GraphAttack: Optimizing Data Supply for Graph Applications on In-Order Multicore Architectures

ANINDA MANOCHA, Princeton University
TYLER SORENSEN, UC Santa Cruz
ESIN TURECI and OPEOLUWA MATTHEWS, Princeton University
JUAN L. ARAGÓN, Universidad de Murcia
MARGARET MARTONOSI, Princeton University

Graph structures are a natural representation of important and pervasive data. While graph applications have significant parallelism, their characteristic pointer indirect loads to neighbor data hinder scalability to large datasets on multicore systems. A scalable and efficient system must tolerate latency while leveraging data parallelism across millions of vertices. Modern Out-of-Order (OoO) cores inherently tolerate a fraction of long latencies, but become clogged when running severely memory-bound applications. Combined with large power/area footprints, this limits their parallel scaling potential and, consequently, the gains that existing software frameworks can achieve. Conversely, accelerator and memory hierarchy designs provide performant hardware specializations, but cannot support diverse application demands.

To address these shortcomings, we present GraphAttack, a hardware-software data supply approach that accelerates graph applications on in-order multicore architectures. GraphAttack proposes compiler passes to (1) identify idiomatic long-latency loads and (2) slice programs along these loads into data Producer/Consumer threads to map onto pairs of parallel cores. Each pair shares a communication queue; the Producer asynchronously issues long-latency loads, whose results are buffered in the queue and used by the Consumer. This scheme drastically increases memory-level parallelism (MLP) to mitigate latency bottlenecks. In equal-area comparisons, GraphAttack outperforms OoO cores, do-all parallelism, prefetching, and prior decoupling approaches, achieving a 2.87× speedup and 8.61× gain in energy efficiency across a range of graph applications. These improvements scale; GraphAttack achieves a 3× speedup over 64 parallel cores. Lastly, it has pragmatic design principles; it enhances in-order architectures that are gaining increasing open-source support.

CCS Concepts: • **Computer systems organization → Multicore architectures**; **Heterogeneous (hybrid) systems**; **Special purpose systems;**

This research was supported in part by the DARPA SDH Program under agreement No. FA8650-18-2-7862, NSF Award No. 1763838, and the U.S. Government. Aninda Manocha was supported by the NSF Graduate Research Fellowship. Prof. Aragón was supported by the Spanish State Research Agency under grant TIN2016-75344-R (AEI/FEDER, EU) and by Fundación Séneca-Agencia de Ciencia y Tecnología, Región de Murcia, Programa Jiménez de la Espada (20580/EE/18). The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies or endorsements, either expressed or implied, of DARPA, NSF, or the U.S. Government.
Authors' addresses: A. Manocha, E. Tureci, O. Matthews, and M. Martonosi, Princeton University, Princeton, NJ 08544; emails: {amanocha, esin.tureci}@princeton.edu, luwa.matthews@gmail.com, mrm@princeton.edu; T. Sorensen, UC Santa Cruz, Santa Cruz, CA 95064; email: tyler.sorensen@ucsc.edu; J. L. Aragón, Universidad de Murcia, Murcia, 11 30100, Spain; email: jlaragon@um.es.

1544-3566/2021/09-ART53 $15.00
https://doi.org/10.1145/3469846

Additional Key Words and Phrases: Graph analytics, hardware-software co-design, parallelism

ACM Reference format:
Aninda Manocha, Tyler Sorensen, Esin Tureci, Opeoluwa Matthews, Juan L. Aragón, and Margaret Martonosi. 2021. GraphAttack: Optimizing Data Supply for Graph Applications on In-Order Multicore Architectures. *ACM Trans. Arch. Code Optim.* 18, 4, Article 53 (September 2021), 26 pages.
https://doi.org/10.1145/3469846

1 INTRODUCTION

Graphs provide a natural abstraction for relational data and as such, they efficiently capture key properties of complex and pervasive systems, such as social networks and language models. Graph algorithms operate over these structures and have applications in many domains, including recommendation systems, information spread modeling, and disease classification [23, 25, 46, 67]. In the era of big data, graph applications process datasets that are growing rapidly and the slowdown of chip frequency and transistor scaling necessitates innovations that span architecture, compiler, and application design to achieve performance gains.

While significant strides in software/hardware acceleration have been made for other application domains in the big data era, e.g., DNNs, graph applications have not experienced similar successes. There are three fundamental problems specific to graph applications that have hindered this process. (1) Unlike DNNs, graph algorithms do not have repeating blocks of homogeneously parallel, dense computations. Instead, both computation and parallelization are highly variable depending on both the problem and the input graph structure [51], which is far from ideal for accelerator design [57]. (2) Any hardware acceleration for graph applications must compete for precious resources, e.g., area/energy, in a system design. (3) Graph applications are characterized by irregular memory access patterns that pose a data supply challenge. Each vertex computation typically involves several pointer indirect accesses and conditional updates to node data. With datasets much larger than the **last-level cache** (**LLC**), these neighbor accesses often miss in the cache, leading to long-latency DRAM accesses that occupy (or stall) system resources. These accesses are difficult to predict and have variable dependent computation chains, which confound prefetching, runahead, and decoupled architecture designs.

A performant and scalable solution for graph processing should have: (1) hardware efficiency, i.e., high utilization of system resources, and (2) sufficient programmability, i.e., the ability to program and execute a variety of algorithm implementations. As a final design philosophy, we believe that solutions must also have (3) an accessible architecture implementation that can feasibly be integrated into existing open-source hardware design and compiler frameworks. In the golden age of computer architecture, it is now more possible than ever to realize hardware-software co-designs, even outside of an established chip design company. Existing solutions (both in software and hardware) have fallen short of fully meeting these requirements, especially in today's increasingly heterogeneous graph application landscape.

Existing Software Solutions for Graph Applications. Software frameworks for graph applications aim for efficient parallel solutions via shared memory CPUs [52, 66] and GPUs [44, 58]. These frameworks largely target the data-parallel loops across vertices (and sometimes edges) and aim to parallelize these loops efficiently in a homogeneous (i.e., same program per thread and single device) manner, an approach also known as *do-all parallelism*. They use many threads to perform irregular accesses simultaneously, **exploiting memory-level parallelism (MLP)**. Many innovations in these frameworks involve load-balancing, efficient data representation, synchronization reduction, and dynamic switching between neighbor flow directions (push or pull). While

these approaches show improvements over baseline implementations, they are limited by underutilized hardware. For CPUs, the target cores are typically commercial **Out-of-Order** (**OoO**) processors, which are designed with latency tolerance mechanisms, e.g., **load-store queues** (**LSQs**) and **reorder buffers** (**ROBs**). These components have high area and energy overheads and yet prior work has shown that without additional hardware augmentation, they struggle with graph analytic kernels due to complex load–load dependency chains, leaving components clogged and poorly utilized [11, 36]. Thus, OoO designs are not an attractive choice for graph analytics at a small scale, e.g., edge devices that are severely resource-constrained, nor a large scale, e.g., data centers that aim for multicore scalability. GPUs on the other hand have poor performance when contiguous threads access non-contiguous memory regions, which stems from the **neighbor memory accesses (NMAs)** in graph applications [39].

Existing Hardware Solutions for Graph Applications. To address the inefficiencies of modern chips for graph applications, several areas of hardware innovations have been proposed. Specialized prefetchers [6–8, 11, 54, 63] offer flexible programming models, but incur high area and energy overheads relative to simple, **in-order** (**InO**) cores, as well as additional memory traffic. Furthermore, many designs aim to service large, power-hungry OoO cores; offering modest performance improvements while incurring poor core utilization. In contrast, other approaches propose accelerator designs with specialized pipelines and memory scratchpads [21, 38, 43]. These pipelines offer high performance for specific algorithm implementations, but cannot adapt to different optimized variants produced by state-of-the-art software frameworks [37, 66]. Additionally, chip designers of a heterogeneous SoC must be careful not to overspecialize with too many accelerators for single applications, as competition for precious area and energy resources due to fixed transistor budgets and power limitations cause increasing optimization to have diminishing performance returns [17, 57].

Another set of hardware solutions propose processing near- or in-memory [5, 65, 69] to hide latency. These approaches require invasive changes to the memory system, which cause significant barriers to implementation. Memory systems are complex and small modifications can impact many components of the system, e.g., cache hierarchies, NoCs, and DRAM, requiring significant effort to verify [29]. This contrasts with *modular* solutions, designs that propose architecture components with simple interfaces for existing core models and memory systems to target. Such innovations can be contained and implemented on top of a variety of accessible open-source designs [1, 9, 64].

Decoupled Architectures for Latency Tolerance. Finally, decoupled architectures aim to provide efficient data supply by overlapping memory accesses with computation [49]. Programs are sliced to create two parallel instruction streams that handle memory and compute individually; the memory access stream runs ahead and supplies data to the computation stream in advance. More recently, DeSC [19, 20] introduced a specialized *Terminal Load Buffer*, to allow loads to be performed asynchronously with respect to core operations. The *Supply* core, responsible for memory access, can issue requests and continue with its operations, while outstanding requests have entries reserved in the buffer for their returned data. This allows the Supply to runahead of the computation stream and avoid long data supply latencies. However, requests tracked in the Terminal Load Buffer must have no dependent memory accesses, otherwise the Supply stream cannot runahead. Unfortunately, graph applications involve load-load dependency chains where control flow dependent on irregular memory accesses inhibits DeSC and existing decoupled architecture designs from leveraging efficient program slicing.

Our Approach: GraphAttack. To overcome the shortcomings in existing prior approaches for graph analytics, we present GraphAttack: a hardware-software data supply optimization for graph

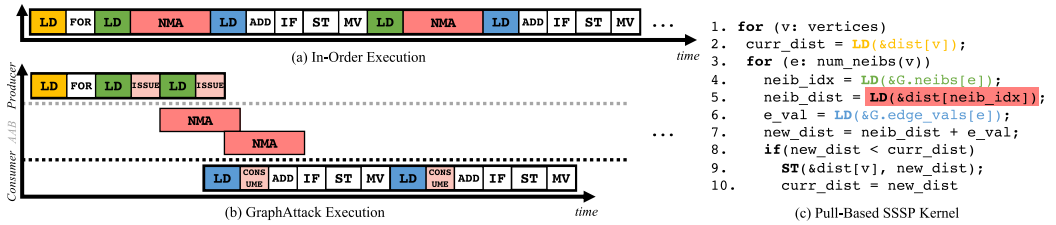

Fig. 1. Comparison of two execution timelines of the SSSP kernel (c), with the NMA highlighted in red: (a) a single InO core and (b) a GraphAttack pair. GraphAttack mitigates long-latency effects by slicing the kernel such that the Producer has no instructions dependent on the NMAs and can therefore asynchronously issue them to exploit MLP (overlapping NMAs).

application acceleration on InO multicore architectures. GraphAttack is composed of compiler techniques and modest hardware additions that together automatically identify idiomatic pointer indirect NMAs and mitigate their latencies. Our approach's design principles are in line with the open-source hardware renaissance seen in modern SoC designs. This movement entails emerging designs that opt for dense fabrics composed of many simple processing elements, e.g., InO cores, rather than large, complex OoO cores [2, 10, 57]. These architectures provide a pragmatic and accessible chip foundation for innovations in both hardware and software, enabling their development outside of a legacy chip design company.

GraphAttack builds on the key insight that the critical loops in graph analytic kernels can often be sliced into two instruction streams such that: (1) a *Producer* thread performs all computation to issue the NMAs and (2) a *Consumer* thread performs all computation dependent on their returned data. This slicing enforces a one-way dependence: the Consumer depends on values issued by the Producer, but never the opposite. In hardware, GraphAttack maps these slices onto parallel cores and augments each Producer/Consumer pair with an **asynchronous access buffer** (**AAB**) that receives asynchronous memory requests, which are loads or atomic **read-modify-writes** (**RMWs**). These accesses can be handled by the memory hierarchy simultaneously with respect to core operations, preventing the Producer core from having to wait for their data to return. The AAB issues these requests to the memory hierarchy and tracks all outstanding requests simultaneously to exploit MLP and reduce the apparent cost of each long-latency DRAM access through overlapping accesses. By targeting the AAB, the Producer runs ahead and issues long-latency memory accesses before the Consumer needs their data, preventing stalls that InO cores would typically experience.

Figure 1 illustrates how GraphAttack achieves latency tolerance with this efficient Producer/-Consumer slicing of the **Single-Source Shortest Paths** (**SSSP**) kernel (Figure 1(c)). Figure 1(a) and (b) present executions of two inner-loop iterations of the kernel with the NMAs shaded in red. Figure 1(a) displays the execution timeline of a single InO core, where all instructions are executed in order. This execution incurs the long latency of each NMA. We further characterize the consequences of this behavior in Section 2.1. Figure 1(b) demonstrates GraphAttack running on InO Producer and Consumer cores and effectively overlapping the long NMA latencies to mitigate their costs.

We developed GraphAttack in the context of a full-system multicore chip design project. As such, GraphAttack is an *optimization* on top of simple, InO multicore architectures, which enables significant software and hardware flexibility. That is, the system can be augmented with simple hardware components to accelerate certain application domains, and in the absence of acceleration opportunities, the system can fall back to traditional parallelism on the InO cores. We envision its implementation in a a standalone specialized datascience computer, e.g., [2], or a co-processor

as part of a larger heterogeneous design. This coarse-grained reconfigurability allows reuse of the same hardware for distinct application domains or kernel phases within the same workload. Furthermore, GraphAttack has a nominal area/energy overhead, as it only requires a small number of hardware queues that confine all modifications to remain independent of the cores and memory system. These modular, hardware-efficient innovations therefore maintain high programmability and portability, and leave opportunity for further specializations for other application domains.

In summary, this article presents GraphAttack, a hardware-software data supply optimization for graph application execution on InO multicore architectures. Our key contributions are:

— A characterization of widespread graph applications running on InO processors; our results show that the long latencies from *NMAs* are the prominent performance bottleneck and thus, offer an opportunity for significant optimization (Section 2).
— The GraphAttack compiler (Section 3), which given a graph application kernel (1) automatically identifies the bottleneck NMAs and (2) slices the program into Producer/Consumer instruction streams, where the Producer asynchronously issues the NMAs, and the Consumer performs all computation dependent on these accesses.
— The GraphAttack modular hardware components (Section 4), which consist of simple core-to-core communication queues that allow the Producer to efficiently provide data for the Consumer. *AAB* allow the Producer to asynchronously issue NMA memory requests (loads or RMWs) whose data can later be dequeued and used by the Consumer.
— An experimental evaluation (Section 6), which shows:
  (a) GraphAttack realizes significant performance improvements over existing approaches, i.e., a 2.87× geomean speedup and 8.61× geomean improvement in energy efficiency over OoO cores under the same area budget.
  (b) Scalable performance gains, as GraphAttack achieves a 3× speedup over traditional parallelism in a 64-core system.
  (c) The reconfigurable design of GraphAttack allows it to support a range of applications, even when the entire application cannot be efficiently decoupled. We present a case study where GraphAttack achieves a 2.21× speedup over OoO cores on **Direction-Optimizing (DO) Breadth-First Search (BFS)**, an implementation that alternates between decoupled push-based phases, and traditional do-all pull-based phases.

While the lean and modular design of our innovations allows GraphAttack to be applied to OoO cores, it yields the most energy and area efficiency while harnessing the simplicity of InO cores. This demonstrates that minimal, yet precise hardware and software tailoring can result in significant performance gains in simple multicore systems that have active open-source support.

2 BACKGROUND AND MOTIVATION

2.1 Key Bottlenecks in Graph Processing

Many graph kernels are characterized by irregular memory access patterns that occur in a data-parallel manner over vertices when their neighbors are accessed. This results in a specific class of long-latency memory accesses, which we refer to as *NMAs*. These memory accesses are pointer indirect accesses of the form **`A[B[i]]`** to read the vertex property data of neighbors. While many other application domains involve pointer indirect accesses of this form, graph applications characteristically exhibit control flow dependent on these accesses, which makes this domain particularly challenging to accelerate. Figure 1(c) presents pseudocode for the pull-based SSSP kernel, as generated by the graph processing framework: GraphIt [66]. Line 5 highlights the frequently occurring NMA, which queries the distance to each neighbor `neib_idx` along edge `e` of vertex `v`.

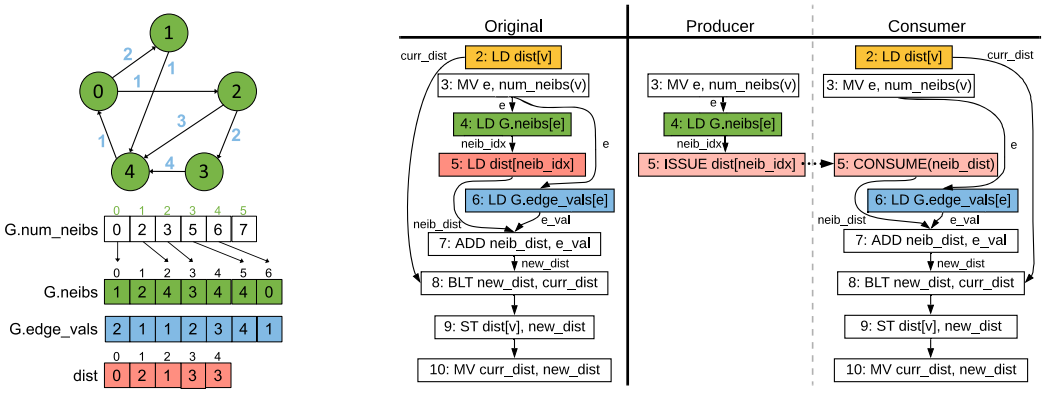

Fig. 2. LHS: CSR format efficiently represents a graph structure G with three arrays that store neighbor locations. Indexing into these arrays to update the vertex property array incurs pointer indirect memory accesses (red). RHS: GraphAttack performs compiler analysis of the dependency graph (left) of the SSSP kernel to identify the NMA (red) and create a Producer/Consumer program slicing (right) such that the Producer can asynchronously issue the NMA and runahead to exploit MLP.

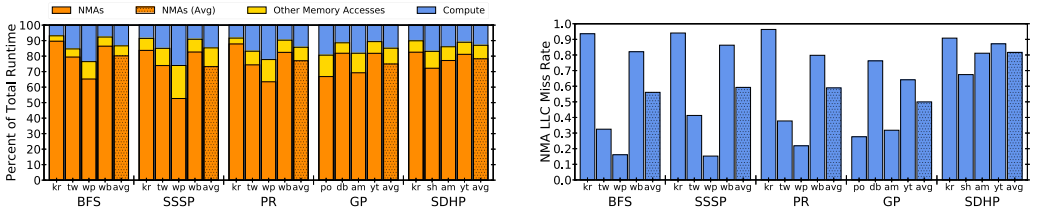

Fig. 3. LHS: Runtime analysis of an InO core running benchmark graph/sparse applications and inputs demonstrates that NMAs can significantly increase runtime if not effectively mitigated. RHS: NMAs have severe LLC miss rates, often times >50%.

The distance to v is *conditionally* updated if a shorter path (sum of edge values) to v is discovered through its neighbor `neib_idx` and the connecting edge between them with weight `e_val`.

Graph applications commonly use the **Compressed Sparse Column (CSC)** and **Compressed Sparse Row (CSR)** formats to store graph data in compact arrays that allow accessing neighbor indices and edge weights in a streaming manner. Figure 2 (left) illustrates a graph and its efficient representation with three arrays `G.num_neibs`, `G.neibs`, and `G.edge_vals`. Accessing these arrays to determine neighbor locations as well as the *vertex property array* (red), i.e., the distance array in SSSP, to read and update neighbor data, however, involves pointer indirection. The vertex property array is accessed in the innermost kernel loop, giving rise to NMAs. Moreover, parallel push-based implementations, which perform neighbor updates on different threads, *require* an atomic RMW to ensure correct updates, as different vertices (threads) can conflict on a common neighbor.

To measure the performance impact of NMAs on overall performance, we gather classic graph algorithms—BFS, SSSP, and **PageRank** (**PR**)—as well as emerging kernels used in recommendation systems and sparse neural networks—**Graph Projections** (**GP**), and **Sparse-Dense Hadamard Product** (**SDHP**). These implementations are further detailed in Table 6, and all come from competitive frameworks [55, 58, 66]. We measure the contribution of memory access latency to total runtime while running these kernels across four different datasets (Table 6) on an InO core model (see Section 5). The left-hand side of Figure 3 shows that NMA latencies take up on average ~ 5×

as much as the compute times in these graph applications. The computation in these applications comprises a small percentage, i.e., less than 20% on average, of the total runtime, highlighting the low arithmetic intensity of this domain and motivating the need for efficient data supply. The right-hand side of Figure 3 shows that on average, the NMAs miss in the LLC over 50% of the time, frequently requiring long-latency accesses to main memory. Given the high frequency of these DRAM accesses, there is opportunity to overlap them and exploit MLP. Despite their latency tolerance mechanisms, OoO cores see similar trends where their runtimes are dominated by NMA latencies. Prior works have attributed this to load–load dependency chains that inhibit these cores from exploiting significant MLP [11, 54]. These characterizations highlight that graph applications are problematic for modern commercial processors. However, these results suggest that precise targeting of NMAs can significantly improve the performance and scalability of these kernels.

2.2 Shortcomings in Existing Hardware/Sofware Solutions

Specialized Graph Processing Hardware. Fixed-function graph analytic accelerator designs and **processing-in-memory** (**PIM**) trade general-purpose flexibility for high performance through specialization. These designs are area and energy efficient for specific kernels. However, they are severely limited by the applications they support. Many accelerator pipelines and memory technologies are tailored to a *vertex-centric* programming model [5, 21, 38, 43, 65, 69] and thus not amenable to several important graph algorithms, e.g., edge-centric kernels in graph neural networks, recommendation systems, and sparse neural networks [15, 62, 68]. Their hardware additions are therefore a low return investment in chip designs that target multi-kernel/-phase applications [12, 15, 62].

Furthermore, different variants of vertex-centric kernels can have optimal performance depending on the algorithm and dataset. *Pull-based* variants are communication-efficient, as vertices only update their local values, while *push-based* implementations, where vertices push updates to neighbors, are more work-efficient at the cost of more communication often via atomic RMWs [13, 60]. In fact, for some workloads, e.g., BFS, it is beneficial to switch strategies during computation [12, 66], requiring a flexible programming model that accelerator designs lack.

Flexible Latency Tolerance. In contrast, many latency tolerance approaches build upon general-purpose cores. **Decoupled Access/Execute** (**DAE**) architectures [19, 20, 49] aim to overlap memory accesses and computation by slicing programs such that one thread, the *Access*, exclusively handles memory accesses, while the other, the *Execute*, performs value computation. Ideally, the Access runs ahead of the Execute, asynchronously issuing requests and enqueuing data without predicting access patterns. This has been successful for sparse linear algebra routines, where pointer indirect accesses have no memory dependencies and there exists sufficient compute to overlap with memory accesses. However, many graph applications suffer from *loss of decoupling* [14] events, where control flow dependencies and small compute to memory ratios limit data supply. Given this, DeSC explicitly states that it cannot feasibly accelerate graph applications [19].

The other alternative on this front is prefetching, which faces two key issues: (1) imperfect accuracy and timeliness increases cache pollution and memory bandwidth pressure and (2) adding specialized units to fetch data, e.g., long-latency pointer indirect accesses, often adds energy and area consumption. Such units can require significantly more area than an entire InO core. Prior work has used up to 12 InO cores [8] as prefetchers for one OoO core. While these InO cores may comprise a small area overhead relative to the OoO core, judicious hardware-software co-design can enable these simple InO cores to be utilized as full general-purpose cores instead of solely as prefetching units. Moreover, the OoO core can be replaced with several InO cores to unlock significant parallelism and energy/area savings. We describe this in more detail next.

Table 1. GraphAttack Handles NMAs with More Efficiency and Programmability Compared to Prior Latency Tolerance Techniques

Approach	Sufficient Latency Tolerance	Core Model	Memory Overhead	Hardware Additions/Specializations
DSWP [42]	No, partition heuristic not suitable	Any	None	Minimal (synchronization array)
DeSC [19]	No, does not identify terminal RMWs	Out-of-Order	None	Moderate (CAMs for out-of-order data supply)
IMP [63]	No, inaccurate prefetches for low-degree vertices	Any	Extra energy and traffic (inaccurate prefetches)	Moderate (prefetching engine, can be triggered in regular workloads)
DROPLET [11]	No, inaccurate prefetches for low-degree vertices	Out-of-Order	Extra energy and traffic (inaccurate prefetches)	Moderate (property prefetcher, data in page table and L2 req queue)
Software Prefetching [7]	No, compiler pass struggles with complicated control flow	Any	Extra energy and traffic (late prefetches)	None
Event-Triggered Prefetching [8]	No, compiler pass struggles with complicated control flow	Out-of-Order	Extra energy and traffic (late prefetches)	High (several in-order cores as prefetching units)
Tesseract [5]	Yes, only for VP apps	Out-of-Order	Utilization of multiple DRAM units	High (specialized 3D-stacked memory tech.)
Graphi-cionado [21]	Yes, only for VP apps with small datasets	Specialized pipeline	Reduced energy and traffic (on-chip scratchpad)	High (specialized modules and large scratchpad, for VP apps only)
HATS [32]	Yes, for datasets with sufficient locality	Any	Reduced energy and traffic (locality scheduling)	Moderate (specialized engine for each per-core cache)
GraphAttack	Yes, through Producer runahead	Any (pref. in-order)	None	Minimal (small queues)

In-Order vs. Out-of-Order Cores. OoO cores have been the staple of commercial multicore systems as they perform exceptionally well for many application domains. However, graph workloads exhibit complex load–load dependency chains whose latencies cannot be hidden by traditional OoO mechanisms, leading to poor core utilization [11, 36]. On the other hand, InO cores can be up to 5× more energy efficient per-cycle than OoO cores [31], and 30× smaller than OoO cores [8]. In light of these comparisons, our real-system, energy- and area-constrained, multicore chip design favors InO cores to harness their simplicity and energy efficiency for acceleration of graph analytics. By augmenting a simple, InO multicore system with minimal additions, our approach achieves high performance and scalability without sacrificing general-purpose flexibility.

A Taxonomy for Data Orchestration. In summary, prior approaches for latency tolerance and graph applications as well as their pitfalls can be broken down into four categories: (1) **Decoupled and Runahead Architectures** are limited in the runahead they can achieve due to complex load–load dependency chains that inhibit efficient program slicing for decoupling. Our approach offers tailored compiler techniques that leverage knowledge about graph application behavior to split these dependency chains. (2) **Prefetchers** for irregular accesses tend to be inaccurate for graph analytics due to complex control flow dependencies. Our approach avoids speculation to guarantee no data fetch inaccuracy. (3) **In-/Near- Memory** and (4) **Accelerator** designs propose invasive hardware modifications that severely limit portability and programmability. We strive for modest hardware additions that precisely target application bottlenecks in order to achieve performance on par with accelerators, yet can support a wide variety of algorithm implementations. Table 1 presents a summary of these works in comparison to GraphAttack.

2.3 GraphAttack Overview

GraphAttack overcomes the inefficiencies of prior latency tolerance approaches for graph applications without overspecializing hardware components. At a high level, our data supply approach utilizes compiler techniques to map irregular memory accesses and graph computations onto a multicore architecture augmented with modest hardware capable of performing select memory accesses asynchronously. The goal of asynchronous memory accesses is to minimize the apparent cost of each individual access by overlapping the long-latency costs of several DRAM accesses

and exploiting MLP. By slicing programs such that NMAs are decoupled from their dependent instructions and issued asynchronously, GraphAttack mitigates their latency effects and specifically addresses the control flow dependencies of graph applications that challenge prior works.

The compiler first performs an analysis of a kernel's dependency graph to automatically identify where the NMAs occur. It then slices the programs into two kernels: the Producer and the Consumer. The Producer kernel is transformed to perform all necessary instructions to issue NMA memory requests. Such instructions determine NMA addresses (and arguments for RMWs). Meanwhile, the Consumer kernel is transformed to perform all instructions dependent on the data returned by the NMAs. The compiler then replaces the NMA with a special **`ISSUE`** instruction on the Producer and **`CONSUME`** instruction on the Consumer, thereby allowing for asynchronous communication between these two decoupled kernels. The **`ISSUE`** instruction is a compiler API call to the AAB that issues the loads and produces the loaded values to a queue to be then consumed by the consumer through the **`CONSUME`** compiler API call to the AAB.

Figure 2 presents an example of the compiler's dependency graph analysis for the pull-based SSSP kernel (Figure 1(c)) whose NMA is highlighted. Slicing the program along the NMA results in Producer/Consumer threads such that the Producer does not perform any instructions dependent on the NMA. For example, in SSSP, the Consumer is responsible for the computation to update a vertex's data if a new shortest path to the vertex is discovered. This slicing must be performed carefully to ensure correct memory consistency behavior, i.e., the sliced program must guarantee the same behavior as the non-sliced program. Prior approaches (e.g., [19]) used complex hardware structures, e.g., CAMs, to check for and resolve memory conflicts between Producer and Consumer streams. Rather than require heavy-weight and invasive hardware components, we instead require that the Producer, Consumer, and the NMA all operate on disjoint addresses within a single loop iteration so that memory conflicts will not occur. This can be automatically checked with a standard alias analysis in the compiler. If the compiler is unable to verify disjoint addresses across these three components, it notifies the user and falls back to a do-all parallel compilation model. While this condition may seem strict, we found that all graph applications we evaluated satisfied this condition, with the exception of SSSP (discussed in Section 3.1).

In hardware, GraphAttack tracks outstanding asynchronous NMA requests via a small buffer, the AAB, between each Producer/Consumer pair. When the Producer executes an **`ISSUE`** instruction, it sends the memory request to the AAB, which the Consumer queries when executing a **`CONSUME`** instruction. The AAB is agnostic to the core and memory subsystem; it operates independently with *no* changes to internal core state or structure and performs accesses through the memory hierarchy with *no* changes to coherence protocols or caching policies. Despite its flexibility, GraphAttack advocates for InO cores because its precise targeting of memory latency bottlenecks yields significant speedups and even greater energy efficiency gains without complex OoO hardware. Similar to GPUs, InO cores offer simplicity that makes them ideal building blocks when designing programmable accelerators in a heterogeneous system.

With its tightly coupled compiler slicing and small hardware components, GraphAttack optimizes the performance of graph analytic kernels, as illustrated in Figure 1. Specifically, it creates an efficient program slicing where the Producer asynchronously issues NMAs and the Consumer performs dependent memory accesses. This targeting of the AAB enables overlapping of long NMA latencies, as seen in Figure 1(b). GraphAttack differs from prior decoupling approaches [19, 49] by allowing the Consumer to access memory, which enables a strict one-way dependence such that the Producer *never* depends on the Consumer. This is key to performance; removing control flow dependencies on the long-latency NMAs frees the Producer from stalling, so it can runahead and issue multiple NMAs to be overlapped. Overall, GraphAttack builds on top of general-purpose InO cores with minimal hardware additions and precise compiler techniques to improve energy

Table 2. The Three RMW Operations Supported in GraphAttack

Operation	Description
COMPARE_EXCH(addr,cmp,val)	Compares cmp to the value stored at addr; if they are equal, stores val at addr; returns the comparison result
MIN(addr,to_min)	If to_min is less than the value stored at addr, then to_min is stored at addr; returns the comparison result
ADD(addr,to_add)	Adds to_add to the value stored at addr; returns the previous value

efficiency and leverage scalability to manycore systems that can exploit large amounts of parallelism in graph analytic workloads. These design choices offer flexible support for both push- and pull-based implementations of vertex-centric applications and multi-phase data analytic workloads.

3 A COMPILER FOR PRODUCE/CONSUMER SLICING OF GRAPH APPLICATIONS

The GraphAttack compiler performs hardware-efficient program slicing to target NMAs in graph applications while preserving the original program semantics. This slicing approach is inspired by decoupled software pipelining [48], but (1) focuses on graph applications using domain-specific insights about their access patterns; and (2) utilizes a specialized buffer to perform asynchronous accesses that are key for Producer runahead in this application domain. For practical reasons, our compiler operates on C++ programs and it is implemented as a series of LLVM compiler passes. Our approach is not tied to any particular aspect of either, other than the input being an imperative programming language with constructs such as `for` loops and the ability to operate (load, store, and RMW) on memory locations.

3.1 Data Parallel Loops

The GraphAttack compiler takes as input: a data parallel `for` loop, similar to OpenMP's `for` loop pragma [41]. In graph applications, the outermost loop (over vertices) is often parallelized (line 1 in Figure 1(c)). As is common in do-all parallel loops, these loops must be free of data-races. That is, if two loop iterations access the same memory location, then neither access can be a write. In other words, loop iterations must be independent, i.e., able to be executed in any interleaving. However, some graph applications iteratively update vertex data, e.g., distances to vertices. Thus, GraphAttack extends the **data race free** (**DRF**) model to these applications by utilizing two rules with explicit use-cases:

Read-Modify-Write (RMW) Operations. Push-based implementations require atomic updates to neighbors as multiple vertices can simultaneously update a common neighbor's data. Thus, the programming model resolves inter-loop write conflicts through a set of standard RMW operations (shown in Table 2). Note that RMWs in GraphAttack guarantee *atomicity*, but not *synchronization* and can only be used for associative and commutative reduction operations over neighbor updates.

Asynchronous Updates. By default, the compiler terminates the slicing procedure if any memory conflict across loop iterations is found. This allows simple hardware implementations, e.g., those that do not need to track memory conflicts, to maintain the semantics of the original program. However, one relaxation is beneficial for certain graph applications. The user must explicitly enable this relaxation and ensure their program remains correct under the relaxed semantics.

A loop iteration may write to a memory location that another iteration reads. However, the written value is not guaranteed to be visible until the implicit barrier at the end of the parallel `for` loop. We refer to this behavior as an *asynchronous update* and it is crucial for competitive

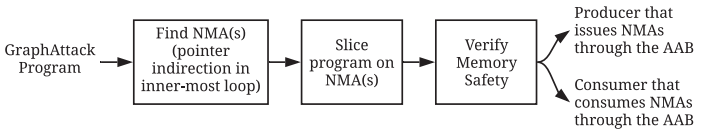

Fig. 4. The GraphAttack compiler flow.

performance in some pull-based graph kernels. This exception is required because some parallel implementations of graph applications contain potentially redundant work, as it is more scalable for multiple threads update data simultaneously, and compute on the same (potentially) stale data, rather than sequentializing updates. Because graph applications are iterative, they will continue computing until a convergence criteria is met, e.g., there are no further updates because the global best result has been discovered. Updates based on stale data will not affect the correctness of these algorithms. For example, a loop iteration in the SSSP kernel of Figure 1(c) can update its vertex many times (line 9). However, other iterations (operating in parallel) may read a stale value for this vertex when pulling neighbor updates (line 5). This will not affect the global minimum distance that has been discovered so far. Eventually, the updated values will be discovered by other threads. Because the algorithm iterates until it finds a *global* minimum, early updates are not required for correctness, but can speed up convergence. For example, SSSP on the Twitter dataset traverses roughly 2× fewer edges when allowed to asynchronously update.

3.2 Program Slicing Conditions and Procedure

The program slicing procedure is split into three steps, outlined in Figure 4. First, the compiler finds the NMA(s) by examining loop nestings and address origins. Next, the program is sliced into Producer/Consumer streams. Finally, the memory safety is verified through pointer analysis. If the compiler fails at any of these steps, the user is given a diagnostic message and the compiler falls back to a traditional do-all parallel compilation mode.

Finding NMAs. The compiler first aims to find the NMA(s) in the graph kernel. Heuristically, it searches for loads (and RMWs) in the innermost loop for which the addresses originates from an earlier load. Our approach specifically targets single pointer indirect accesses of the form **`A[B[i]]`**, which is most common in graph applications, the focus of this work. The GraphAttack compiler and architecture design have the flexibility to be extended to support different irregular memory access patterns, e.g., **`A[B[C[i]]]`**, that may occur in other workloads; we leave this to future work. For example, examining accesses in the innermost loop of the SSSP kernel (Figure 1(c)) locates the NMA as the load on line 5 because its address originates from the prior load on line 4. If the compiler cannot find an NMA, then the slicing fails.

Producer/Consumer Slicing. Once the compiler has found the NMA(s), it slices the program into two streams to be mapped onto the Producer and Consumer. In terms of a dependency graph, the Producer slice is a backwards slice, i.e., instructions collected on a reverse traversal of the dependency graph, starting from the NMA argument values. For loads, this argument is simply the memory address. For RMWs, the arguments are the address and the other arguments listed in Table 2. On the Producer, NMA(s) are replaced with an API call **`ISSUE`**, which *issues* the NMA. This instruction takes in all required arguments for the original NMA, including the address and any required values (for RMWs). Additionally, a distinct opcode indicates the type of access, i.e., load or RMW. The Producer issues the memory request through the AAB, which tracks the request and enqueues the returned value. Because the result of the request is tracked through the queues, the **`ISSUE`** instruction has no return value itself. The Consumer slice is created in four parts:

(1) The NMA(s) are replaced with an API call to **CONSUME**. This call takes no arguments and simply returns the value that the memory operation would have normally returned.
(2) A backwards slice of the control flow dependencies on the NMA(s). This pass ensures that every **ISSUE** performed by the Producer will match with a **CONSUME** on the Producer. The backwards slice needs only consider control-dependencies, as the **CONSUME** instruction has no arguments, and thus does not have any data-dependencies.
(3) A forward slice on the data returned by the **CONSUME** instructions, which allows the Consumer to perform any computation dependent on the NMA(s). All instructions identified in this slice must have access to all their original operands; thus an additional backwards slice from each instruction is performed.
(4) A backwards slice, including both data and control dependencies, on all store instructions.

We have validated this compilation scheme works across our benchmark suite (see Section 5). However, we assume that the only output from the kernel occurs through memory, e.g., memory stores. Step 4 ensures that all outputs are performed on the consumer, regardless on their dependency on the NMA. However, if the kernel can have output through other mechanisms, e.g., a syscall, then the slicing mechanism will need to be updated, e.g., by putting the syscall on the consumer.

Duplicate Instructions. Similar to prior decoupling approaches, this slicing may result in duplicate instructions appearing on the Producer and Consumer, e.g., `for` loop overheads. Because the bottleneck of graph applications is clearly the latency from the NMA, there is flexibility in how duplicate instructions are handled, so long as they preserve the semantics of the original program. Here, we detail how different types of duplicated instructions are handled.

If a duplicated instruction performs logic or arithmetic, then it is performed on both slices (e.g., node 3 in Figure 2). For duplicated load instructions, we opt to perform the load only on the Producer, and then communicate the value to the Consumer through the AAB, which feeds into the Communication Queue (described in Section 4), which buffers data in issue order similar to DAE architectures [19, 49]. The Producer first loads the value using a normal memory access operation, and then sends the value to the Consumer via an API call **PRODUCE**. The Consumer uses the same API call to retrieve produced values from the Communication Queue as it does for the asynchronously issued values, i.e., using **CONSUME**. Having only one processor perform the load alleviates bandwidth pressure, which is important, especially for graph applications. RMW instructions are sliced similarly; they are performed entirely on the Producer, and the result is sent to the Consumer through the communication queue.

We note that the Consumer handles all store operations, as they typically do not appear in the Producer backward slices of the NMA(s) in graph applications. This is similar to prior DAE approaches, where the Consumer (or Execute) is responsible for the computation required for the stored values. The difference in GraphAttack is that the Consumer also computes the address and stores data directly to memory rather than sending the value back to the Producer. This simplifies our design, as the Producer and Consumer do not have to synchronize or use invasive and costly components, e.g., CAMs, for store operations. However, this comes at the expense of a memory aliasing check (described next) to ensure the sliced program maintains the original semantics.

Figure 2 shows an example of dependency graph slicing with SSSP. The NMA (instruction 5) is decoupled into an **ISSUE**/**CONSUME** pair. Instruction 3 is duplicated; the remaining instructions are sliced based on the forward/backward slicing criteria described above. If the compiler is given a program where the slicing is not possible, e.g., if issuing an NMA depends on the result from another NMA, then the slicing procedure is terminated.

Memory Safety. To ensure the sliced program maintains the same behavior as the original program, the compiler performs a safety check to ensure that the Producer, Consumer, and the NMA(s)

do not have any memory conflicts. Specifically, the set of written addresses performed by different actors, i.e., the Producer, Consumer, and NMAs, must be disjoint. This is different from the parallel `for` loop notion of data-race freedom discussed earlier, as this check analyzes intra-loop iteration memory accesses. This check is required because decoupled architectures do not guarantee coherence among the actors, as the Producer and Consumer operate in different temporal planes in order for the Producer to gain runahead. Thus, a Producer could read a stale value if it has run far enough ahead of an NMA, or the Consumer, even on the same loop iteration. To implement this check, we use alias analysis, which can be aided by compiler annotations, e.g., `restrict`. Graph applications typically contain non-aliasing data-structures and do not perform complex pointer arithmetic, thus, in our experience, we did not find any applications that violate this check.

Since alias analysis is fundamentally imprecise, the GraphAttack compiler throws an exception that can be overwritten by the programmer if an address region can have overlapping accesses by the Producer and Consumer. In the dependency graph of the sliced SSSP kernel in Figure 2, the Consumer writes to `dist[v]` (node 10), and the NMA reads from `dist[neib_idx]` (node 5). The compiler is unable to validate that `v != neib_idx` and throws an exception. However, a programmer can overwrite this exception knowing that a vertex is disjoint from its neighbors to proceed with slicing. This is a common pattern in graph applications, but it is difficult to reliably resolve in a C-like front-end language. However, GraphAttack could be implemented as a backend to a graph DSL, e.g., GraphIt [66], in which case these access patterns are known to be safe.

Compilation Soundness. A successful GraphAttack program slicing efficiently maps to InO cores while preserving the DRF-inspired semantics presented in Section 3.1. Because all loop iterations must be independent, the Producer can runahead to another iteration while the Consumer continues to compute a prior iteration. The memory safety validation in the compiler ensures that the Producer, Consumer, and AAB access disjoint memory, avoiding the need for coherence between these components. Finally, the asynchronous update relaxed semantics occur when the Consumer writes a value, but the Producer has potentially runahead and issued loads to the same address. Thus, read-after-write dependencies are not always guaranteed. The implicit barrier at the end of the parallel loop waits for both the Producer and Consumer to arrive, providing synchronization.

We note that the GraphAttack compiler chain implements heuristics and is not guaranteed to detect NMAs or efficiently slice all graph applications. However, we found these heuristics sufficient for all graph applications we studied. Future work may aim to build a GraphAttack slicing backend into a graph DSL (e.g., [44, 58, 66]) that directly exposes graph vertices and neighbors.

4 GRAPHATTACK HARDWARE SUPPORT

GraphAttack augments pairs of simple *InO* cores with small buffers to retain the flexibility of a general-purpose multicore architecture while offering specialized data supply for graph kernels. Cores can be configured to perform do-all parallelism or decoupling as Producer/Consumer pairs. Each pair is equipped with novel hardware support for *asynchronous* atomic RMW instructions (discussed in Section 3), as well as asynchronous loads. Because only modest hardware additions are necessary for GraphAttack optimizations, InO cores are a practical and more hardware-efficient choice for data supply, enabling significant energy efficiency compared to prior latency tolerance approaches and greater flexibility compared to graph analytic accelerator designs.

4.1 Augmenting Multicore Architectures

Figure 5 presents GraphAttack hardware support that revisits and simplifies traditional DAE architectures [49]. Each Producer/Consumer pair uses two FIFO queues.

The Asynchronous Access Buffer (AAB). This buffer is responsible for tracking outstanding requests as they are served by the memory system. This is inspired by the DeSC [19] *Terminal Load*

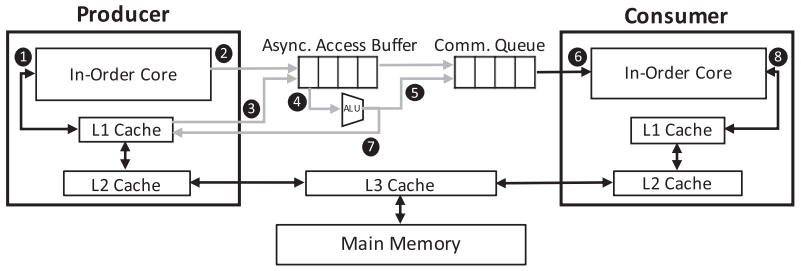

Fig. 5. GraphAttack augments parallel InO Producer/Consumer cores with modest hardware additions: the AAB, its ALU, and the communication queue. Gray arrows highlight the datapath for asynchronous accesses, e.g., RMWs.

Buffer that supports *terminal loads*, whose values are used only on the Compute. The AAB enables significant performance improvements; the compiler splits the dependency chains in graph applications by mapping an **ISSUE** instruction onto the Producer core to target the AAB and asynchronously issue an NMA (the dependent load or RMW). When the Producer encounters this instruction, it issues a memory request and reserves an entry in the AAB for the returned data. The Producer does not have to wait for the memory hierarchy to return this data because the AAB tracks it, so it continues with its next instruction, eliminating the stall that would typically occur in an InO core.

If the NMA misses in the cache, an MSHR is assigned for it, as normal. However, unlike a conventional core, this MSHR records the NMA's position in the AAB instead of a destination register to indicate where to place the data in the AAB once it returns from the memory hierarchy. Thus, the maximum number of outstanding asynchronous memory requests is the number of MSHR entries. When the Producer encounters a **PRODUCE** instruction that is preceded by a normal load, it enqueues the returned data in the AAB without issuing any asynchronous accesses.

The Communication Queue. This queue holds values of all *completed* memory accesses performed (normal loads), or issued (asynchronous), by the Producer to be retrieved by the Consumer in program order. To guarantee that asynchronous accesses maintain program order, the Communication Queue does not interface with the Producer, but only with the AAB as it is populated with data. For all accesses, when the head of the AAB contains data, it is passed over to the Communication Queue for the Consumer to use (via **CONSUME**).

A Modular Design. These queues have simple interfaces disjoint from the core model, as they are simply targeted via API calls. While minimizing core modifications, these modest hardware additions support data communication within each pair and can be implemented on a scratchpad, as seen in many accelerator-oriented systems [4, 21].

4.2 Hardware-Efficient Asynchronous RMWs

Our work innovates over prior decoupled architectures by introducing novel hardware support for *decoupled asynchronous RMWs*, that is, an RMW that can be *issued* asynchronously on one core and the result can be retrieved in another core at a later time. Figure 5 illustrates GraphAttack's proposed hardware additions. Specifically, the AAB is equipped with an ALU to perform arithmetic for atomic RMW operations. Unlike the Terminal Load Buffer in DeSC [19], the AAB supports asynchronous RMWs in addition to loads. This support is key to unlocking significant performance and energy-efficiency gains for parallel push-based implementations of graph processing algorithms.

Similar to asynchronous loads, the compiler statically determines asynchronous RMWs to have no dependents on the Producer slice. These correspond to the NMA(s) in graph applications. When

these accesses reach the memory stage, the AAB issues a request to the memory hierarchy for data and exclusive permissions to the cacheline accessed ❶. The addresses for these asynchronous access requests are then stored in the AAB ❷. As is typical of atomic operations, the memory system guarantees that the asynchronous RMW eventually acquires exclusive permissions to the data's cacheline. Upon receiving the permissions and data ❸, the L1 cache memory controller sends the data to the AAB. A dedicated ALU on the AAB performs arithmetic necessary for the atomic operation ❹. The result of the operation (e.g., a success indicator or a data value, depending on the semantics of the RMW operation) is simultaneously enqueued in the Communication Queue in issue order ❺ (to be retrieved by a corresponding **CONSUME** instruction ❻). Meanwhile, the atomic write of the newly computed data is performed to the memory hierarchy ❼. The RMW then gives up exclusive access to its cacheline. While this description uses an exclusive access mechanism to implement the RMW, we imagine that the RMWs could also be implemented using a simple logic unit that implement a loop over load-link/store-conditional operations, especially because these operations are supported open-source standards, e.g., RISC-V.

Unlike previous DAE approaches, e.g., [19], the Consumer may access memory, e.g., from computation dependent on the result of an NMA. Recall from Section 3 that the compiler ensures that there are no memory conflicts between the Producer, Consumer, and NMAs. Thus, Consumer memory accesses can be performed through the Consumer core's pipeline ❽. This requires no additional hardware or logic because they leverage the Consumer's private caches. Furthermore, because there are no memory conflicts between the Producer, Consumer, and NMAs (with the exception of asynchronous updates), there are no performance impacts due to cache coherence. For push-based graph algorithms, vertex property updates are entirely encapsulated by RMWs.

5 EVALUATION METHODOLOGY

Our compiler and hardware code as well as our benchmarks and simulator configurations are open-source[1,2,3] and are being developed as part of our full-system chip design project.

Compiler. We implement GraphAttack compilation techniques through Clang/LLVM passes. These passes have two different modes: (1) GraphAttack slicing or (2) traditional parallelism. Parallelism is provided via OpenMP and Producer/Consumer streams are mapped to pairs of OpenMP threads. An LLVM pass performs the slicing by searching for instructions of interest (NMAs) to use as anchors and for creating Producer/Consumer slices. The slices are further refined with LLVM's dead-code elimination pass, which identifies unneeded instructions after slicing.

Simulation Infrastructure. To evaluate GraphAttack, we use MosaicSim [30], a cycle-driven simulator designed for hardware-software co-design exploration and heterogeneous systems. The simulator is able to accurately model many performance bottlenecks of modern systems, including memory contention, e.g., cache invalidation costs when multiple threads access the same data. The simulator has been validated against an Intel Xeon processor and has demonstrated agile performance evaluation capabilities for early-stage design of heterogeneous systems [50]. To demonstrate GraphAttack hardware efficiency and flexibility, we employ a single-issue InO core modeled after Ariane, an open-source, InO CPU that is the base core model of emerging manycore chip designs [1, 64], as well as a quad-issue OoO Haswell-like core model. Table 3 compares their microarchitectural features. Each core has private L1 and L2 caches, and all cores share an LLC. The L1 cache also has a simple streaming prefetcher and 32 corresponding MSHR entries, which limits the maximum number of outstanding memory requests.

[1] https://github.com/amanocha/GraphAttack_Applications.
[2] https://github.com/PrincetonUniversity/GraphAttack.
[3] https://github.com/PrincetonUniversity/MosaicSim.

Table 3. Core Model Parameters

Parameter	Out-of-Order	In-Order
Issue Width	4	1
Instr. Window/ROB/LSQ	128/128/128	-
Branch Prediction	Gshare	Gshare
Frequency/Tech. Node	2GHz/22nm	2GHz/22nm
Core Area (mm^2)	8.44	1.01

Table 4. Memory and Queue (Shaded) Parameters

L1	32KB/private/8-way/4-cycle latency
MSHR	32 entries per L1 cache
L2	256KB/private/8-way/11-cycle latency
L3	2MB/shared/16-way/34-cycle latency
DRAM	DDR3L /68GB/s BW/100ns latency
Comm Queue	512 entries
Async. Acc. Buffer	128 entries
Queue Acc. Latency	3 cycles

Producer/Consumer pairs are modeled with *InO* cores, while the AAB and communication queue are modeled as FIFO queues with configurable sizes. Table 4 summarizes these memory hierarchy and queue configurations. We model core area and energy consumption using McPAT [27] and find the OoO core model to be approximately 8× larger than the simpler, InO Ariane cores. Using this ratio, we perform equal-area comparisons between multicore InO and OoO configurations. We use Cacti [34] for area and power modeling of the caches and the FIFO queues and find that the modest GraphAttack hardware additions add negligible (< 1%) area overhead. We use the VAMPIRE tool [18] to model DRAM power. These measurements allow us to quantitatively evaluate energy consumption using a dynamic energy model.

Applications. We evaluate GraphAttack on eight competitive graph application implementations, detailed in Table 6. We obtain the push- and pull-based variants of BFS, SSSP, and PR from GraphIt [66], GP as a direct CPU port of the Gunrock [58] GPU implementation, and SDHP from the Theano DNN framework [55].[4] GP and SDHP are used in emerging application domains, e.g., recommendation systems and sparse neural networks [15, 68]. While SDHP is used in sparse linear algebra rather than graph analytics, its use of a sparse matrix to filter a dense matrix incurs long-latency accesses similar to NMAs, making it amenable to GraphAttack innovations.

Push-based BFS, SSSP, and PR use the **`COMPARE_EXCH`**, **`MIN`**, and **`ADD`** RMW operations, respectively. Pull-based SSSP uses the relaxed asynchronous update semantics (discussed in Section 3). "Vertex-centric" describes applications that can be expressed as iterative computations where nodes are updated based on neighbor data; BFS, SSSP, and PR are vertex-centric. For all eight applications, GraphAttack compiler techniques successfully perform Producer/Consumer slicing.

Input Datasets. Graph application performance can highly depend on the input dataset [51, 66]. Table 5 summarizes the variety of inputs used to capture performance variability. Synthetic inputs (Kronecker networks [26]), follow a power-law distribution, similar to many real-world datasets. Social and web networks, e.g., Twitter and Wikipedia data, serve as concrete real-world inputs. For vertex-centric workloads, we utilize large graphs with millions of nodes to demonstrate that our technique scales to the modern network sizes. However, such large networks have massive data footprints (i.e., hundreds of GB) and the number of computations for these applications scales with the number of nodes in the network. In order to feasibly simulate these programs with such massive data footprints, we sample each network by simulating 20 million edge traversals within the densest and most representative epoch of each application/input combination. GP and SDHP do not require simulation sampling, as their computations scale with the number of edges and thus we utilize smaller datasets for these two applications. We use bipartite social networks and a synthetic power-law graph for GP and smaller Kronecker, social, and web networks for SDHP.

[4]Other frameworks, e.g., Scipy, densify the sparse matrix. This is not energy efficient as it performs many unneeded computations, however, it has high performance mappings to common vectorized hardware.

Table 5. Application Inputs and Properties

Application	Input	Nodes	Edges	Avg/Max Deg.
BFS, SSSP, PR	Kronecker (kr)	34M	1B	31/639K
	Twitter (tw)	53M	2B	37/780K
	Wikipedia (wk)	12M	378M	31/8K
	Sd1 Web (wb)	95M	2B	20/1M
GP (bipartite)	Power (po)	7K/21K	63751	9/6K
	Dbpedia (db)	46K/89K	144K	3/.2K
	Amazon (am)	60K/116K	165K	3/2K
	YouTube (yt)	30K/94K	300K	10/8K
SDHP	Kronecker (kr)	33K	883K	27/6K
	Sinkhorn (sh)	100K	173K	2/4K
	Amazon (am)	28K	80K	3/1K
	YouTube (yt)	30K	300K	10/8K

Table 6. We evaluate GraphAttack on 5 Applications; 3 (BFS, SSSP, PR) Have 2 Variants (push, pull), Yielding 8 Total

App	Description	Vertex Centric	Async Update	RMWs
BFS	Given starting vertex, find min. number of hops to all vertices.	Yes	No	Push only
SSSP	Given starting vertex, find min. distance (sum of edge weights) to all vertices.	Yes	Pull only	Push only
PR	Vertex scores flow to outgoing neighbors until all scores converge.	Yes	No	Push only
GP	Relate vertices in a bipartite graph partition with shared neighbors in the other partition.	No	No	No
SDHP	Compute elementwise multiplication of a sparse matrix and dense matrix.	No	No	No

Prior Latency Tolerance Approaches. We quantitatively compare GraphAttack to general-purpose latency tolerance approaches as well as graph-tailored prefetching.

Out-of-Order (OoO) Cores leverage instruction-level parallelism to overlap computation, e.g., arithmetic instructions, with memory accesses. Although they are the prominent core model utilized in modern multicore architectures, they are poorly utilized by graph applications and consequently incur area and energy overheads, as described in 2.2.

DeSC [19] is a state-of-the-art DAE architecture that leverages decoupled program execution to hide memory access latency. It utilizes a *Supply* core to exclusively handle memory accesses and address computations and a *Compute* core to perform other, e.g., arithmetic, computations. This work introduced *terminal loads*, i.e., loads whose values are used *only* on the Compute, to allow the Supply to runahead and supply data for the Compute. However, because graph applications exhibit control flow dependent on the NMAs, DeSC cannot slice along these bottleneck memory accesses. Furthermore, this approach proposes an area-inefficient hardware implementation that utilizes two OoO cores and CAMs to communicate data between the Supply and Compute.

DROPLET [11] is a state-of-the-art data-aware decoupled prefetcher for graph analytics. This approach advocates for memory hierarchy tailoring; it utilizes a streaming prefetcher in the L2 cache to prefetch streaming memory accesses, i.e., neighbor locations, whose data determine the addresses of NMAs. By prefetching neighbor locations, DROPLET aims to fetch the data for NMAs early and eliminate demand misses and stalling due to DRAM accesses. This prefetching is speculative, which can incur memory traffic and energy overheads; Section 7 further describes prefetching techniques.

6 RESULTS

This section demonstrates GraphAttack's ability to meet its performance goals through latency tolerance, scalability, energy efficiency, programmability, and portability.

6.1 GraphAttack Performance

A chip-to-chip comparison between InO and OoO cores would not be appropriate, as OoO cores have many components that can accelerate executions, e.g., through OoO execution, at the cost of higher area and energy. Thus, we perform an equal-area evaluation. Table 3 shows a conservative estimate that matches 8 InO cores to 1 OoO core. We use this as a basis to compare GraphAttack against various optimizations for latency tolerance and graph analytics on OoO cores. In our first evaluation, Figure 6 compares 2 OoO cores, 16 parallel InO cores, 2 OoO cores with DROPLET

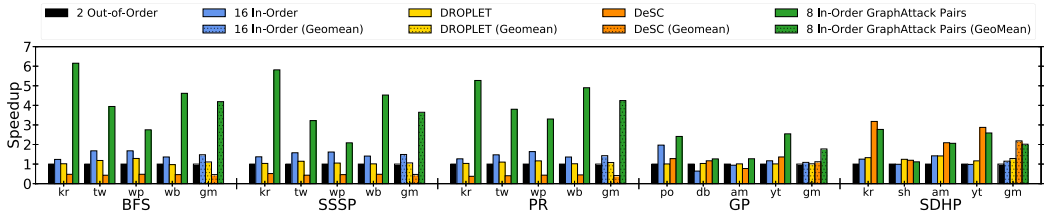

Fig. 6. Speedup comparisons between 2 parallel OoO cores, 16 parallel InO cores, 2 OoO with DROPLET prefetching, 1 OoO DeSC pair, and 8 InO GraphAttack pairs (ratios that equalize for area). GraphAttack significantly outperforms do-all parallelism, prefetching, and decoupling techniques on all graph-traversal apps, while DeSC has comparable performance gains on SDHP.

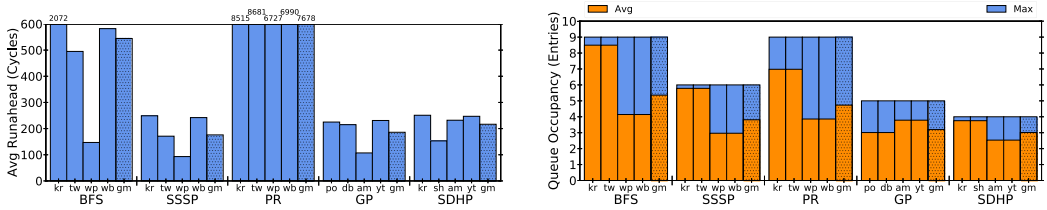

Fig. 7. LHS: GraphAttack achieves significant Producer runahead, with average runahead distances far greater than the DRAM latency of 200 cycles (100 ns). RHS: GraphAttack requires very small (<10 entries) AABs.

prefetching [11], DeSC (1 OoO pair), and 8 GraphAttack (InO) pairs for the push-based variants of vertex-centric applications BFS, SSSP, and PR, as well as GP and SDHP. Due to limited space, we opt to omit results for the pull-based variants; they have similar latency bottlenecks to push-based implementations and thus similar performance gains.

GraphAttack outperforms all prior techniques on vertex-centric applications, achieving up to a 6.16× speedup (BFS on Kron) with geomean 2.87× over 2 parallel OoO cores. It offers the greatest speedups on vertex-centric graph applications because they exhibit control flow dependent on the NMAs that troubles existing latency tolerance approaches. These speedups are even more notable on the synthetic Kronecker network; real-world networks exhibit more locality due to community structure, helping the performance of the baseline (parallel OoO cores). Despite tailoring its prefetchers to graph application access patterns, DROPLET faces challenges on many real-world graphs because they have a large number of low degree vertices (1–2 neighbors), which hurts the accuracy and timeliness of the L2 streaming prefetcher that aims to prefetch addresses of NMAs. Consequently, DROPLET offers limited performance gains that range from $0.97 \times -1.42\times$. While DeSC can identify the NMA and show performance gains in SDHP (no control flow dependency), it struggles on all vertex-centric applications , where control flow dependencies prevent the NMAs from being identified as terminal. GraphAttack, however, delivers performance gains over the baseline for *all* application/input combinations.

To highlight the substantial performance impact of enabling GraphAttack as a data supply optimization on top of a multicore architecture, we also compare it to do-all parallelism with 16 InO cores, where GraphAttack achieves up to a 2.71× speedup (geomean 2.16×). With targeted minimal hardware additions, GraphAttack is able to catapult *general-purpose* hardware, e.g., a 16 InO core system, to reap significant gains.

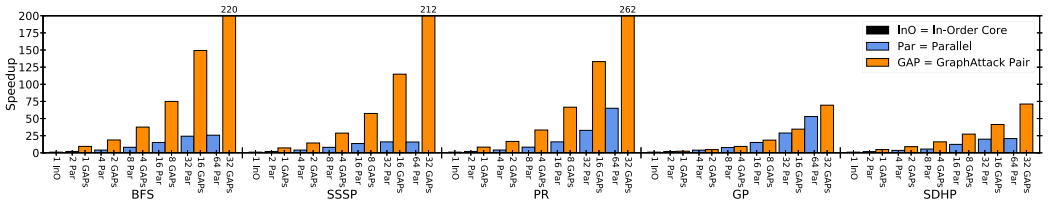

Fig. 8. Speedup comparisons between GraphAttack and traditional parallelism scaling from 1 to 64 cores on Kron and power law bipartite networks. GraphAttack performance improvements scale linearly, enabling scalability for a manycore architecture.

6.2 Exploiting Memory Level Parallelism

GraphAttack relies on MLP to mitigate memory access bottlenecks. MLP stems from Producer runahead, as long-latency memory requests are issued, overlapped, and completed early. The Producer and Consumer begin operating on the application at the same time, creating a brief "warm-up period" where the Consumer waits for the Producer's initial memory accesses to complete. As the Producer continues issuing long-latency memory access requests, it quickly runs ahead of the Consumer. The left-hand side of Figure 7 presents the average runahead distances of all applications and inputs. We measure runahead distance as the number of cycles between when a memory request is issued on the Producer and received on the Consumer. Most runahead distances are greater than the 200-cycle (or 100ns) DRAM latency, effectively removing the latency of NMA accesses to the Consumer. In PR, runahead becomes extreme because the Consumer performs floating point computation but even in this case, the AAB is never filled to capacity.

The right-hand side of Figure 7 presents the AAB's max and average queue occupancy (number of entries) for all applications and inputs. For each application, the max queue occupancy remains constant across inputs and is at most 9 entries. The size of the AAB can be small because the Producer only needs to runahead 200 cycles, which are incurred by 10–15 instructions that only span a few inner loop iterations. Thus, there are many more AAB (and MSHR) entries than necessary. We selected and modeled larger queue sizes as design decisions for future innovations, e.g., batching of memory requests that more aggressively use queues without incurring significant area overheads. The size of the Communication Queue does not impact the Consumer; if it is full, the Producer stalls. While this may lower Producer runahead, the Consumer can continue to read from the queue and only stalls when the queue is empty (waiting for data).

6.3 Scalability

Figure 8 demonstrates the scaling trends of two multicore systems: (1) traditional do-all parallelism and (2) GraphAttack. We measure the performance of each of our eight applications (normalized to that of a single InO core) on the idiomatic Kron (power-law bipartite for GP) input. Since inner loops have no inter-loop dependencies and this input has a power-law nature, traditional parallelism performance (light blue bars) scales nearly linearly from 1–32 cores. At 64 cores, it stops scaling on push-based BFS and SSSP due to synchronization overheads and load imbalances. Meanwhile, GraphAttack (orange bars) performance improvements scale up to 64 cores, with respect to both a single-core baseline and an equal-area do-all parallelism configuration. Because atomic operations are handled exclusively by either the Producer or Consumer, GraphAttack encounters fewer synchronization overheads than the baseline. Furthermore, GraphAttack exhibits less load imbalance as the data parallelism is halved from the perspective of any core, i.e., the data is parallelized amongst all Producer cores and amongst all Consumer cores in two separate groups.

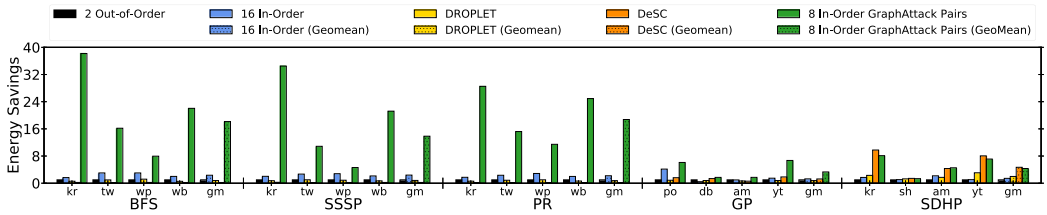

Fig. 9. Energy efficiency comparisons between 2 parallel OoO cores, 16 parallel InO cores, DROPLET prefetching (2 OoO), DeSC (1 OoO pair), and 8 GraphAttack (InO) pairs (ratios that equalize for area). With hardware-efficient innovations, GraphAttack achieves significant energy efficiency gains over all other techniques.

In a 32-core system, configuring the cores as GraphAttack (Producer/Consumer) pairs achieves up to a 7.18× speedup (geomean 2.2×) over do-all parallelism. Compared to a single GraphAttack pair, 32 pairs achieve near linear scaling, up to a 31.56× improvement. GraphAttack begins to saturate memory bandwidth with 64 cores (32 pairs) with *no* additional memory traffic. This demonstrates that our approach effectively transforms the latency bottleneck into a bandwidth bottleneck, which can be alleviated with advanced memory systems, such as **high-bandwidth memory** (**HBM**).

6.4 Energy Efficiency

Figure 9 presents an equal-area energy-efficiency comparison between the same configurations shown in Figure 6, measuring the energy efficiency normalized to that of 2 OoO cores. We measure energy efficiency as energy-delay, a product of energy consumption (described in Section 5) and runtime. GraphAttack achieves greater energy efficiency than all other latency tolerance configurations for *all* applications and inputs, with the exception of SDHP on Kron and YouTube. In these two cases, DeSC achieves significant runtime performance improvements (OoO execution coupled with efficient program decoupling) that outweigh the energy overheads of using OoO cores. Overall, GraphAttack achieves up to a 27× improvement (geomean 8.61×) in energy efficiency over 2 parallel OoO cores due to faster runtime performance and greater hardware efficiency. Given the 2.87× geomean runtime speedup, 6.3× of the energy savings (geomean) is due to core power savings alone. These improvements highlight the stark contrast between significant area and energy overheads of OoO cores and GraphAttack's modest hardware additions on top of simple, InO cores.

6.5 Flexible Application Support

We demonstrate GraphAttack's multi-phase application support on DO BFS, an optimized implementation that switches between pull- and push-based kernels per epoch depending on how many vertices are active [12]. Push-based BFS is more efficient in the early phases of the application when the frontier of active vertices is small, as parallel memory accesses are less likely to conflict. The pull-based variant is more performant once part of the graph has been traversed as vertex computation can break early if a visited parent is found. Unlike the traditional pull-based BFS we studied above, DO BFS applies an early break optimization in its pull phases to eliminate unneeded neighbor traversals. Unfortunately, this optimization creates a control flow dependency on the NMA that GraphAttack cannot efficiently decouple. Thus, we configure the cores to perform do-all parallelism for pull-based phases of DO BFS and enable GraphAttack during the push-based phases. Unlike specialized hardware tailored to push- or pull-based vertex-centric kernels, GraphAttack flexibly supports reconfigurability between decoupling and do-all parallelism on a per-phase basis.

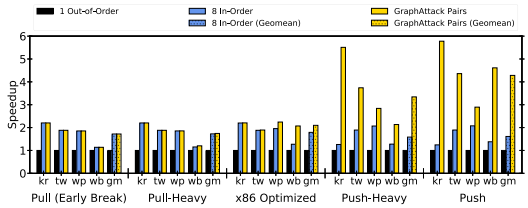

Fig. 10. Speedup comparisons between 1 OoO, 8 parallel InO, and GraphAttack (8 InO) for different direction-optimized BFS variants. Through its flexibility, GraphAttack optimizes the performance of all variants.

Figure 10 shows an equal-area speedup comparison between 1 OoO core, 8 parallel InO cores, and 4 GraphAttack pairs on DO BFS with varying amounts of pull- and push-based phases. GraphAttack consistently outperforms the OoO core and performs as well as or better than do-all parallelism due its flexibility. It offers particularly significant speedups, up to 5.78×, for the push-heavy and push-only variants, where data supply optimizations weigh heavily on the speedup. While an accelerator, e.g., Graphicionado [21], may excel at a particular configuration, e.g., pull-based, it simply cannot support other configurations without sacrificing programmability or specialization, making GraphAttack the most performant option across the board. There are scenarios where pull-based implementations are useful, e.g., when RMW contention is costly, and where push-based implementations are better, as they are more work-efficient. Unlike accelerators, GraphAttack utilizes the same hardware to handle all of these different implementations without requiring different custom modules or invasive modifications for each implementation.

GraphAttack's multi-phase application support can be extended to many other data analytic workloads, e.g., graph neural networks [67], where both data supply optimizations and traditional parallelism are key to optimal performance. Furthermore, the portability and space efficiency of GraphAttack make it a performant and optimal option for heterogeneous systems where it can be coupled with accelerators on the same chip. In cases where the ratio of accelerator communication costs to computation rate is high, parallel InO cores can swap in for accelerators. This flexibility makes GraphAttack a viable optimization for datacenter chips.

7 RELATED WORK

We cover the most closely related works in Section 2, including hardware prefetchers [6, 8, 11, 54], rigid accelerators designs [5, 21, 38, 43], and Decoupled Access/Execute [19, 20, 49]. Section 3 discusses decoupled software pipelining [48, 56]. Here, we discuss a wider range of works in the area.

Prefetching Prefetching is a broad technique that now spans the computing stack. Timeliness is a critical issue; early prefetches may be evicted from the cache before they are used or cause other useful data to be evicted, while late prefetches may lose latency benefits. Speculative prefetching also causes increased bandwidth pressure, limiting scalability in a multicore system. Software and programmable prefetchers [7, 8, 28] employ compiler passes to identify problematic (e.g., pointer indirect, random) memory accesses and schedule prefetch instructions accordingly. However, these passes struggle to schedule timely prefetches for graph applications, where control flow statement(s) lead to variability in execution time. Meanwhile, for appropriate applications, GraphAttack avoids these issues, as it retrieves the data with perfect precision and holds it in communication queues.

Data-aware hardware prefetchers [11, 63] reduce latency for pointer indirection, e.g., `A[B[i]]`. By using a streaming prefetcher for the sequential accesses, the addresses of NMAs can be determined early. However, in graph applications, vertices have varying degrees, which results in a

variable number of streaming accesses. This causes inaccurate prefetches that add bandwidth pressure. Additionally, these prefetches are designed for specific data-structures. GraphAttack flexibly supports our entire application suite (Table 6), which involves different data-structures that experience irregular accesses: CSC/CSR graphs, adjacency lists, and filtered dense matrixes.

Address-correlation prefetching approaches [59, 61] improve the latency of irregular accesses by storing correlated addresses using large metadata structures. However, these approaches are for applications that have repetitiveness in indirect memory accesses (e.g., repeated traversal of a linked list). These patterns do not often appear in graph applications, and thus, these approaches would have limited value in this domain.

Runahead, Slipstream, and Pipelined Architectures Prior work has explored techniques, which rely on runahead execution [16, 35, 40]. These approaches pre-execute application code and generate cache misses early to hide the long latency cost of future demand requests. However, these approaches are limited to short runahead intervals due to their speculative nature. In GraphAttack, Producer runahead is limited only by the size of efficient hardware buffers, and even with small buffers, it is able to achieve the long runaheads that are required by severely latency-bound graph applications. Continuous runahead [22] proposes runtime analysis hardware to dynamically identify dependence chains and pre-execute accordingly. Slipstream and look-ahead architectures [24, 47, 53] employ two execution streams where one stream skips non-essential computation and thus operates speculatively. However, graph applications have irregular, unpredictable control flow that impedes both NMA dependence chain identification and accurate speculation.

Lastly, Pipette [36] aims to improve core utilization for irregular applications by decoupling threads within a multithreaded core and making use of architecturally visible queues. This work shares the same goal as GraphAttack, but requires manual code transformations in order to split applications into pipeline stages and targets OoO cores, whereas GraphAttack automatically decouples graph applications along their NMAs for InO execution.

Specialized Memory Systems Prior techniques for graph processing aim to improve memory bandwidth efficiency through specialized engines within the memory hierarchy. Refs. [4, 32] leverage the power-law nature of real-world networks, and cache or reschedule worklist updates accordingly. Mukkara et al.[33] buffers and coalesces updates throughout the memory hierarchy to exploit temporal locality. Because GraphAttack is flexible and independent of the memory system, its latency improvements synergize with the bandwidth efficiency of these techniques and combining the different approaches would yield even greater performance. Buffets [45] is a storage idiom for orchestrating communication in accelerators, particularly those designed for dense workloads. It provides tools to manage the synchronization in a decoupled manner, similar to DAE. In contrast, GraphAttack focuses on accelerating graph applications on modern hardware composed of general-purpose cores.

Processing-in-Memory [5, 65, 69] leverage a PIM distributed architecture. While PIM has shown great promise, it requires invasive core and memory hierarchy modifications, which are difficult to implement in today's open-source hardware ecosystem. Additionally, they require program adaptation which only specific, e.g, vertex-centric kernels can utilize. GraphAttack, however, requires minimal hardware additions that complement the core models of existing architectures and flexibly support a rich range of applications.

8 CONCLUSION

This article presents a flexible hardware-software approach that targets memory latency bottlenecks in graph applications through compiler techniques and modest hardware additions. GraphAttack provides significant performance and energy efficiency gains due to its (1) innovative program slicing approach that automatically identifies and optimizes for long-latency NMAs

through data Producer/Consumer pairs and (2) small, shared AAB that enable simple, InO multicore architectures to overlap neighbor accesses and exploit MLP. By precisely targeting the latencies that critically bottleneck graph applications, our approach is the first to make InO data supply specialized, practical, and optimal for graph analytics while offering significant application flexibility. Thus, GraphAttack enables scalability to a manycore programmable accelerator or efficient graph computations performed at the edge. This is timely work for the New Golden Age of Computer Architecture [3], where significant efficiency gains are realized with vertical approaches, exemplified by GraphAttack.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful feedback.

REFERENCES

[1] 2019. RISC-V Ariane CVA6. Retrieved from on Aug. 2020 https://github.com/openhwgroup/cva6.
[2] 2020. HammerBlade. Retrieved from on Aug. 2020 https://github.com/bespoke-silicon-group/bsg_bladerunner.
[3] ACM. 2018. John Hennessy and David Patterson. Turing Award lecture. Retrieved on Aug 2019 from https://www.acm.org/hennessy-patterson-turing-lecture
[4] Abraham Addisie, Hiwot Kassa, Opeoluwa Matthews, and Valeria Bertacco. 2018. Heterogeneous memory subsystem for natural graph analytics. In *Proceedings of the International Symposium on Workload Characterization*. IEEE, 134–145.
[5] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the International Symposium on Computer Architecture*. ACM, 105–117.
[6] Sam Ainsworth and Timothy M. Jones. 2016. Graph prefetching using data structure knowledge. In *Proceedings of the International Conference on Supercomputing*. ACM.
[7] S. Ainsworth and T. M. Jones. 2017. Software prefetching for indirect memory accesses. In *Proceedings of the International Symposium on Code Generation and Optimization*. 305–317.
[8] Sam Ainsworth and Timothy M. Jones. 2018. An event-triggered programmable prefetcher for irregular workloads. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM.
[9] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. 2016. *The Rocket Chip Generator*. Technical Report. EECS Department, University of California, Berkeley.
[10] Jonathan Balkind, Michael McKeown, Yaosheng Fu, Tri Nguyen, Yanqi Zhou, Alexey Lavrov, Mohammad Shahrad, Adi Fuchs, Samuel Payne, Xiaohua Liang, Matthew Matl, and David Wentzlaff. 2016. OpenPiton: An open source manycore research framework. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 217–232.
[11] Abanti Basak, Shuangchen Li, Xing Hu, Sang Min Oh, Xinfeng Xie, Li Zhao, Xiaowei Jiang, and Yuan Xie. 2019. Analysis and optimization of the memory hierarchy for graph processing workloads. In *Proceedings of the International Symposium on High Performance Computer Architecture*. 373–386.
[12] Scott Beamer, Krste Asanović, and David Patterson. 2012. Direction-optimizing breadth-first search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 10 pages.
[13] Maciej Besta, Michał Podstawski, Linus Groner, Edgar Solomonik, and Torsten Hoefler. 2017. To push or to pull: On reducing communication and synchronization in graph computations. In *Proceedings of the High-Performance Parallel and Distributed Computing*. ACM, 93–104.
[14] Peter L. Bird, Alasdair Rawsthorne, and Nigel P. Topham. 1993. The effectiveness of decoupling. In *Proceedings of the International Conference on Supercomputing*.
[15] Beidi Chen, Tharun Medini, and Anshumali Shrivastava. 2019. SLIDE : In defense of smart algorithms over hardware acceleration for large-scale deep learning systems. In *Proceedings of the 3rd MLSys Conference*. ACM.
[16] James Dundas and Trevor Mudge. 1997. Improving data cache performance by pre-executing instructions under a cache miss. In *Proceedings of the International Conference on Supercomputing*. ACM, 68–75.
[17] Adi Fuchs and David Wentzlaff. 2019. The accelerator wall: Limits of chip specialization. In *Proceedings of the International Symposium on High Performance Computer Architecture*. 1–14.

[18] Saugata Ghose, Abdullah Giray Yaglikçi, Raghav Gupta, Donghyuk Lee, Kais Kudrolli, William X. Liu, Hasan Hassan, Kevin K. Chang, Niladrish Chatterjee, Aditya Agrawal, Mike O. Connor, and Onur Mutlu. 2018. What your DRAM power models are not telling you: Lessons from a detailed experimental study. In *Proceedings of the ACM on the Measurement and Analysis of Computing Systems*. Vol. 2 (2018), 38.

[19] Tae Jun Ham, Juan L. Aragón, and Margaret Martonosi. 2015. DeSC: Decoupled supply-compute communication management for heterogeneous architectures. In *Proceedings of the International Symposium on Microarchitecture*. ACM.

[20] Tae Jun Ham, Juan L. Aragón, and Margaret Martonosi. 2017. Decoupling data supply from computation for latency-tolerant communication in heterogeneous architectures. *ACM Transactions on Architecture and Code Optimization* 14, 2 (June 2017), Article 16, 1–27.

[21] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. 2016. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *Proceedings of the International Symposium on Microarchitecture*. IEEE. 13 pages.

[22] Milad Hashemi, Onur Mutlu, and Yale N. Patt. 2016. Continuous runahead: Transparent hardware acceleration for memory intensive workloads. In *Proceedings of the International Symposium on Microarchitecture*. 1–12.

[23] Matt J. Keeling and Ken T. D. Eames. 2005. Networks and epidemic models. *Journal of The Royal Society Interface* 2, 4 (June 2005), 295–307.

[24] Sushant Kondguli and Michael Huang. 2019. R3-DLA (reduce, reuse, recycle): A more efficient approach to decoupled look-ahead architectures. In *Proceedings of the International Symposium on High Performance Computer Architecture*. IEEE.

[25] Ioannis Konstas, Vassilios Stathopoulos, and Joemon M. Jose. 2009. On social networks and collaborative recommendation. In *Proceedings of the International Conference on Research and Development in Information Retrieval*. ACM, 195–202.

[26] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. 2010. Kronecker graphs: An approach to modeling networks. *Journal of Machine Learning Reseach* 11 (March 2010), 985–1042.

[27] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the International Symposium on Microarchitecture*. ACM, 469–480.

[28] Chi-Keung Luk and Todd C. Mowry. 1996. Compiler-based prefetching for recursive data structures. *SIGPLAN Notices* 31, 9 (Sept. 1996), 222–233.

[29] Yatin A. Manerkar, Daniel Lustig, Margaret Martonosi, and Michael Pellauer. 2017. RTLCheck: Verifying the nemory consistency of RTL designs. In *Proceedings of the International Symposium on Microarchitecture*. ACM, 463–476.

[30] Opeoluwa Matthews, Aninda Manocha, Davide Giri, Marcelo Orenes Vera, Esin Tureci, Tyler Sorensen, Tae Jun Ham, Juan L. Aragón, Luca Carloni, and Margaret Martonosi. 2020. MosaicSim: A lightweight, modular simulator for heterogeneous systems. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*.

[31] Milad Mohammadi, Tor M. Aamodt, and William J. Dally. 2017. CG-OoO: Energy-efficient coarse-grain out-of-order execution near in-order energy with near out-of-order performance. *ACM Transactions on Architecture and Code Optimization* 14, 4 (2017), 1–26.

[32] Anurag Mukkara, Nathan Beckmann, Maleen Abeydeera, Xiaosong Ma, and Daniel Sanchez. 2018. Exploiting locality in graph analytics through hardware-accelerated traversal scheduling. In *Proceedings of the International Symposium on Microarchitecture*.

[33] Anurag Mukkara, Nathan Beckmann, and Daniel Sanchez. 2019. PHI: Architectural support for synchronization- and bandwidth-efficient commutative scatter updates. In *Proceedings of the International Symposium on Microarchitecture*.

[34] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P. Jouppi. 2009. CACTI 6.0: A tool to model large caches. *HP Labs* 27 (2009), 28.

[35] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N. Patt. 2003. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Proceedings of the International Symposium on High-Performance Computer Architecture*. IEEE, 129.

[36] Quan M. Nguyen and Ddaniel Sanchez. 2020. Pipette: Improving core utilization on irregular applications through intra-core pipeline parallelism. In *Proceedings of the 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture*.

[37] Tony Nowatzki, Vinary Gangadhan, Karthikeyan Sankaralingam, and Greg Wright. 2016. Pushing the limits of accelerator efficiency while retaining programmability. In *Proceedings of the International Symposium on High Performance Computer Architecture*. 27–39.

[38] Eriko Nurvitadhi, Gabriel Weisz, Yu Wang, Skand Hurkat, Marie Nguyen, James C. Hoe, José F. Martínez, and Carlos Guestrin. 2014. GraphGen: An FPGA framework for vertex-centric graph computation. In *Proceedings of the International Symposium on Field-Programmable Custom Computing Machines*.

[39] Molly A. O'Neil and Martin Burtscher. 2014. Microarchitectural performance characterization of irregular GPU kernels. In *Proceedings of the International Symposium on Workload Characterization*. IEEE, 130–139.

[40] Onur Mutlu, Hyesoon Kim, and Y. N. Patt. 2005. Techniques for efficient processing in runahead execution engines. In *Proceedings of the International Symposium on Computer Architecture*. 370–381.

[41] OpenMP Architecture Review Board. 2018. OpenMP application program interface Version 5.0. Retrieved on Aug. 2019 from https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf.

[42] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. 2005. Automatic thread extraction with decoupled software pipelining. In *Proceedings of the 38th International Symposium on Microarchitecture*. IEEE, 107–118.

[43] Muhammet Mustafa Ozdal, Serif Yesil, Taemin Kim, Andrey Ayupov, John Greth, Steven Burns, and Ozcan Ozturk. 2016. Energy efficient architecture for graph analytics accelerators. In *Proceedings of the International Symposium on Computer Architecture*. IEEE, 166–177.

[44] Sreepathi Pai and Keshav Pingali. 2016. A compiler for throughput optimization of graph algorithms on GPUs. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 1–19.

[45] Michael Pellauer, Yakun Sophia Shao, Jason Clemons, Neal Crago, Kartik Hegde, Rangharajan Venkatesan, Stephen W. Keckler, Christopher W. Fletcher, and Joel Emer. 2019. Buffets: An efficient and composable storage idiom for explicit decoupled data orchestration. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 137–151.

[46] Clara Pizzuti. 2008. GA-Net: A genetic algorithm for community detection in social networks. In *Proceedings of the Parallel Problem Solving from Nature*, Günter Rudolph, Thomas Jansen, Nicola Beume, Simon Lucas, and Carlo Poloni (Eds.). Springer, Berlin, 1081–1090.

[47] Zach Purser, Karthik Sundaramoorthy, and Eric Rotenberg. 2000. A study of slipstream processors. In *Proceedings of the International Symposium on Microarchitecture*. 269–280.

[48] Ram Rangan, Neil Vachharajani, Manish Vachharajani, and David I. August. 2004. Decoupled software pipelining with the synchronization array. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*. IEEE, 177–188.

[49] James E. Smith. 1982. Decoupled access/execute computer architectures. In *ACM SIGARCH Computer Architecture News* 10, 3 (April 1982), 112–119.

[50] Tyler Sorensen, Aninda Manocha, Esin Tureci, Marcelo Orenes-Vera, Juan L. Aragón, and Margaret Martonosi. 2020. A simulator and compiler framework for agile hardware-software co-design evaluation and exploration. In *Proceedings of the International Conference On Computer Aided Design*. 1–9.

[51] Tyler Sorensen, Sreepathi Pai, and Alastair F. Donaldson. 2019. One size doesn't fit all: Quantifying performance portability of graph applications on GPUs. In *Proceedings of the International Symposium on Workload Characterization*. IEEE.

[52] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R. Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. 2015. GraphMat: High performance graph analytics made productive. *Proceedings of the VLDB Endowment* 8, 11 (July 2015), 1214–1225.

[53] Karthik Sundaramoorthy, Zach Purser, and Eric Rotenburg. 2000. Slipstream processors: Improving both performance and fault tolerance. *SIGOPS Operating Systems Review* 34, 5 (Nov. 2000), 257–268.

[54] Nishil Talati, Kyle May, Armand Behroozi, Yichen Yang, Kuba Kaszyk, Christos Vasiladiotis, Tarunesh Verma, Lu Li, Brandon Nguyen, Jiawen Sun, John Morton, Agreen Ahmadi, Todd Austin, Michael O'Boyle, S. Mahlke, Trevor Mudge, and Ronald Drelinski. 2021. Prodigy: Improving the memory latency of data-indirect irregular workloads using hardware-software co-design. In *Proceedings of the International Symposium on High Performance Computer Architecture*.

[55] Theano Development Team. 2016. Theano: A python framework for fast computation of mathematical expressions. arXiv:1605.02688. Retrieved from https://arxiv.org/abs/1605.02688.

[56] Neil Vachharajani, Ram Rangan, Easwaran Raman, Matthew J. Bridges, Guilherme Ottoni, and David I. August. 2007. Speculative decoupled software pipelining. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*. IEEE.

[57] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. 2010. Conservation cores: Reducing the energy of mature computations. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*.

[58] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. 2016. Gunrock: A high-performance graph processing library on the GPU. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*. ACM, 12 pages.

[59] Thomas F. Wenisch, Michael Ferdman, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. 2009. Practical off-chip meta-data for temporal memory streaming. In *Proceedings of the International Symposium on High Performance Computer Architecture*. IEEE, 79–90.

[60] Joyce Jiyoung Whang, Andrew Lenharth, Inderjit S. Dhillon, and Keshav Pingali. 2015. Scalable data-driven PageRank: Algorithms, system issues, and lessons learned. In *Proceedings of the European Conference on Parallel Processing*. 438–450.

[61] Hao Wu, Krishnendra Nathella, Dam Sunwoo, Akanksha Jain, and Calvin Lin. 2019. Efficient metadata management for irregular data prefetching. In *Proceedings of the International Symposium on Computer Architecture*. ACM, 449–461.

[62] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2018. How powerful are graph neural networks? In *International Conference on Learning Representations (ICLR)*. OpenReview.net.

[63] Xiangyao Yu, Christopher J. Hughes, Nadathur Satish, and Srinivas Devadas. 2015. IMP: Indirect memory prefetcher. In *Proceedings of the International Symposium on Microarchitecture*. 178–190.

[64] Florian Zaruba and Luca Benini. 2018. Ariane: An open-source 64-bit RISC-V application class processor and latest improvements. Technical talk at the RISC-V Workshop, Retrieved on from https://www.youtube.com/watch?v=8HpvRNh0ux4.

[65] Mingxing Zhang, Youwei Zhuo, Chao Wang, Mingyu Gao, Yongwei Wu, Kang Chen, Christos Kozyrakis, and Xuehai Qian. 2018. GraphP: Reducing communication for PIM-based graph processing with efficient data partition. In *Proceedings of the International Symposium on High Performance Computer Architecture*. 544–557.

[66] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2018. GraphIt: A high-performance graph DSL. In *Proceedings of the ACM on Programming Languages* OOPSLA (2018).

[67] Jie Zhou, Ganqu Cui, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, and Maosong Sun. 2018. Graph neural networks: A review of methods and applications. arXiv:1812.08434. Retrieved from https://arxiv.org/abs/1812.08434.

[68] Tao Zhou, Jie Ren, Matú Medo, and Yi-Cheng Zhang. 2007. Bipartite network projection and personal recommendation. *Physical Review. E, Statistical, Nonlinear, and Soft Matter Physics* 76, 4 (Oct. 2007).

[69] Youwei Zhuo, Chao Wang, Mingxing Zhang, Rui Wang, Dimin Niu, Yanzhi Wang, and Xuehai Qian. 2019. GraphQ: Scalable PIM-based graph processing. In *Proceedings of the International Symposium on Microarchitecture*. ACM, 712–725.

Received March 2021; revised June 2021; accepted June 2021