates to capture all relevant details [16].

```
void square(double &v) {
     v = v * v;
}

// ...

square(A[i]);
```

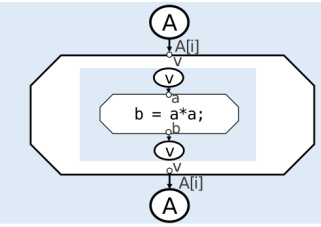

Figure 2: C language source code and its corresponding SDFG representation [17].

Tracing, another technique, allows for a temporal understanding of a program's behavior by logging event-specific data over time. Tracing functions by documenting specific events or functions during program execution, providing a chronological account of these events and their corresponding data [12, 15, 16].

In conclusion, dynamic analysis offers several distinct advantages in the study of a program's behavior concerning data locality. As the program is being executed, it offers more precise practical insights into hardware oriented data locality optimization. Further, dynamic analysis can be employed in conjunction with actual data, making it more representative of real-world scenarios.

However, it's important to note the inherent disadvantages of dynamic analysis. The act of running an entire program can be time-consuming and costly, particularly for larger and more complex software. Additionally, isolating and scrutinizing specific parts of a program under the dynamic analysis approach can be complicated, especially when compared to static analysis methods (Section 3.2).

3.2 Static Analysis

Unlike dynamic analysis, static analysis takes a different tack in examining a program's data locality. Instead of operating the program in real-time to gather data, static analysis scrutinizes the program's source code itself. By transforming the source code into an intermediate representation (IR) that centers on data, and subsequently analyzing this IR, static analysis is able to uncover memory-related issues [17–20].

There are myriad IRs in use, like MLIR, which are predominantly control-flow oriented, facilitating optimizations pivoting around control elements like loop restructuring [21]. However, in the context of data locality, data-centric IRs such as SDFG ([22]), PROGRAPH ([23]), and LabVIEW ([24]) provide a more direct approach. By prioritizing memory, its movements, and its computation-induced alter-

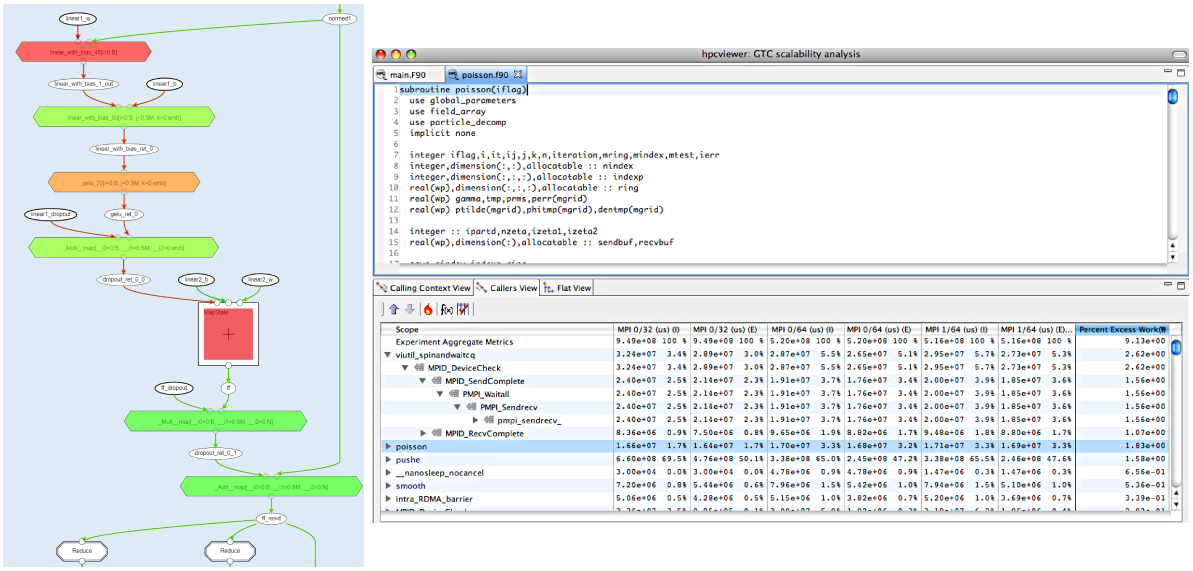

Figure 3: Left: Coloring in memory volume onto memlets and arithmetic intensity onto tasklets in an SDFG [18]. Right: `HPCToolkit`'s viewer, which displays the program's source code and its corresponding memory access information [16].

ations, these IRs allow for both automated ([22]) and, when paired with visual aids, manual enhancements of data locality [18, 20, 22].

Taking the example of SDFGs, the entire data flow of a program can be represented as a directed graph. Nodes within this graph symbolize N-dimensional arrays of data, computations (tasklets), or map scopes that denote general parallelism (such as loops). The edges, or memlets, in an SDFG represent explicit data movements [22]. An example of the SDFG IR is provided in Figure 2.

As the SDFG IR of a program is constructed, it is possible to compute memory-related properties crucial for data locality. For instance, each memlet carries information regarding the volume of data transported between nodes [22], and tasklets and nested SDFGs can be annotated with metadata related to the number of executions and arithmetic operations undertaken [18]. Consequently, SDFGs offer a comprehensive view of the program and facilitate the identification of data movement bottlenecks on a large scale.

Despite static analysis's robust capability for macroscopic program analysis - a trait not shared with dynamic analysis - it does not provide the same level of detail. Given that performance bottlenecks are often induced by memory accesses that are tied to physical access patterns and hence are hardware-specific, static analysis alone may not accurately predict, say, the number of cache misses for a particular function. However, the advantage of static analysis lies in the fact that it does not necessitate program execution, thereby enabling quicker and cost-effective optimization of logical data movements compared to dynamic analysis.

3.3 Cache Simulation

Positioned between dynamic and static analysis lies the realm of simulation-based approaches, of which cache simulation is particularly noteworthy. Cache simulation is a method used to simulate a program's data accesses on a virtual memory hierarchy model. This process allows for an in-depth examination of both spatial and temporal data locality, as elaborated on in Section 2.4.

The process of setting up a cache simulator can be divided into two ways:

In the first approach, the program is pseudo-executed without implementing any real computations. This process starts by constructing a virtual memory hierarchy that includes caches. As the program proceeds through its lifecycle:

- Corresponding space is allocated in the simulated memory for each instance of allocation, and reciprocally, space is deallocated as per the program's instructions.

- The simulator emulates each memory access operation, both read and write, according to how a CPU would handle the task. This entails an initial probe in the L1 cache, followed by a potential cache miss procedure if the required data is absent, as discussed in Section 2.3.

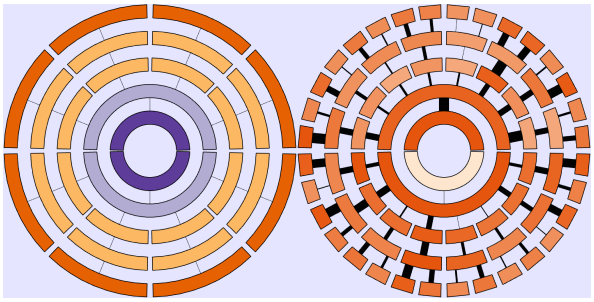

Figure 4: Left: A radial design representing the hardware topology. Main memory resources are depicted in deep purple and processing units in dark orange, with different layers of caches in between. Right: An complex architecture is portrayed, featuring performance data tagged onto the hardware resources through a color code, and transactions among them indicated by line thickness [14].

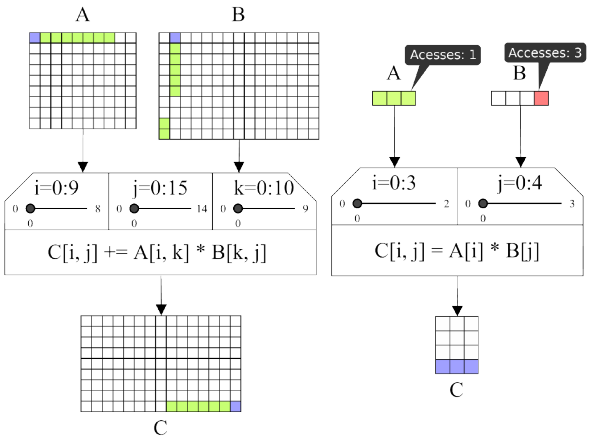

Figure 5: Left: Illustration of data layouts emphasizing spatial locality via cache lines. Right: Visualization of correlated accesses to elements A and B with respect to accesses to $C[3, x], x \in \{0, 1, 2\}$ [19].

This methodology facilitates a comprehensive and accurate representation of the system's memory hierarchy and its interaction with the computing process [19, 25].

The second approach uses dynamic analysis (Section 3.1) to generate memory traces. These traces are then rerun through the simulator, enabling an enriched understanding of memory access and management behavior within the program [26].

The deployment of cache simulators extends beyond mere prediction of cache misses. When operated in a step-by-step manner, these tools permit the exploration of data access patterns of a procedure at a granular level. Such detailed inspection can uncover potential enhancements in spatial locality, either through modifications in data layout or access strategies, ultimately contributing to improved performance [19, 25, 26].

Moreover, cache simulation enables close-up performance analysis of a program, such as focusing on a single function within the source code or limiting memory traces to a specific functional context.

Despite its advantages, cache simulation demands an in-depth understanding of the target architecture, including aspects like cache hierarchy, cache replacement policy, and cache coherence protocol. Any inaccuracies in these parameters can lead to misleading results, potentially causing optimization attempts to inadvertently degrade the program's data locality.

3.4 Visualization Techniques

Visualization is an essential aspect of data locality analysis, providing the vital link between the analysis results and user comprehension. Effective visualization techniques should balance intuitiveness and informational value. Generally, visualizations for memory-related data can be classified into three categories, each catering to a specific level of detail:

3.4.1 High-Level View

This category of visualization provides the most abstract or "bird's eye" view of data locality, aiming to deliver a global understanding of the program's performance. It emphasizes the logical data movement behavior, spotlighting the performance impact of individual parts of the program [14, 16, 18, 19], as illustrated in Figure 3. An integral feature of such visualizations is hierarchical clustering, which allows users to zoom into specific parts of the program, such as a loop nest. Despite providing a broad performance landscape, these visualizations do not shed light on the root causes of identified bottlenecks, warranting a more detailed examination.

3.4.2 Intermediate-Level View

At the intermediate level, visualizations offer more detailed insights than high-level overviews, targeting specific segments of the program like functions or loops. One common technique involves displaying the hardware topology to visualize physical data movements across different levels of the memory hierarchy, as shown in Figure 4. This more granular perspective assists in understanding performance

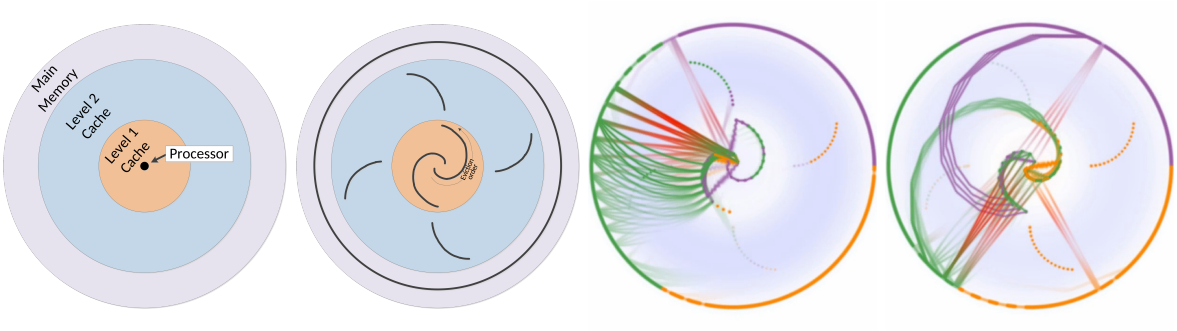

Figure 6: Left: Radial design used in [26]. Glyphs arrange themselves into groupings indicating storage on the same cache, with data closer to the boundaries between the levels more likely to be evicted. Right: A comparison of a standard 16×16 matrix multiplication and an optimized version using 4×4 blocking. The standard version shows poor data reuse for two of the three matrices [26].

bottlenecks and can help identify the most promising optimization opportunities [14, 26]. Yet, it does not provide information about underlying problematic data layout or access patterns, necessitating a deeper, fine-grained examination.

3.4.3 Detailed View

The detailed or fine-grained visualizations delve into specific aspects like data layout and access patterns within specific program segments, for instance, a loop nest. Figure 5 displays two examples of such visualizations. The left image illustrates the data layout of a matrix, highlighting the spatial locality of elements within a cache line. The right image presents the access patterns of a loop nest, showing the correlation between accesses to different arrays. This level of visualization aids in identifying potential optimization routes, such as reshaping data to improve spatial locality or reordering the loop to enhance data access patterns [19]. However, these visualizations provide insights into only one program segment at a time and lack a broader picture of the program's overall data locality. Therefore, it's advisable to use a mix of visualization techniques to obtain a comprehensive understanding of a program's data locality.

4 Exemplary Memory Access Visualization Tools

This section examines several prominent works dedicated to visualizing memory movements and accesses. Each tool will be discussed in terms of its data gathering methods, visualization techniques, and demonstrated results. We will then contrast these tools, highlighting their strengths and weaknesses.

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

The tool `MemAxes`, developed by Giménez et al. [14], utilizes dynamic analysis (Section 3.1) to generate an event log of memory accesses. Each logged event incorporates contextual information, facilitating a link back to the source code and recording the memory hierarchy depth at which the memory access occurred. This feature allows for the identification of problematic code lines, similar to `HPCToolkit` [16], as illustrated in Figure 3. Recording the resolution depth of memory access enables the determination of resource utilization across each memory module and the quantification of physical data movements between them. This information is then visualized using a radial design of the hardware topology, as seen in Figure 4. `MemAxes` also supports the display of additional attributes such as access times, latencies, and memory addresses through histograms.

In practical applications, `MemAxes` has been employed successfully to detect and mitigate performance bottlenecks, even without prior knowledge of the application's source code [14]. Performance engineers can, for instance, identify large load imbalances or significant spikes in access times, and use these insights to hypothesize the cause of a performance bottleneck. This hypothesis can then be explored further through the backlink to the source code. This approach demonstrates that low-level visual aids are not necessary for optimizing data locality in an unfamiliar program.

4.2 Abstract Visualization of Runtime Memory Behavior

Choudhury et al. [26], similar to `MemAxes`, employs dynamic analysis to record an event log of memory accesses during runtime (Section 3.1). This log is subsequently input into a cache simulator (Section 3.3) to create the radial visualizations represented in Figure 6. The authors generate numerous such visualizations throughout the program's execution, effectively creating an animation showcasing the evolution of data movements within the memory hierarchy. This dynamic approach, they argue, is more intuitive than static visualizations, such as those of `MemAxes`. While this tool does indeed enhance the understanding of large-scale memory access and caching behavior, its granularity is insufficient for general bottleneck resolution.

4.3 Boosting Performance Optimization with Interactive Data Movement Visualization

The tool developed by Schaad et al. [18, 19] enables two-tier program analysis:

At the global level, static analysis (Section 3.2) is used to compile the program source code into an SDFG graph, providing an overview as shown in Figure 3. This graph, with its color-customizable nodes and edges, aids in identifying problematic program sections, especially when utilizing the automatic node and edge collapsing feature for easy zooming.

For in-depth analysis of data locality and reuse behavior, the tool uses cache simulation (Section 3.3) to offer detailed views of specific program segments, as depicted in Figure 5.

The authors successfully employed this tool to significantly optimize two applications. After pinpointing problem areas in the global view, the engineers utilized the local view for a thorough investigation and subsequent optimization of these areas.

4.4 Comparison

Among the three tools, the animation provided by Choudhury et al. [26] offers the most intuitive understanding of large-scale memory access and caching behavior. However, comprehensive program optimization requires contextual information about inefficient memory access locations, which is supplied by `MemAxes` [14] and the tool by Schaad et al. [18]. Of all three, Schaad et al.'s tool provides the most detailed low-level visualizations. The tool's ability to depict the influence of data layout on cache hit ratio proves invaluable for optimizing data locality. However, the tool's reliance on cache simulation necessitates consideration of parameterization, as discussed in Section 3.3.

5 Conclusions

Conclusion

Future Work:

- Data Gathering and Visualizations can always be improved
- But we can use the gathered data to automatically optimize programs [17] 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)* (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] Sameer Shende. "Profiling and tracing in linux". In: *Proceedings of the Extreme Linux Workshop*. Vol. 2. 1999.

[13] Marty Itzkowitz et al. "Memory profiling using hardware counters". In: *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*. 2003, p. 17.

[14] 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.

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

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

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

[20] 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.

[21] William S Moses et al. "Polygeist: Raising C to polyhedral MLIR". In: *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE. 2021, pp. 45–59.

[22] 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.

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

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

[25] 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.

[26] 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.

[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.