Hardware-Software Co-designed Near-Cache Accelerator for Graph Pattern Mining

Julian Pavon, Ivan Vargas Valdivieso, Osman Ünsal and Adrian Cristal
Department of Computing Science
Barcelona Supercomputing Center
Barcelona, Spain
Email: {julian.pavon, ivan.vargas, osman.unsal, adrian.cristal}@bsc.es

***Abstract*—Graph Pattern Mining (GPM) algorithms extract meaningful information within graph structures, making them fundamental building blocks for multiple application domains. However, their performance in multicore CPUs is bottlenecked by cache pollution and hard-to-predict divergence control caused by index matching operations that dominate the execution time.**

To address these challenges, this paper introduces Aventado, a hardware-software co-designed Near-Cache Accelerator for GPM workloads on commercial multi-core CPUs. By executing index matching operations near the Last-Level Cache (LLC), Aventado reduces data movement and cache pollution in upper cache levels while minimizing divergence control and enhancing parallelism. To this end, Aventado integrates a parallel control logic that simultaneously executes multiple index matching operations, maximizing utilization of the LLC bandwidth and enhancing overall execution efficiency. Furthermore, Aventado includes virtual memory support, ensuring compatibility with commodity operating systems. Designed as a decoupled programmable accelerator, it operates via memory-mapped registers. Our evaluation demonstrates that Aventado outperforms software and hardware approaches by $26.7\times$ and $1.7\times$ respectively, while incurring a negligible area overhead of 0.3% over the CPU baseline.

I. Introduction

Graph Pattern Mining (GPM) algorithms identify specific patterns within an input graphs, but their high computational cost often leads to long execution times [1], [2]. These algorithms follow three main steps: graph traversal using vertex and Breadth-First Search BFS orders, symmetry breaking, and set operations to identify matching subgraphs [1]–[3].

While BFS order exposes data locality [4] by reusing the data from a vertex to visit its neighbors, *set operations* introduce significant performance challenges. These operations depend on index matching, (i) resulting in cache pollution, as a large portion of the loaded elements are discarded, leading to inefficient memory utilization [3]. (ii) They incur hard-to-predict divergence control (e.g., frequent if-else branches), which reduces the effectiveness of branch predictors.

Prior work has explored various approaches to improve the performance of commercial CPUs for GPM algorithms, including software-based pruning techniques [2], [5], [6], as well as Near-Data Processing (NDP) architectures [3], [6]. **Software-based approaches** improve performance by eliminating redundant computations [2], [5] and reducing the index matching search space [6]. However, they do not address the inherent cache pollution and divergence control caused by index matching operations. As a result, their execution time remains dominated by inefficient cache utilization and frequent branch mispredictions. On the other hand, **NDP solutions** execute index matching directly in main memory (e.g., DRAM) [3], [6], reducing data movement and cache pollution. However, NDP has key limitations: (i) it struggles to exploit the locality in GPM algorithms, as interleaved data distribution across memory channels in commercial CPUs forces cross-channel communication, increasing NoC traffic and reducing parallelism [7], and (ii) it relies on physically contiguous data, which has limited scalability due to fragmentation and the scarcity of large contiguous memory blocks in real HPC systems and data center systems [8].

To address the aforementioned limitations, this work explores Near-Cache Processing (NCP) as a means to improve GPM performance on commercial multi-core CPUs, motivated by two key insights. (i) GPM algorithms inherently exhibit data locality, but cache pollution diminishes its benefits. Processing data directly in lower cache levels, such as the Last-Level Cache (LLC), helps mitigate cache pollution in upper levels (e.g., L1), preserving data locality benefits while avoiding cross-channel communication overheads seen in NDP. (ii) NCP can be integrated with existing virtual memory support, removing the need of physically contiguous memory.

To this end, we introduce Aventado, a hardware-software co-designed near-LLC accelerator carefully designed to accelerate index matching operations. Aventado works as a decoupled programmable unit, receiving instructions via memory-mapped registers. It handles all index matching operations, while cores manage other computations. Aventado includes virtual memory support by reusing the cache hierarchy TLBs, avoiding the need for dedicated memory mappings. Experimental results show that, on average, Aventado outperforms by $26.7\times$ and $1.7\times$ software and NDP solutions, respectively, while incurring a negligible 0.3% area overhead over the baseline CPU.

II. Aventado: Reducing cache pollution

Index matching operations generate transient data into the cache, evicting useful vertex and neighbor lists before they can be fully exploited, which reduces locality and increases memory access overhead. Computing index matching in the

LLC prevents index matching transient data from evicting critical vertex data in upper cache levels. With its larger capacity and lower eviction pressure, the LLC absorbs transient data, preserving data locality. This approach also minimizes data movement between caches, enhancing cache utilization and overall performance for GPM workloads.

To better understand the benefits of near-LLC computing, we evaluate[1] the impact of transient data in the L1 and L2 caches. To this end, load/store instructions accessing data generated by index matching operations bypass the L1 and L2 caches, directly accessing the LLC. We use the largest graph from §IV, whose memory footprint significantly exceeds the LLC capacity, allowing to represent behaviors indicative of non-cache resident workloads. As shown in Fig. 1.a, this approach reduces L1 and L2 miss rates by 73% and 34%, respectively, with only a marginal 2% increase in LLC misses. These results demonstrate that confining index matching data to the LLC significantly reduces cache pollution in upper cache levels with minimal impact on LLC performance.

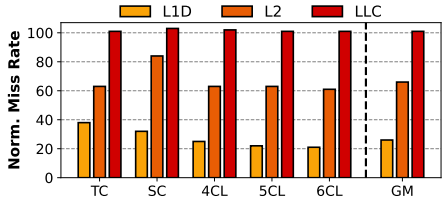


Fig. 1. Miss rate results for all the algorithms evaluated (§IV-A), comparing index matching data access directly in the LLC against the baseline system. Lower is better.

III. Aventado Design

This section details the software-hardware components of Aventado to support efficient NCP for index matching.

A. Aventado-ISA

To enable the software stack to control Aventado, we introduce four instructions, as shown in Fig. 2. These instructions are designed to execute intersection (*aventado_intersect*) and difference (*aventado_difference*) operations. The core offloads these instructions to Aventado through store instructions to memory-mapped registers.

Instruction	Functionality
aventado_intersect *[addr0], len0, [addr1], len1*	Intersects two input sets
aventado_difference *[addr0], len0, [addr1], len1*	Differences two input sets

Fig. 2. Description of Aventado instructions and their functionalities.

Fig. 3 provides an example code of extending a triangle counting algorithm with the proposed Aventado ISA. The baseline algorithm (Fig. 3.a) is based on the *index pre-comparison* approach proposed by Dai et. al. [6] that removes runtime symmetry breaking operations by pre-computing them. In the Aventado-based algorithm (Fig. 3.b), for each neighboring vertex *u1* of *u0*, *aventado_intersection* (line 10) computes the intersection between the neighbors of *u0* and *u1*.

[1]§IV-A outlines our experimental methodology

a) TC Algorithm with Software Prunning

```
1: define triangleCounting(Graph G)
2: num_triangles = 0
3: preprocessing_prune_graph(G)
4: for each u0 in G.nodes do
5:     Nu0 = G.pruned_neighbors_list[u0]
6:     for each u1 in Nu0 do
7:         Nu1 = G.pruned_neighbors_list[u1]
8:         Nu0,u1 = intersection(Nu0, Nu1)
9:         num_triangles += Nu0,u1.size()
10: return num_triangles
```

b) TC Algorithm with HALIS ISA

```
1: define triangleCounting(Graph G)
2: num_triangles = 0
3: preprocessing_prune_graph(G)
4: for each u0 in G.nodes do
5:     Nu0 = G.pruned_neighbors_list[u0]
6:     for each u1 in Nu0 do
7:         Nu1 = G.pruned_neighbors_list[u1]
8:         Nu0,u1 = aventado_intersect(Nu0, Nu0.size(),
                                       Nu1, Nu1.size())
9:         num_triangles += Nu0,u1.size()
10: return num_triangles
```

Fig. 3. Code transformation example to utilize Aventado ISA. (a) Baseline version of triangle counting algorithm with software pruning technique [6]. (b) Aventado-based triangle counting algorithm. Aventado instruction is highlighted using a green background color.

B. Aventado-Hardware

Fig. 4 provides an overview of the Aventado hardware design. Aventado features a data channel to the LLC slice, enabling Aventado to load and store data directly from the LLC during execution and a translation channel to the shared TLB (STLB) in the L2C to request virtual-to-physical translation for the memory requests done by Aventado. Aventado is composed of three main hardware modules:

Instruction Queue. It works as the interface between the proposed ISA and our NCP accelerator. When a store instruction targets Aventado' memory space, the store operation bypasses the caches, and the data is stored directly in the instruction queue ❶. Since stores are issued to memory at commit —after both address and data have been computed —this guarantees the arriving ordering of Aventado instructions to the *Instruction queue*. Additionally, Aventado' memory space is marked as non-cacheable to prevent reordering by the cache hierarchy.

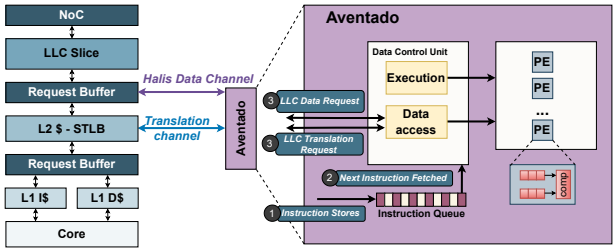


Fig. 4. Overview of Aventado hardware integrated into the CPU cache hierarchy.

Data Control Unit. This module is the main orchestrator in Aventado. It fetches instructions from the *Instruction Queue* ❷, executes the corresponding load/store operations based on the Aventado instruction ❸, and controls the execution of the processing elements (PEs) ❹.

Instructions are fetched in-order from the Instruction Queue. The *Data Control Unit* can execute up to *number of PEs* concurrent instructions (eight in our evaluation). When accessing the LLC, the *Data Control Unit* first issues an address translation request to the STLB to obtain the physical page tag. It then generates *dataLength/cacheLineSize* requests to the

LLC, where *dataLenght* is given by Aventado instructions. If data spans multiple pages, this module determines the exact number of pages required and generates the corresponding translation requests.

Processing Elements. Each processing element is composed of two cache line size buffers and a comparator, and executes the index matching operation on the data stored in the buffers.

C. Area overhead analysis

To evaluate the area impact of our design, we physically implemented all Aventado hardware modules using SystemVerilog and Synopsys' ICC2 Place and Route tool [9] with a 7nm technology node. In our evaluation, we set the instruction queue to 384 bytes (16 instructions) and 8 PEs. Aventado adds $0.08mm^2$ extra per-core, resulting in a negligible area overhead of 0.9% and 0.3% compared to the baseline core and SoC (including a Aventado instance per core), respectively.

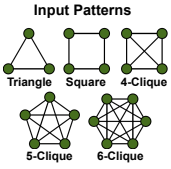
IV. Evaluation

A. Experimental Methodology

We model and evaluate Aventado using an in-house, cycle-accurate, industry-grade simulator that simultaneously runs both functional and microarchitectural performance simulation on a workload. The simulated system consists of 16 Neoverse-N1-like [10] out-of-order cores, three levels of cache (1MB LLC-slice per core) and 4 HBM2e memory channels. Each core features an Aventado module.

Benchmarks. We evaluate Aventado using GAPBS+GraphPi [5], [11] benchmarks, leveraging GAPBS's efficient data structures and GraphPi's algorithmic optimizations. We mine five different patterns shown in Fig. 5: triangle (TC), square (SC), four clique (4CL), five clique (5CL), and six clique (6CL), which have been used extensively to evaluate prior work (e.g., [1], [3], [6]). We develop a Aventado-based implementation for each evaluated pattern. Although we have evaluated our work using these patterns, Aventado is agnostic to any specific pattern and can be used to accelerate any arbitrary user-defined pattern.

Datasets. We use four real-world graphs from the sparse matrix collection [12], their main characteristics are shown in Fig. 5. These datasets are diverse in size and connectivity, thus providing different scenarios to evaluate our work.



Real-World Datasets Used for Evaluation

Graph	#Vtx	#Edge	Average Degree	Size
Wiki-Vote	7.1k	103.7k	14.6	0.5MB
Ca-Astro	18k	396k	21	2.2MB
Com-Youtube	1.1M	5.9M	5	38.7MB
Soc-Pokec	1.6M	30.6M	18	123.9 MB

Fig. 5. Input patterns and real-world datasets used in the evaluation.

Other Evaluated Approaches. Together with Aventado, we evaluated NDMiner [3] and DIMMining [6], two NDP accelerators directly integrated into DIMM modules. To obtain their performance numbers, we run their CPU baseline code using the same baseline CPUs and multiply speedup factors for commonly evaluated patterns and input graphs.

B. Results

Fig. 6 compares Aventado' performance with the baseline software and NDP accelerators. All reported results are normalized to the GAPBS+GraphPi baseline.

Aventado versus the baseline software. On average, Aventado outperforms the baseline by 19.4×, 12.8×, 29.6×, 34.5× and 37.1× for TC, SC, 4CL, 5CL, and 6CL respectively. These performance gains are mainly due to Aventado (i) reducing cache pollution in L1 and L2, (ii) reducing data movement between the cache hierarchy and core, and (iii) efficiently accelerating index matching through specialized hardware. We make two observations: First, dense patterns (e.g., 4CL) cause more cache pollution than sparse patterns (e.g., SC) due to stricter symmetry breaking requirements [3], thus, Aventado performs better for dense patterns. Second, since larger patterns require more index matching operations, the performance benefits of Aventado scale with the number of vertices in the mined pattern.

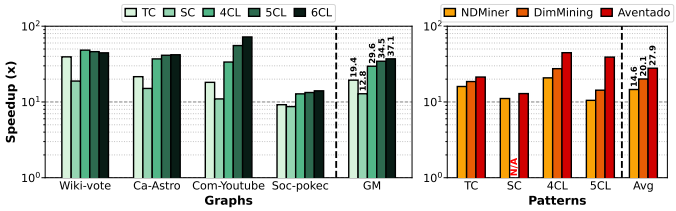


Fig. 6. Performance comparison between Aventado, NDMiner [3], and DimMining [6]. All results are normalized to the GAPBS+GraphPi [11] [5] baseline.

Aventado versus state-of-the-art NDP accelerators On average, Aventado outperforms NDminer and DIMMining by 1.9× and 1.4×, respectively. We make two observations: First, compared to NDP, Aventado avoids cross-channel communication, effectively exploiting data locality in GPM algorithms. As a result, Aventado significantly outperforms the two evaluated NDP solutions. Second, NDP solutions implement specialized hardware, such as large reordering tables [3] or systolic arrays [6], to accelerate index matching. While these hardware components are optimized for specific GPM operations, delivering significant performance benefits, they can be bulky and remain idle for extended periods in real HPC and data center environments that process diverse query workloads. In contrast, Aventado offers a cost-effective solution by seamlessly integrating into a general-purpose CPU pipeline with negligible silicon area overhead.

V. Conclusions

We present Aventado, a novel hardware-software co-designed near-cache accelerator that provides efficient hardware support for index matching operations in GPM workloads. While Aventado is agnostic to any user-defined pattern, it is also adaptable to any general-purpose architecture. Our evaluation shows that Aventado is highly area efficient (0.5% overhead) while providing 26.7× and 1.7× better performance than software and hardware baselines, respectively.

REFERENCES

[1] X. Chen, T. Huang, S. Xu, T. Bourgeat, C. Chung, and A. Arvind, "Flexminer: A pattern-aware accelerator for graph pattern mining," in *ISCA*, 2021.

[2] D. Mawhirter, S. Reinehr, C. Holmes, T. Liu, and B. Wu, "Graphzero: Breaking symmetry for efficient graph mining," *arXiv preprint arXiv:1911.12877*, 2019.

[3] N. Talati, H. Ye, Y. Yang, L. Belayneh, K.-Y. Chen, D. Blaauw, T. Mudge, and R. Dreslinski, "Ndminer: accelerating graph pattern mining using near data processing," in *ISCA*, 2022.

[4] A. Mukkara, N. Beckmann, M. Abeydeera, X. Ma, and D. Sanchez, "Exploiting locality in graph analytics through hardware-accelerated traversal scheduling," in *MICRO*, 2018.

[5] T. Shi, M. Zhai, Y. Xu, and J. Zhai, "Graphpi: high performance graph pattern matching through effective redundancy elimination," in *SC*, 2020.

[6] G. Dai, Z. Zhu, T. Fu, C. Wei, B. Wang, X. Li, Y. Xie, H. Yang, and Y. Wang, "Dimmining: pruning-efficient and parallel graph mining on near-memory-computing," in *ISCA*, 2022.

[7] B. C. Schwedock and N. Beckmann, "Leviathan: A unified system for general-purpose near-data computing," in *MICRO*, 2024.

[8] K. Zhao, K. Xue, Z. Wang, D. Schatzberg, L. Yang, A. Manousis, J. Weiner, R. Van Riel, B. Sharma, C. Tang *et al.*, "Contiguitas: The pursuit of physical memory contiguity in datacenters," in *ISCA*, 2023.

[9] Synopsys, https://www.synopsys.com/.

[10] A. Pellegrini, N. Stephens, M. Bruce, Y. Ishii, J. Pusdesris, A. Raja, C. Abernathy, J. Koppanalil, T. Ringe, A. Tummala *et al.*, "The arm neoverse n1 platform: Building blocks for the next-gen cloud-to-edge infrastructure soc," *IEEE Micro*, vol. 40, no. 2, pp. 53–62, 2020.

[11] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite," 2017. [Online]. Available: https://arxiv.org/abs/1508.03619

[12] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *TOMS*, 2011.