Motivation: General Memory Specialization for Massive Multi-Cores

In the last two decades, computer architects have heavily relied on *specialization* and *scaling up* to continue performance and energy efficiency improvement as Moore's law fading away. The former customizes the system for particular program behaviors (e.g. the neural engine in Apple chips to accelerate machine learning), while the latter evolves into massive multi-core systems (e.g. 96 cores for AMD EPYC 9654 CPU).

This works until we hit the "memory wall" – as modern systems continue to scale up, data movements have become increasingly the bottleneck. With the 7nm process, it takes 1.38pJ to multiply two fp32, but 28pJ to read them from a 1MB SRAM and ~100pJ/hop to move them within the network on chip (NoC). The latency also suffers from a similar trend. Unfortunately, as shown in Fig 1(a), conventional memory systems are extremely inefficient in reducing data movements, suffering from excessive NoC traffic and limited off-chip bandwidth to bring the data to computing cores.

These inefficiencies originate from the essential core-centric view: the memory hierarchy simply reacts to individual requests from the core, but is unaware of *high-level program behaviors*. There are attempts to exploit programs' memory semantics to make the memory more efficient, e.g., caching frequently reused data, prefetching future data, etc. However, these microarchitectural approaches are fundamentally *oblivious*, as they have to guess highly irregular and transient memory semantics from the *primitive* memory abstraction of simple load and store instructions.

This calls for a fundamental redesign of the memory interface to bridge the semantic gap and enable *general memory specialization*. Unlike primitive load/store instructions on a few bytes, the extended memory interface should express rich memory semantics. Therefore, the memory system can promptly adjust to evolving program behaviors and efficiently orchestrate data and computation together throughout the entire system. For example in Fig 1(b), ❶ simple computations can be directly associated with memory requests and naturally distributed across the memory hierarchy without bringing all the data to the core; ❷ high-level access patterns enable proactive communication of intermediate results; ❸ data affinity information, i.e. data X and Y are used together and should be close to each other, unlocks automatic data layout optimization to further reduce data movements. More importantly, the new interface should integrate seamlessly with conventional von Neumann ISAs, enabling end-to-end memory specialization while maintaining generality and transparency.

Overall, my research enables general memory specialization for massive multi-core systems that unlocks order-of-magnitude speedup/energy efficiency on plain-C programs – With semantic-rich memory abstractions to fundamentally bridge the memory semantic gap, the computation can be freely scheduled in the system near the data, and the data can be carefully mapped to and communicated between memory resources to provide maximal locality and parallelism. Such data-computation orchestration is the key to continuing the performance and energy efficiency scaling.

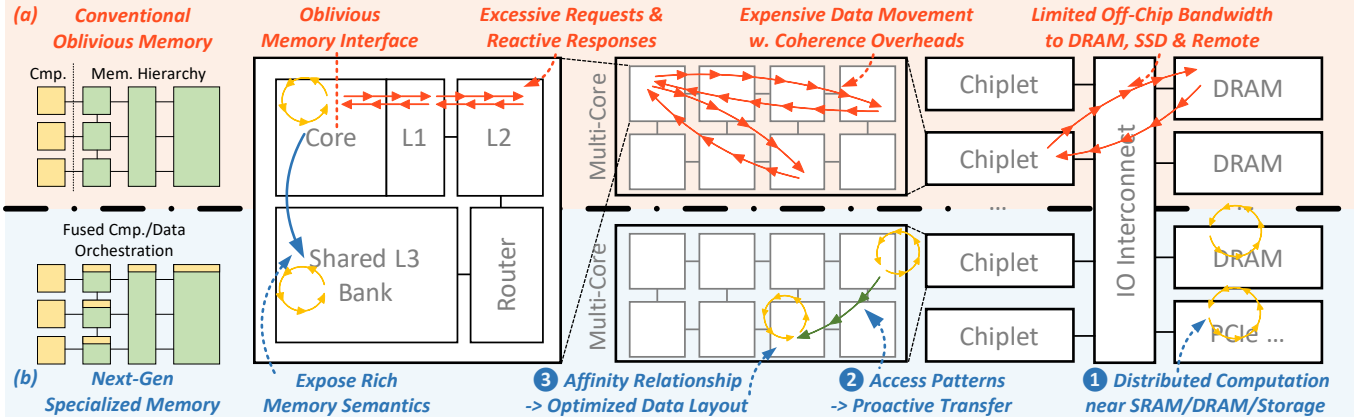
Key Contributions

- **Intellectual Contribution:** I developed end-to-end support for general memory specialization that breaks the long-existing semantic gap between the original program and the microarchitecture, with a novel architectural abstraction and thorough software-hardware codesign.
- **Academia Record:** I have published 8 peer-reviewed papers on major top-tier computer architecture conferences (ISCA, HPCA, ASPLOS, MICRO), with 5 as the first author.
- **Awards**: I received best paper runner-up in HPCA '21 and MICRO '22, and Micro Top Picks 2020 Honorable Mention.
- **Industry Impact:** My research on near-data computing has also influenced Nvidia latest Hopper GPU and Intel 4th Gen Xeon Scalable CPU architecture, which significantly reduces data movement costs in real hyperscale data centers.
- **Community:** My research work is open-sourced. I also serve as a maintainer for gem5, a widely used open source simulator for computer architecture, and contributed my implementation of AVX vector instructions.

Accomplished Works: On-Chip Data-Computation Orchestration

My doctoral studies explore three fundamental aspects for general memory specialization: What is the fundamental memory abstraction needed for such specialization? What are the new opportunities to co-design software and hardware with the new abstraction? How to effectively apply such abstraction to various computing and memory substrates?

Figure 1: Conventional Oblivious (Top) vs. Future Specialized Memory System (Bottom). View in Color



Abstraction for Data-Computation Orchestration Conventional von Neumann ISAs employ a very primitive memory abstraction: fine-grained reads/writes on a few bytes (red arrows in Fig.1). The memory subsystem simply reacts to individual requests but lacks a holistic view of program behaviors. However, such high-level information is available in the original program but lost once compiled into this primitive abstraction. Therefore, the memory abstraction should be fundamentally enhanced to bridge this semantic gap – it should be general to capture a wide range of program behaviors, while still providing sequential semantics for seamless interaction with the remaining program.

Our key insight is that memory access patterns can serve as the missing cornerstone of the next-gen memory abstraction. We call them *streams*, which define sequences of memory accesses as the program executes instead of focusing on a single request. Streams are flexible, from simple strided accesses to complex data-dependent accesses such as indirect `A[B[i]]` and pointer-chasing `p=p.next`. Streams are also prevalent, covering on average more than 60% memory accesses [1]. This losslessly captures long-term program behaviors, unlocking many specializing opportunities. To name a few, the reuse information could guide the system to bypass certain cache levels without reuse. The access pattern enables deep yet accurate prefetching with efficient address generation. Without the stream abstraction, these require complex hardware.

Decoupled Memory Accesses in General Purpose Processors My first work leverages this new abstraction to reduce memory access overheads and enable "perfect" prefetching in general-purpose architectures [1]. As streams represent the entire access pattern, a small dedicated hardware unit (we call it the stream engine) can be configured before entering the loop to directly access the memory. Stream values are communicated between the stream engine and the rest of the program (executed by the core) via remapped registers. This prevents wasting precious pipeline resources, e.g. issuing slots, reorder buffer entries, etc., on repeated address generation and memory instructions, which reduces 30% executed instructions. More importantly, stream patterns are *precise*, enabling *deep* and *accurate* prefetching to hide the memory latency. Overall, decoupling streams yields $1.67\times$ speedup and $1.53\times$ energy efficiency, and even outperforms an oracle helper thread that runs ahead of the core to prefetch data due to the reduced instruction count.

This also demonstrates some interesting tradeoffs besides the significant performance and energy efficiency improvement. For example, with streams, a much smaller out-of-order core with issue width 2 can outperform the one with issue width 8 by $1.43\times$. This technique has been adopted in the tensor memory accelerator in the latest Nvidia GPU, which can be configured with up to 5-dimensional affine access pattern and aggressively prefetch the tensor data, as well as the data stream accelerator (DSA) in the 4th Gen Xeon CPU. This clearly demonstrates the power of a better memory interface.

Near-Cache Computing Unfortunately, so far the *memory hierarchy* is still *oblivious* – it simply reacts to individual core/stream requests, but lacks a holistic view to make informed decisions about prefetching, evicting, bypassing, or even performing NDC. Instead, we enable a *proactive* cache system by offloading the stream pattern to L3 banks [2]. Once offloaded, streams proactively transfer data back to the core and bypass the private caches without reuse. Also, streams automatically migrate between L3 banks following the access pattern. Synchronization between the core and streams is coarse-grained, i.e. one message per multiple iterations, but still precise to maintain sequential semantics for transparent offloading. This enables deep, low-cost, yet accurate prefetching without a complex OOO pipeline and a system with 4-wide inorder cores even outperforms one with 8-wide OOO cores and aggressive prefetchers with $3\times$ less energy.

Furthermore, we can extend streams beyond pure address generation to general computations that consume or produce stream values. This enables *near-stream computing* [3], where computations are scheduled *along with* stream accesses to improve locality and reduce unnecessary data movement. E.g. in vec-add `C[i]=A[i]+B[i]`, instead of fetching all three arrays to the core, performing the addition and writing back, we can offload three streams $S_{A,B,C}$, with S_A and S_B forwarding their data to S_C. S_C can directly write back the result in place. Irregular workloads, e.g. graphs, trees, benefit even more from this new technique as irregular accesses usually have lower locality in private caches. Evaluated on workloads with various access patterns (affine, irregular indirect, and pointer-chasing), 52% of operations can be associated with streams, and over 90% of them are offloaded to L3 cache. This yields 76% network traffic reduction and $2.13\times$ speedup over a state-of-art NDC technique, enabling scaling to many cores without sacrificing transparency or generality.

Affinity Optimized Data Layout After offloading computations near-data, the data layout dictates the spatial locality and parallelism and is especially critical when it involves multiple data structures. E.g. in vec-add `C[]=A[]+B[]`, `A[i]`, `B[i]` and `C[i]` should be mapped to the same or neighboring location to avoid unnecessary forwarding traffic or even pathological network traffic patterns. Another example is indirect accesses `V[E[i]]` in CSR graphs, where the edge `E[i]` should be close to the pointed vertex `V[]`, to reduce the indirect traffic. Despite its importance, generations of NDC works either ignore it or require manual placement, which is technically intractable as it requires subtle coordination through the system: to optimize the layout for spatial affinity and parallelism, to dynamically adjust to program phases, to maintain the virtual address space, etc. All these boil down to a lightweight yet expressive spatial affinity abstraction, i.e. X should be close to Y, that has always been missing in conventional programming interfaces. This is not trivial, as arbitrary affinity relationships vary from large arrays to nodes of cache line size in those pointer-based data structures.

To tackle this, we propose affinity alloc, the first general framework to directly optimize data layout, and the first one to codesign data structures to improve data affinity for NDC [4]. The idea is to capture the essential data affinity relationship in a lightweight memory allocator interface. For example, when allocating the vectors `A[N]`, `B[N]` and `C[N]`, the programmer can specify that they should be element-wise aligned for NDC. Another example is pointer-based data structures, e.g. linked list, with the new node allocated closer to the previous one. It adopts a clean layered design with co-optimized data structures, runtime libraries, minimal OS extensions and μArch tables. Evaluated with both dense and irregular applications, it achieves $2.26\times$ speedup over a state-of-the-art NDC technique. Also, it achieves $1.75\times$ speedup and 65% traffic reduction on real-world power-law graphs with a high node degree, which are notoriously hard to partition.

Fusing Near-/In-Cache Computing While this new abstraction captures the essential program semantics, it remains neutral to the underlying hardware details. This makes it possible to apply it to improve the programmability and usability of other emerging computing paradigms, or even to fuse multiple paradigms. A particularly interesting case is bit-serial in-cache computing, in which each cache bitline is a vector lane, forming a massive vector unit (e.g. a 64MB L3 cache with 256x256 SRAM arrays has 2M vector lanes). Due to the massive parallelism, it is especially effective for large dense computations, e.g. matrix operations, but is not as efficient for irregular computations. Hence we need a hybrid paradigm.

Streams capture essential program semantics to effectively fuse both paradigms. First, dense regular operations are often represented as computations involving affine streams. By relaxing the sequential semantics of stream, we can easily exploit the massive parallelism provided by in-cache computing without introducing new abstractions. Also, irregular sparse operations can still be handled as near-cache computing using indirect and pointer-chasing streams. We evaluated PointNet++, a state-of-the-art point cloud application. The dense multi-layer perceptron (MLP) is handled by in-cache computing, while irregular operations, e.g. sampling centroids, gathering feature vectors, are left as near-cache computing. This unifies two computing paradigms using a *single* abstraction, with $1.92\times$ speedup over CPU ($1.20\times$ over single paradigm) [5].

Future Works: Heterogeneous, Efficient and Secure Memory Specialization

Embracing Heterogeneity and Disaggregation My prior work on hybrid near-cache and in-cache computing already demonstrates the power of a heterogeneous near-data computing system, in which computation is flexibly offloaded to suitable computing substrates with a unified abstraction. There is huge potential in this direction as the system continues to scale up with more and more computing and memory resources connected. For example, AMD adopts chiplet architectures with vertically stacked L3 caches, providing massive on-chip storage (1152MB for EPYC 9684X) and making near-cache computing even more attractive. When the data can not fit in the on-chip storage, it is natural to further decouple compuate logics into the memory hierarchy, i.e. DRAM, persistent memory, or even storage. These memory resources also become more and more disaggregated to provide more flexible provisioning of memory capacity and bandwidth shared between multiple hosts, but at the price of higher latency and limited bandwidth for remote accesses, which could also benefit from NDC. The compute substrates can also be heterogeneous, from general-purpose CPU and GPU architectures to more domain-specific accelerators and low-level in-memory computing using bitline operations.

It is certainly challenging to adopt NDC on such a heterogeneous and massive system: how to optimize the data layout to balance parallelism and locality, how to dynamically schedule computation to the efficient substrate, and most importantly, how to design a unified program abstraction that fuses all these paradigms and eases the pain of massive adoption. It certainly requires full-stack codesign from user applications, runtime libraries to architectural interfaces and microarchitectures to tackle these challenges. Based on our previous success on on-chip NDC, I believe our approach is promising to enable a more heterogeneous NDC system.

Fully Utilized General-Purpose Architectures While the compute throughput in general architectures is constantly increasing, they are severely underutilized due to inefficiencies in the memory hierarchy and core pipelines. For example, high-end GPUs from Nvidia in 2023 (H100) provide $12.1\times$ more tensor operation throughput, but only $2.5\times$ memory capacity and $2.2\times$ memory bandwidth compared to prior models in 2018 (V100). To improve the performance and energy efficiency, architects introduce many domain specific accelerators, e.g. TPU from Google, NPUs in mobile SoCs, etc., with explicit dataflow to improve utilization of the compute units.

Interestingly, there are some high-level similarities between today's general-purpose processors and domain-specific accelerators: they both employ a tiled design with cores or processing elements (PEs) connected with an on-chip network. But when we zoom in, they are very different in terms of orchestration of data and computation – general-purpose processors usually manages the data implicitly with caches, while accelerators usually explicitly dictate how data is moved and reused by mapping a dataflow graph.

Our prior work already demonstrated that explicitly encoding the access pattern in the memory abstraction unlocks new opportunities for data orchestration. This could eventually creating a spatial accelerator overlay on general architectures, significantly improving the utilization of existing computing resources and avoiding data movement back and forth between general-purpose processors and accelerators.

Security and Memory Specialization While my previous research focuses more on improving the generality, performance and energy efficiency of memory specialization, security is an inevitable challenge, and at the same time opportunity, for the wide adoption of memory specialization. On the one hand, with more and more information and logic being offloaded and flown across the entire system, we need to be extremely careful to avoid introducing new side channels. On the other hand, memory specialization may help improve security. For example, a wide range of attacks exploits the on-chip resources such as caches, prefetchers, interconnects, etc. If sensitive computation is offloaded off-chip and performed near memory, all these attacks are naturally defended. Specialized computing units near memory could also help efficiently encrypt and decrypt data on the fly, providing higher security without impeding the performance. There are many open and exciting questions about the intersection between security and memory specialization.

End Note I envision a fully flexible and heterogeneous memory-specialized system in the next decade, in which data and computation are harmoniously orchestrated to provide next-gen performance and energy efficiency. I would like to to explore all these directions discussed above to realize this vision. I am also dedicated to seeing these ideas adopted in the industry. The introduction of streams in Nvidia makes this practical to explore many of these ideas in the near term, and I will work with contacts at Intel who sponsored much of this work.

[1] **Zhengrong Wang** and Tony Nowatzki. "Stream-Based Memory Access Specialization for General Purpose Processors". In: *2019 46th International Symposium on Computer Architecture (ISCA)*. 2019. DOI: 10.1145/3307650.3322229.

[2] **Zhengrong Wang**, Jian Weng, Jason Lowe-Power, Jayesh Gaur, and Tony Nowatzki. "Stream Floating: Enabling Proactive and Decentralized Cache Optimizations". In: *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 2021. DOI: 10.1109/HPCA51647.2021.00060.

[3] **Zhengrong Wang**, Jian Weng, Sihao Liu, and Tony Nowatzki. "Near-Stream Computing: General and Transparent Near-Cache Acceleration". In: *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 2022. DOI: 10.1109/HPCA53966.2022.00032.

[4] **Zhengrong Wang**, Christopher Liu, Nathan Beckmann, and Tony Nowatzki. "Affinity Alloc: Taming Not-So Near-Data Computing". In: *2023 56th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2023. DOI: 10.1145/3613424.3623778.

[5] **Zhengrong Wang**, Christopher Liu, Aman Arora, Lizy John, and Tony Nowatzki. "Infinity Stream: Portable and Programmer-Friendly In-/Near-Memory Fusion". In: *2023 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2023. DOI: 10.1145/3582016.3582032.