Hardware-Accelerated Hypergraph Processing with Chain-Driven Scheduling

Qinggang Wang, Long Zheng, Jingrui Yuan, Yu Huang, Pengcheng Yao, Chuangyi Gui, Ao Hu, Xiaofei Liao, Hai Jin
National Engineering Research Center for Big Data Technology and System,
Service Computing Technology and System Lab, Cluster and Grid Computing Lab,
School of Computer Science and Technology, Huazhong University of Science and Technology, China
{*qgwang, longzh, jryuan, yuh, pcyao, chygui, ahu, xfliao, hjin*}*@hust.edu.cn*

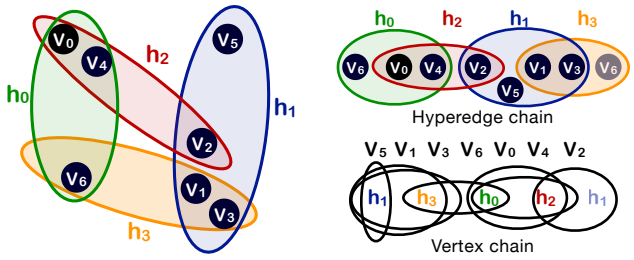
***Abstract*—Beyond ordinary graphs, hypergraphs are a graph representation to flexibly express complex multilateral relationships between entities. Hypergraph processing can be used to solve many real-world problems, e.g., machine learning, VLSI design, and image retrieval. Existing hypergraph processing systems handle a hypergraph in order of its hyperedge and vertex indices. This makes processing hypergraphs on general-purpose architectures suffer significantly from excessive off-chip memory accesses, most of which however are redundant in frequently accessing *overlapped* hyperedges and vertices, but the index-ordered scheduling destroys this potential locality.**

In this paper, we propose a novel *Generate-Load-Apply* (GLA) execution model to improve locality in hypergraph processing. The key insight of GLA is to use a concept of *chain* to characterize the overlapped feature of a hypergraph, exposing data reuse opportunities missed in existing hypergraph systems. The precondition of driving GLA model is to generate expected chains on the fly, but the software solution is so expensive that its overheads may outweigh the benefits achieved from the chain-driven scheduling. We further present ChGraph, the first hardware-accelerated hypergraph processing engine near each core. ChGraph is specialized in accelerating the chain generation and the chain-guided data loading (to hide memory access latency) while the general-purpose cores are responsible only for handling the apply operations of GLA. We evaluate ChGraph against a state-of-the-art hypergraph processing system Hygra on six hypergraph algorithms using five large real-world hypergraphs. Results on a simulated 16-core system show that ChGraph reduces the number of off-chip memory accesses by up to 4.56× and achieves up to 4.73× speedup while introducing only 0.26% area overhead.

***Keywords*-hypergraph processing; locality; chain; overlapped feature; hardware-accelerated; scheduling;**

I. Introduction

Ordinary graphs are widely used to represent pairwise relationships between entities, where entities are represented by vertices and the relationship between two entities is shown as an edge [39], [42], [52]. However, more complex multilateral relationships, involving rich interactions among multiple entities, are poorly expressible in ordinary graphs [24]. A more generalized graph model, called *hypergraph* [6], emerges to address this problem. A hypergraph contains a set of vertices and hyperedges, in which each hyperedge is allowed to connect an arbitrary number of vertices so as to capture multilateral relationships with superiority of flexible expression and abstraction [15], [28], [32]. Hypergraphs have been widely used in many fields, such as machine learning [20], [47], VLSI design [7], [48], and image retrieval [3], [8].



(a) An example hypergraph (b) The overlap-inducing chains

Figure 1. An illustrative example for hypergraph processing. (a) An example hypergraph with 7 vertices and 4 hyperedges; and (b) The overlap-inducing hyperedge and vertex chains. The hyperedge chain contains a chain of hyperedges, i.e., $\langle h_0, h_2, h_1, h_3 \rangle$. The vertex chain is similar as $\langle v_5, v_1, v_3, v_6, v_0, v_4, v_2 \rangle$.

A typical hypergraph example [11] is an author collaboration network. Using the ordinary graph, the key information on the co-authors of a paper may be lost suppose vertices represent authors and each edge indicates the existence of a paper co-authored by two authors [17]. Moreover, consider an application that measures the scholarly impact for each author by running a PageRank-like algorithm [44]. Analysis on this ordinary graph may also cause high-scoring authors to make the same impact on their co-authors (regardless of the quality of co-authored papers), such that the analysis results can be inaccurate. Instead, a hypergraph enables expressing these complex semantics naturally, in which authors are vertices (e.g., $v_0, \ldots, v_6$ in Figure 1(a)) and their co-authored papers can be considered as hyperedges (e.g., $h_0, \ldots, h_3$). We would like to particularly note that the ordinary graph is a special case of the hypergraph.

With the advent of numerous hypergraph applications, there has a growing interest in hypergraph analytics [30], [32], [44]. Hypergraph processing is an iterative procedure, in which the states of hyperedges and vertices are computed alternatively in each iteration [24], [41]. Recently, a few software frameworks have been developed to enhance hypergraph processing by improving its parallelism [41], load imbalance [24], communication overheads [15], and programming productivity [18], [22]. These earlier hypergraph systems typically handle a hypergraph in a strict order of its hyperedge and vertex indices. This may cause high cache

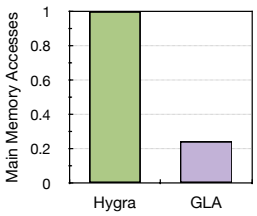


Figure 2. GLA reduces the total number of main memory accesses by 4.09× over a state-of-the-art hypergraph system Hygra [41] for PageRank on *Web-trackers*.

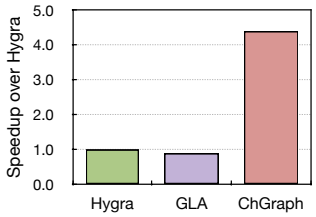


Figure 3. GLA runs 1.14× slower than Hygra [41] for PageRank on *Web-trackers*, but ChGraph reverses this situation with 4.39× performance improvement over Hygra.

misses with poor locality. Let us consider the example in Figure 1. Processing h_0 and h_2 in Figure 1(a) both need to access v_0 and v_4. Assume the cache is sized of 4. After processing h_0, h_1 is next processed and thus v_0 and v_4 have to be replaced out of cache. This makes the follow-up processing of h_2 cache-missed, resulting in off-chip main memory accesses. Further, the cache interference and limited memory bandwidth may worsen this situation. In this paper, we focus on investigating whether and how to exploit this kind of missed locality in hypergraph processing.

We observe that the underlying locality in hypergraph processing is closely related to the upper *overlapped* feature of hypergraphs, in the sense that the majority of vertices (hyperedges) can be shared by at least two hyperedges (vertices) as discussed in §II-D. In this research, we present a concept of **chain** to characterize the overlap feature of a hypergraph, exposing overlap-inducing locality easily. Figure 1(b) shows the two chains for hyperedges and vertices, respectively. Taking the hyperedge chain as an example, we can explicitly identify overlapped vertices of hyperedges, allowing to exploit the locality of hyperedge processing more easily. For example, the two hyperedges h_0 and h_2 both need to access v_0 and v_4, which can be reused with cache hits by processing h_2 closely behind h_0. In this context, the hyperedge order in the chain can naturally capture a locality-aware execution sequence.

In this paper, we propose a chain-driven *Generate-Load-Apply* (GLA) execution model to improve locality in hypergraph processing by exploiting fully overlapped opportunities of a hypergraph. In each iteration, the GLA first generates overlap-inducing hyperedge and vertex chains (as in Figure 1(b)). Then, these hyperedges (vertices) in the hyperedge (vertex) chain are scheduled along with their order in this chain and further load their associated incident vertices (hyperedges). Finally, the apply operation is invoked to update their value. In this way, overlapping data can be naturally reused by any two successive hyperedges (vertices) with fewer off-chip main memory accesses. Figure 2 shows the benefits of applying GLA against a state-of-the-art index-ordered hypergraph processing system Hygra [41] for the PageRank algorithm on the *Web-trackers* hypergraph [26], reducing 4.09× main memory accesses.

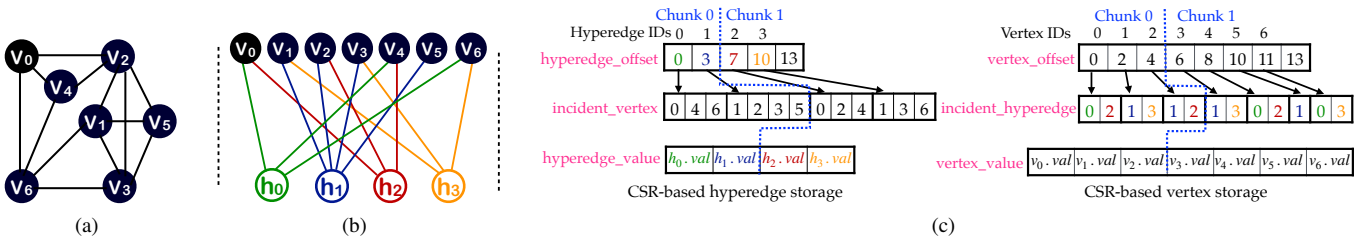
However, as in Figure 3, the software-based GLA solution does not improve overall performance. The reasons behind are twofold: First, the overlapped hypergraph data is activated dynamically and not necessarily used in every iteration. Thus, this active information can only be captured at runtime and the expected chains have to be generated online. Second, the core of chain generation is to find a maximally-overlapped successor but it introduces expensive sorting overheads that may outweigh the benefits achieved by the chain-driven idea.

We further propose **ChGraph**, the first hardware accelerated hypergraph processing engine, to reverse this situation (with a speedup of 4.39× against Hygra for PageRank on *Web-trackers*). Inspired by the *decoupled access-execute* (DAE) design philosophy [43], we architect ChGraph near each general-purpose core. Each ChGraph core runs ahead to generate the overlap-inducing hyperedge and vertex chains dynamically in a pipelined manner and to prefetch the relevant data on-the-fly along the chains. Once the requisite data is prepared, the general-purpose core is invoked to apply the updates. This design allows running a hypergraph algorithm in a neat way that ChGraph focuses only on the data preparing work while general-purpose cores are responsible for computation. In particular, unlike the standard access processor in a DAE architecture used for managing all memory accesses, ChGraph handles only read accesses to hypergraph data. This makes ChGraph itself cheap and also simplifies the overall system design. The unidirectional communication from ChGraph to the core is achieved through a first-in-first-out buffer, enabling parallel and overlapped execution between them.

HATS [34] proposes to leverage hardware acceleration to improve the scheduling order of graph applications effectively. However, HATS is not a drop-in replacement for ChGraph since it is inadequate to solve challenges in hypergraph processing. Consider the conventional graph is a special version of the hypergraph. ChGraph is more general than HATS to handle not only graph applications but also hypergraph applications (as discussed in §II-C).

This paper has the following key contributions:

- We characterize locality-oblivious memory accesses in hypergraph processing and reveal the intrinsic connection between the underlying locality of caches and the upper overlapping feature of hypergraphs.
- We propose *chain* to characterize the overlapped feature of a hypergraph, enabling exploiting locality in hypergraph processing easily by a novel chain-driven *Generate-Load-Apply* (GLA) execution model proposed further.
- We architect ChGraph, the first hardware accelerated hypergraph engine near each general-purpose core, to accelerate GLA model effectively and efficiently.
- We prototype ChGraph in RTL. Results on a simulated 16-core system show that ChGraph outperforms Hygra



(a) (b) (c)

Figure 4. The hypergraph representation and its storage format for the hypergraph in Figure 1. (a) The clique-expanded representation, (b) The bipartite representation, (c) The CSR-based bipartite graph storage for hyperedges and vertices

by up to 4.56× reduced main memory accesses and by up to 4.73× performance improvement. ChGraph needs just 0.094mm^2 and 61mW at 65nm process, taking only 0.26% area and 0.19% power overhead against a general-purpose core.

The rest of this paper is organized as follows. §II explains the background and motivation. Our approach is overviewed in §III. §IV presents the GLA execution model and ChGraph is elaborated in §V. §VI discusses experimental results. §VII surveys related work and §VIII finally concludes the paper.

II. Background and Motivation

We first review the preliminaries of hypergraph processing. We then investigate its locality-oblivious memory access behaviors, finally motivating our chain-driven approach.

A. Preliminaries

Hypergraph. A hypergraph $G = \langle V, H \rangle$ consists of a set of vertices and hyperedges. Each hyperedge h can connect an arbitrary number of vertices. $|V|$ ($|H|$) represents the number of vertices (hyperedges) in G. The degree of a hyperedge h (a vertex v) is the number of incident vertices (hyperedges) (i.e., the ones connected to h (v)), denoted as $deg(h)$ ($deg(v)$). $N(h)$ indicates the incident vertex set of h. $N(v)$ denotes the incident hyperedge set of v. Every vertex (hyperedge) can also be associated with some attributes, denoted as $v.val$ ($h.val$). In particular, two hyperedges h_i and h_j are called *overlapped* if they share at least one vertex (i.e., $N(h_i) \cap N(h_j) \neq \varnothing$). Similarly, we say two vertices are overlapped if they are connected by any hyperedges. For a directed hypergraph, the incident vertices of a directed hyperedge can be divided into a source vertex set and a destination vertex set. In this paper, a hypergraph is considered undirected if no special instructions.

Figure 1(a) depicts a hypergraph with 7 circled vertices and 4 ellipse hyperedges. Since h_0 contains the incident vertices v_0, v_4, and v_6, its degree is 3, i.e., $deg(h_0) = 3$. Meanwhile, since v_0 is contained in both h_0 and h_2, its degree is 2, i.e., $deg(v_0) = 2$. h_0 and h_2 are overlapped since $N(h_0) \cap N(h_2)$ (i.e., $\{v_0, v_4\}$) is not empty.

Hypergraph Representation. There are two classic hypergraph representations: *clique-expanded representation* [24] and *bipartite representation* [41]. The former represents each hyperedge as a clique among its incident vertices.

Algorithm 1 Hypergraph Processing Procedure

Input: $G\langle V, H\rangle$ – Hypergraph
HF – Hyperedge value update function
VF – Vertex value update function
Output: `vertex_value` and `hyperedge_value`

1: VertexInit(`vertex_value`)
2: HyperedgeInit(`hyperedge_value`)
3: ActiveVertexSet $FrontierV$.Setup( )
4: **while** $cur\_iteration\_num < iteration\_num_{max}$ **do** ▷ Hyperedge Computation
5: /*Active vertices update their incident hyperedges using HF*/
6: ActiveHyperedgeSet hs = VertexPro(G, $FrontierV$, HF)
7: ActiveHyperedgeSet $FrontierE$ = hs
8: $FrontierE$.isEmpty( ) /*Loop until frontier is empty*/ ▷ Vertex Computation
9: /*Active hyperedges update their incident vertices using VF*/
10: ActiveVertexSet vs = HyperedgePro(G, $FrontierE$, VF)
11: $FrontierV$ = vs
12: $FrontierV$.isEmpty( ) /*Loop until frontier is empty*/
13: $cur\_iteration\_num$++
14: **end while**
/*Processing the bipartite edge $\langle v_{id}, h_{id}\rangle$ to calculate the influence of the value of v_{id} on that of h_{id} */
15: **procedure** HF(v_{id}, h_{id})
16: `hyperedge_value`$[h_{id}]$ += $\frac{\texttt{vertex\_value}[v_{id}]}{v_{id}.getOutDegree()}$
17: **end procedure**
/*Processing the bipartite edge $\langle h_{id}, v_{id}\rangle$ to calculate the influence of the value of h_{id} on that of v_{id} */
18: **procedure** VF(h_{id}, v_{id})
19: $addend = \frac{1-\alpha}{|V| \times deg(v_{id})}$
20: `vertex_value`$[v_{id}]$ += $addend + \alpha \times \frac{\texttt{hyperedge\_value}[h_{id}]}{h_{id}.getOutDegree()}$
21: **end procedure**

Figure 4(a) shows the clique-expanded representation for the hypergraph in Figure 1(a). As the number of vertices increases, there will have a sharp increase in the number of edges for the clique-expanded representation, further leading to high storage overhead. Existing hypergraph systems usually adopt the latter, which represents each hyperedge as a distinctive vertex connecting its incident vertices. The bipartite representation for the hypergraph in Figure 1(a) is shown in Figure 4(b), where a solid circle ● indicates a vertice while a hollow circle ○ shows a hyperedge.

In the bipartite representation, a bipartite edge (i.e., the colored solid line in Figure 4(b)) is stored based on the *compressed sparse row* (CSR) format for space saving. Figure 4(c) shows the CSR-based bipartite storage for hyperedges and vertices, respectively. Let us consider the hyperedge CSR format (the vertex CSR format is similar). It has three arrays: `hyperedge_offset`, `incident_vertex`, and `hyperedge_value`. The `incident_vertex` array stores the

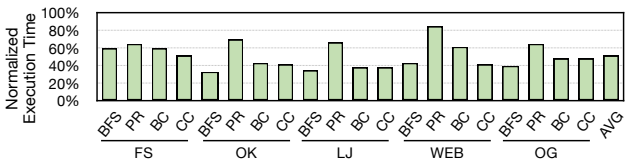


Figure 5. Normalized execution time of main memory accesses by running Hygra [41]. The trials run on a server configured with 2×14-core Intel Xeon E5-2680v4 CPUs (with each core containing 32KB L1 cache and 256 KB L2 cache, and sharing 35MB L3 cache) and a 64GB 4-channel DDR4.

incident vertices of hyperedges. The `hyperedge_offset` array maintains the offset information that indicates the incident vertex scope in the `incident_vertex` array for each hyperedge. The attribute of a hyperedge is stored in the `hyperedge_value` array.

Hypergraph Processing. Algorithm 1 shows a basic procedure of hypergraph processing, which relies on an iterative process with two basic kennels: *hyperedge computation* and *vertex computation* [18], [24], [41]. First, the vertex and hyperedge attributes are initialized (Lines 1-2) and active vertices are set (Line 3). The set of active vertices and hyperedges are denoted as $FrontierV$ and $FrontierE$, respectively. In each iteration, hyperedge computation starts first. In this period, every active vertex will be processed to update its incident hyperedges via an algorithm-specific *hyperedge update function* (HF) (Lines 6-7). Note that the `VertexPro` function, widely used in existing software frameworks [18], [41], iterates all active vertices in the index-ordered sequence. This makes it easy to understand and write hypergraph applications [41]. After all active vertices are finished, vertex computation is launched. It computes and updates vertex values using a *vertex update function* (VF) based on the newly activated hyperedges (Lines 10-11). The procedure is repeated until no active hyperedges and vertices are left (Lines 8 and 12) or a given maximum iteration count is reached (Line 4). The HF and VF functions in Lines 15-21 exactly show an implementation for PageRank, where the HF computes the influence of an active vertex's value, and the VF computes the influence of an active hyperedge's value.

To parallelize hypergraph processing, a practical solution is to divide hyperedges and vertices into many chunks (as shown in Figure 4(c)), which are then assigned to different cores for parallel processing [41].

B. Excessive Memory Accesses in Hypergraph Processing

As in Algorithm 1, the `VertexPro` and `HyperedgePro` engines adopted in existing hypergraph systems follow an index-ordered order to handle vertices and hyperedges. This may cause high-rate cache misses with excessive off-chip memory accesses, leading to sub-optimal performance.

To demonstrate that hypergraph processing is bottlenecked by main memory accesses, we conduct a set of experiments to count the percentage of cycles stalled on main memory accesses for hypergraph processing operating over Hygra [41].

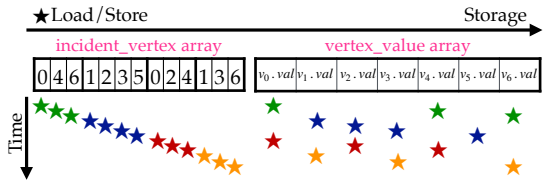


Figure 6. A simplified example for illustrating main memory access patterns to the `incident_vertex` array and `vertex_value` array by running the vertex computation procedure on the hypergraph in Figure 1 in an ascending order of hyperedge index, i.e., $\langle h_0, h_1, h_2, h_3 \rangle$

Figure 5 shows the normalized execution time arising from off-chip memory accesses by benchmarking *Breadth First Search* (BFS), *PageRank* (PR), *Betweenness Centrality* (BC), and *Connected Components* (CC) on five real-world hypergraphs (i.e., *Friendster* (FS), *com-Orkut* (OK), *LiveJournal* (LJ), *Web-trackers* (WEB), and *Orkut-group* (OG)). Dataset details can be found in Table II. All results are reported by the Intel VTune Profiler [21]. We see that off-chip main memory accesses take 51.08% of overall execution time on average. For PageRank on WEB, this ratio can be as high as 84.01%, dominating overall performance of hypergraph processing.

We further investigate the major villains that contribute most to off-chip memory accesses. Since *vertex computation* has a similar process with *hyperedge computation*, let us consider *vertex computation* as an example for illustration purposes. Given the hypergraph in Figure 1(a). Figure 6 shows the pattern of memory accesses to the `incident_vertex` and `vertex_value` arrays during vertex computation. Assume hyperedges are processed in an ascending order of hyperedge index (i.e., $\langle h_0, h_1, h_2, h_3 \rangle$), this will result in strong spatial locality for `incident_vertex` and `hyperedge_offset` accesses. Note that `hyperedge_offset` is not drawn in Figure 6 due to space limitations and its memory access pattern is similar to `incident_vertex`. However, `vertex_value` accesses exhibit poor spatial and temporal locality. It is observed that the accesses to `vertex_value` and `hyperedge_value` dominate most of (>90.80%) memory accesses, accounting for a major reason of cache misses (as discussed in §VI-C).

C. Limitations of Existing Efforts

Hardware-Accelerated Traversal Scheduling. HATS [34] presents a hardware-accelerated traversal scheduler to improve the locality of ordinary graph applications significantly by carefully reordering the vertex scheduling sequence. Nevertheless, HATS is not an ideal solution for handling hypergraph applications.

First, HATS, designed for conventional graph processing, is not able to represent and process hypergraphs. In graph processing, each graph is often stored in a CSR format, and only vertices need to be updated in each iteration. Hypergraph processing is far more complex to treat hyperedges differently from vertices. As discussed in §II-A, vertices in a hypergraph are often stored in the CSR format and

Figure 7. Performance comparison of ChGraph with the variant of HATS (denoted as HATS-V). All results is normalized to HATS-V.

hyperedges are in the CSR format. However, HATS fails to traverse them alternatively due to the lack of control mechanisms. Furthermore, vertices and hyperedges need different update semantics while HATS updates only vertices and thus may produce incorrect results when handling hypergraphs.

Second, HATS generates the scheduling sequence for vertices only for ordinary graph applications. However, hypergraph applications are significantly different that the scheduling sequence of vertices and hyperedges need to be generated alternately in every iteration. From another perspective, since the conventional graph can be considered a special case of the hypergraph, ChGraph is general enough to handle graph applications. In other words, HATS can be understood as a special version of ChGraph for handling graph applications only as discussed in §VI-I.

To demonstrate the inadequacy of HATS for hypergraph processing, we have modified HATS to support hypergraph applications (denoted as HATS-V), with the following modifications. First, we use the index renumbering to help HATS distinguish vertices and hyperedges. Second, to enable alternative traversal, a new control logic is added into each stage of the HATS pipeline. Third, the update function is modified to treat hyperedge computation differently from vertex computation.

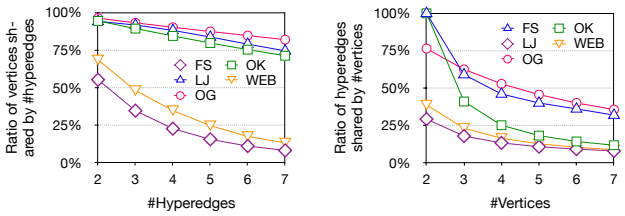
Figure 7 shows the performance of HATS-V against ChGraph. We see that HATS-V is still inferior to ChGraph by 2.56×∼3.01×. The reasons are simple. HATS does not take into account the overlap feature of a hypergraph and thus cannot benefit from overlap-inducing locality. Also, HATS-V needs to traverse two redundant bipartite edges to find a neighbor with much extra overhead.

Hardware Prefetchers. Hardware prefetchers [1], [2], [50] are also developed with indirect accesses to improve memory efficiency. These hardware prefetchers aim to hide access latency for saturating memory bandwidth. In contrast, ChGraph focuses on utilizing bandwidth fully without prefetching too much noisy data by changing the scheduling order. More detailed results are discussed in §VI-H.

D. Overcoming Inefficiencies

In this work, we find that the underlying locality in hypergraph processing is closely related to the *overlapped* feature of hypergraphs, in the sense that most of vertices (hyperedges) can be shared by multiple hyperedges (vertices).

Figure 8(a) shows the ratios of vertices that can be shared with a different number of hyperedges traced from five real-world hypergraphs. It is observed that 55.37%∼96.32% of



(a) Ratio of shared vertices (b) Ratio of shared hyperedges

Figure 8. The sharable ratio variation of (a) vertex with respect to different number of hyperedges and (b) hyperedge with respect to different of vertices

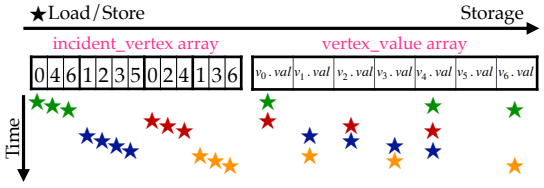


Figure 9. A simplified example for illustrating main memory access patterns to the `incident_vertex` array and `vertex_value` array by running the vertex computation procedure in the overlap-inducing chain order, i.e., $\langle h_0, h_2, h_1, h_3 \rangle$

vertices can be shared by two hyperedges. This hallmark overlapping feature gives us great opportunities to improve locality, such that most off-chip memory accesses can be transformed into fast on-chip cache accesses. The basic idea is to handle active hyperedges along with a new order following the overlap-inducing sequence. This prioritizes to schedule overlapped hyperedges together, ensuring that their shared incident vertices can be reused once it is loaded into the cache. As shown in Figure 8(b), the vertex overlapping has a similar observation as the hyperedge overlapping.

Figure 9 shows that overlap-inducing order can improve temporal locality for `vertex_value`. Let the overlap-inducing hyperedge order be $\langle h_0, h_2, h_1, h_3 \rangle$. Assume the cache is sized of 4. The inclusions (i.e., v_0, v_4, and v_6) of h_0 are first loaded into the cache, followed by v_0 and v_4 reused when h_2 is processed. When h_1 starting being processed, v_2 resides in the cache at this moment and hence can be reused. But vertices v_0, v_4, and v_6 will be swapped out of the cache by v_1, v_3, and v_5. Therefore, when h_3 is scheduled, v_1 and v_3 will be reused again. In total, 8 off-chip memory accesses have been incurred while the overlap-oblivious solution requires 12 ones.

Note that the overlap-inducing order may lead random accesses to the first incident vertex of each hyperedge in `incident_vertex`, but most of its remaining elements still enjoy spatial locality. Compared to the dramatically-improved locality in `vertex_value`, the overall locality can be still improved significantly (as discussed in §VI-C).

In order to identify overlap-inducing orders, we decompose a hypergraph into a set of disjoint hyperedge and vertex chains. Each hyperedge (vertex) chain consists of a number of hyperedges (vertices). As witnessed in Figure 8, since hyperedges and vertices are often overlapped, a chain can naturally capture the overlapping hyperedges or vertices of a hypergraph easily for exposing data reuse opportunities.

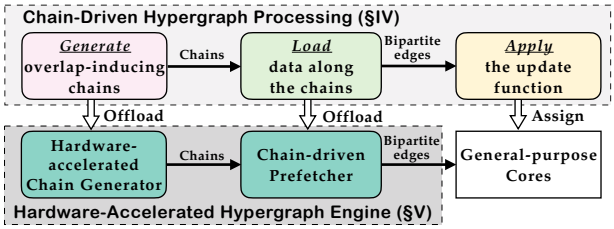


Figure 10. Overview of our approach

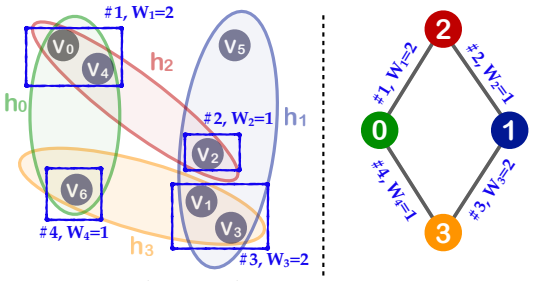
Nevertheless, it is still difficult, if not impossible, to apply the chain-driven scheduling into hypergraph processing. First, to capture the overlapping patterns of a hypergraph, an intuition is to repeatedly traverse the whole hypergraph, leading to expensive preprocessing overheads. Encoding the overlapping semantics efficiently and concisely is challenging. Second, a hypergraph may be decomposed into many chains. It is also difficult to find a chain that can maximize locality opportunities. Even worse, the overlapped data of a hypergraph is not necessarily used in every iteration, depending on whether they are activated at runtime. Thus, the generation of a chain needs to consider not only the structural overlapping but also the activity information, making the situation complex. Third, related to the second challenge, a chain depends on runtime activity information, rendering to generate a chain on-the-fly. This would incur expensive runtime overheads that may outweigh its benefits achieved. How to ensure the efficiency of the chain-driven scheduling remains challenging.

III. Overview

Figure 10 shows the overview of our solution. The key innovation of this work lies in a novel chain-driven hypergraph processing that can *effectively* expose overlap-inducing locality in hypergraph processing. To further exploit locality *efficiently*, we propose a hardware-accelerated hypergraph engine, ChGraph, which is specialized in accelerating the expensive chain generation and data prefetching, enabling to unleash the potential of general-purpose cores with more computation parallelism and fewer memory requests. ChGraph functions like a lightweight plug-in for the *Generate* and *Load* phases while the general-purpose core is in charge of the *Apply* phase. ChGraph communicates with the core through a FIFO buffer.

Chain-Driven Hypergraph Processing (§IV). We first extract the structural overlapping information of a hypergraph to build an overlap-aware abstraction graph, which is a concise lossy representation with a good tradeoff between space efficiency and expression ability. Based on this abstraction graph, we then present an elegant approach that can transform the complex chain generation problem into a simple graph traversal problem. Finally, a novel chain-driven execution model is applied to explore the overlap-inducing locality in hypergraph processing effectively.

Hardware-Accelerated Hypergraph Engine (§V). This part aims to accelerate the chain-driven execution for



(a) Example hypergraph (b) Hyperedge OAG

Figure 11. A simplified example illustration. (a) An example hypergraph. (b) The *overlap abstraction graph* (OAG) for hyperedges. For facilitating descriptions, each edge in the OAG is numbered as `#N`. W_i is the weight of the edge `#i`.

exploiting locality efficiently. We architect a hardware-accelerated hypergraph engine, namely ChGraph, near each general-purpose core. In addition to inter-core parallelism, ChGraph provides the intra-core pipeline parallelism that allows generating multiple chains simultaneously. ChGraph takes charge of inefficient memory management of general-purpose cores and uses a chain-guided data prefetcher to hide the memory latency. This frees up the potential of general-purpose cores in concentrating only on computing while the timely data provision is ensured by ChGraph, making hypergraph processing performant.

IV. Chain-Driven Hypergraph Processing

This section first introduces the chain definition, and then presents our chain-driven hypergraph execution model.

A. Graph-based Chain Model

Given a hypergraph, the key to a chain is to identify the overlap information. We propose the *overlap-aware abstraction graph* (OAG) to achieve this goal. For illustration purposes, we consider the hyperedge as an example.

Definition 1 (*Overlap-aware Abstraction Graph* (OAG)). *Given a hypergraph, a hyperedge OAG is a weighted undirected graph where each vertex represents a hyperedge. An edge between h and h' indicates if h and h' are overlapped, and its weight represents $|N(h) \cap N(h')|$.*

Figure 11 illustrates how to generate a hyperedge OAG (*abbr.* H-OAG) from a hypergraph. For facilitating descriptions, each edge in H-OAG is numbered as `#N` and W_i denotes the weight of the edge `#i`. As shown in Figure 11(b), we use four vertices 0, 1, 2, and 3 to represent h_0, h_1, h_2, and h_3, respectively. According to the OAG definition, a non-empty overlapping set of any two hyperedges will create an edge between them. For example, since $N(h_0) \cap N(h_2)$ (i.e., $\{v_0, v_4\}$) is not empty, an edge `#1` between 0 and 2 is created and its weight W_1 is $|N(h_0) \cap N(h_2)|$ (i.e., 2).

The OAG is stored in the CSR format. For space efficiency, users can set a threshold W_{min} to prevent creating the edges whose weights are less than W_{min}. This simplifies the OAG by discarding those unimportant edges that improve little locality. In essence, this presets a good

Algorithm 2 Chain-Driven Hypergraph Processing

Input: $G\langle V, H\rangle$ – Hypergraph
HF – Hyperedge value update function
VF – Vertex value update function
V-OAG – The overlap abstraction graph for vertices
H-OAG – The overlap abstraction graph for hyperedges
Output: `vertex_value` and `hyperedge_value`

1: VERTEXINIT(`vertex_value`)
2: HYPEREDGEINIT(`hyperedge_value`)
3: ActiveVertexSet $FrontierV$.SETUP( )
4: **while** $cur\_iteration\_num < iteration\_num_{max}$ **do** ▷ Hyperedge Computation
5: /*Active vertices update their incident hyperedges using HF*/
6: $chains$ = **GENERATE**($FrontierV$, V-OAG)
7: **while** $chains$!= NULL **do**
8: $data$ = **LOAD**(G, $chains$.pop())
9: ActiveHyperedgeSet hs = hs ∪ **APPLY**(HF, $data$)
10: **end while**
11: ActiveHyperedgeSet $FrontierE$ = hs
12: $FrontierE$.ISEMPTY( ) /*Loop until frontier is empty*/ ▷ Vertex Computation
13: /*Active hyperedges update their incident vertices using VF*/
14: $chains$ = **GENERATE**($FrontierE$, H-OAG)
15: **while** $chains$!= NULL **do**
16: $data$ = **LOAD**(G, $chains$.pop())
17: ActiveVertexSet vs = vs ∪ **APPLY**(VF, $data$)
18: **end while**
19: $FrontierV$ = vs
20: $FrontierV$.ISEMPTY( ) /*Loop until frontier is empty*/
21: $cur\_iteration\_num$++
22: **end while**

tradeoff between space overhead of OAG and representation ability of overlapping semantics. Note that a setting of $W_{min} > 1$ may miss some overlapping information, but the final correctness is not affected since the data that miss the overlapping information will be safely scheduled in order of their indices. Since many hyperedges are overlapped by one or two vertices in real-world hypergraphs, in this work we empirically set $W_{min} = 3$, which reduces the OAG size significantly while still preserving high locality that can be exploited (as discussed in §VI-F). Note that the OAG construction can be achieved by a hypergraph preprocessing, the overheads of which can be amortized by multiple executions of different hypergraph applications.

Definition 2 (Chain). *A chain c_l^j is a sequence of connected vertices of an OAG, i.e., $\langle V^0, V^1, ..., V^{|c_l^j|-1}\rangle$, where l can be h or v, indicating the type of chain. j represents the chain identifier. $|c_l^j|$ represents its length.*

Consider the hypergraph in Figure 11(a). There are four potential hyperedge chains meeting the chain definition based on the H-OAG in Figure 11(b). For example, the chain rooting from h_0 is $c_h^0 = \langle h_0, h_2, h_1, h_3\rangle$. More details of how to generate expected chains are discussed in §IV-B. Once a directed chain is generated, the scheduling order of hyperedges in the hypergraph can be then specified.

B. Chain-Driven Execution Model

We next introduce our chain-driven *Generate-Load-Apply* (GLA) execution model to show how it exploits overlap-inducing locality. Similar to Hygra [41], the hypergraph data are logically divided into several chunks (as shown in Figure 4(c)), which are assigned to different cores for parallel processing. Each chunk has a hyperedge OAG or a vertex OAG. Note that our GLA model is compatible and flexible with other partitioning methods [19], [25], [32].

Algorithm 3 Chain Generation

1: **procedure** EXPLORE($root$, $Frontier$, OAG, c, D)
2: Set the vertex $root$ in the OAG as visited
3: Remove $root$ from $Frontier$
4: INSERT(c, $root$) /*Insert into the queue pointed by c*/
5: **if** $root$ has unvisited active neighbors & $D < D_{max}$ **then**
6: $N \leftarrow OAG$.GETNEIGHBORVERTICES($root$)
7: SORT(N) /*Sort them based on the edge weights*/
8: **for each** $v \in N$ **do**
9: **if** v is unvisited & $v \in Frontier$ **then**
10: EXPLORE($root$, $Frontier$, OAG, c, $D+1$)
11: **else**
12: NEWCHAIN(c)
13: **end if**
14: **end for**
15: **else**
16: NEWCHAIN(c)
17: **end if**
18: **end procedure**
19: **procedure** GENERATE($Frontier$, OAG)
20: **for each** $root \in Frontier$ **do**
21: EXPLORE($root$, $Frontier$, OAG, c, D)
22: **end for**
23: **return** c
24: **end procedure**

Algorithm 2 depicts the procedure of our GLA model. In each iteration, both the hyperedge computation kernel and the vertex computation kernel are divided into three phases: *chain generation*, *chain-guided data loading*, and *update applying*. We next describe them respectively.

Chain Generation. In each iteration, each chunk and its OAG will be assigned to a core, which first performs Algorithm 3 to generate chains. Each thread repeatedly takes the active data in the chunk as an initial root (Lines 20-22). Active data here is a worklist in the synchronous execution model, in the sense that the data value has been changed in the current iteration and it will be used in the next iteration [18], [41]. Then, all other active vertices in the OAG starting from the root vertex will be explored in a depth-first order until all active vertices are visited. The initial depth (i.e., D) is set to 0 (Lines 1-18). Note that the state of a vertex in OAG is determined by the state of data (e.g., hyperedge) it represents. To avoid the skewed distribution of the chain lengths, we set D_{max} to 16 by default (Line 5), which seems a nice sweet spot for yielding good performance (discussed in §VI-F).

In each exploration, the visited root vertex is added into a queue to build chains (Line 4). Since the queue is shared, when a chain is generated, its range is recorded as the offset of its beginning vertex in the queue (Lines 12 and 16). Specifically, before a new chain starts generation, NEWCHAIN(c) records the chain queue's offset to store its first vertex, where c points to the next position of the chain queue. When there is a set of active neighbors to be visited, the one with the highest edge weight with the root vertex will

be selected (Lines 6-10). To avoid the costly weight sorting overhead, we enforce to store the CSR-based edges of each vertex in a descending order according to their weights.

Consider the example in Figure 11(b). Assume four vertices are all active. When ❶ is taken as the root, we first set it as visited and inactive. Meanwhile, h_0 represented by ❶ is added as the first element of the chain $c_h^0 = \langle h_0 \rangle$. Then, the procedure starts exploring unvisited active neighbors (i.e., ❷ and ❸) connecting to the root vertex. Since the edge `#1` has a larger weight than `#4`, we take ❷ as a selected new root and repeat the above procedure until no vertices are left active. Finally, a chain $c_h^0 = \langle h_0, h_2, h_1, h_3 \rangle$ is generated.

Chain-Guided Data Loading. For each generated hyperedge chain, we next introduce how to get the chain-ordered hyperedge and load all bipartite edges of a hyperedge (Lines 8 and 16 in Algorithm 2). We propose to use the *tuple*, denoted as $\{h_{id}$, v_{id}, `hyperedge_value`$[h_{id}]$, `vertex_value`$[v_{id}]\}$, to facilitate the data loading with the following advantages. First, it shields users from the details of data access so that developers only need to focus on the requested data itself. Second, it enables exploiting data reuse at a coarse-grained level, simplifying the system design.

Consider the hyperedge chain $c_h^0 = \langle h_0, h_2, h_1, h_3 \rangle$. Each chain has a unique tuple. The first bipartite edge of h_0 in Figure 4(b) (i.e., $\langle h_0, v_0 \rangle$) and its two associated vertex data are fetched and packaged into the tuple $data = \{h_0$, v_0, `hyperedge_value`$[h_0]$, `vertex_value`$[v_0]\}$. When the next bipartite edge (i.e., $\langle h_0, v_4 \rangle$) of h_0 is loaded, h_0 and `hyperedge_value`$[h_0]$ have already resided in the tuple so that only two elements (i.e., v_4 and `vertex_value`$[v_4]$) of the tuple are updated. After h_0 is finished, h_2 is next processed and its first bipartite edge (i.e., $\langle h_2, v_0 \rangle$) of h_2 will be loaded. At this moment, v_0 and `vertex_value`$[v_0]$ are resident in the cache, so only h_2 and `hyperedge_value`$[h_2]$ are needed to be loaded from the off-chip memory. Other hyperedges in the chain are accessed similarly.

Update Applying. For each loaded tuple $data = \{h_{id}$, v_{id}, `hyperedge_value`$[h_{id}]$, `vertex_value`$[v_{id}]\}$, the vertex update function (i.e., VF shown in Algorithm 1) takes it as input and calculates the influence of the value of h_{id} (i.e., `hyperedge_value`$[h_{id}]$) on that of v_{id} (i.e., `vertex_value`$[v_{id}]$). When the new value of v_{id} is different from the original value, v_{id} is returned for subsequent computation (Line 17 in Algorithm 2). The *apply* operation for *hyperedge computation* is similar (Line 9 in Algorithm 2).

V. ChGraph

This section introduces the specialized hardware to accelerate the chain-driven hypergraph processing.

A. Overall Architecture

System Architecture. Figure 12(a) shows our system architecture. Coupled with the general-purpose core, ChGraph is specialized in accelerating the chain generation and chain-ordered data prefetching. The general-purpose cores are in charge of initialization, configuration, and data processing. ChGraph is easy and flexible to implement at any level of the cache hierarchy. We select to architect ChGraph in the L1 cache level since it achieves a nice sweet spot in three dimensions: routing, timing, and capacity considerations [51]. ChGraph accesses the main memory via the L2 cache and passes the fetched tuple data (discussed in Section IV-B) to general-purpose cores via a FIFO buffer. General-purpose cores logically divide the hyperedges and vertices into chunks, dispatched further to different cores for parallel processing. The per-chunk OAGs and hypergraph data are stored in the main memory.

Specifically, each general-purpose core initializes the data values and states in the chunk, and then completes the configuration of its private ChGraph engine. When ChGraph receives activation information (e.g., hyperedges) from the general-purpose core, it starts generating chains based on the OAG. Following the chain order, ChGraph then prefetches bipartite edges, which are then sent to the general-purpose core and handled in turn by an algorithm-specific update function (i.e., VF or HF as in Algorithm 1). For the GLA model, ChGraph enables achieving more benefit gains against its overheads for two reasons. First, ChGraph architects a sophisticated pipelining design for chain generation. Second, the accurate chain-ordered prefetching hides memory latency significantly. We note that OAG entries can be discarded rather than written back to the memory when they are evicted from the cache. This is because the OAG structure is built statically from a hypergraph and it will not be changed at runtime. Thus, no cache inconsistency exists. The data access overhead on the OAGs this way can be therefore improved significantly.

ChGraph Microarchitecture. Figure 12(b) further shows the microarchitecture of ChGraph, including a hardware-accelerated chain generator and a chain-driven prefetcher. The communication between two components through a *chain FIFO* buffer, enabling a coarse-grained inter-component pipeline to fuse the phase-by-phase execution. Before the two components start working, the core configures ChGraph for conveying the hypergraph and OAG to ChGraph (❶). In addition, the hyperedge and vertex states are maintained in a `bitmap` with 1 (0) indicating that they are active (inactive), also passed to ChGraph.

***Hardware-accelerated Chain Generator* (HCG)**. After obtaining the essential information from the general-purpose core, HCG gets an active hyperedge (or vertex) data based on the `bitmap` and takes the corresponding vertex in the OAG as the initial root to generate chains. The generated chains are stored in the *chain FIFO* buffer (❷).

***Chain-Driven Prefetcher* (CP)**. When the chains reside in the FIFO buffer, CP gets started (❸). It sequentially reads hyperedges (vertices) according to their chain order, and prefetches the bipartite edges of each data, which are stored

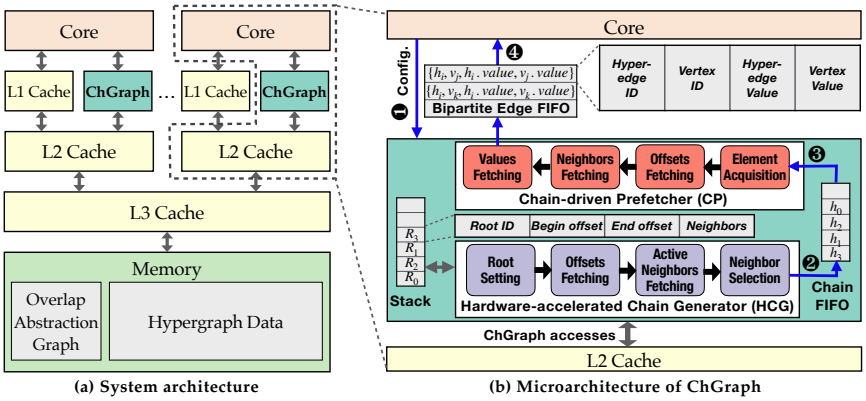


Figure 12. The overall architecture

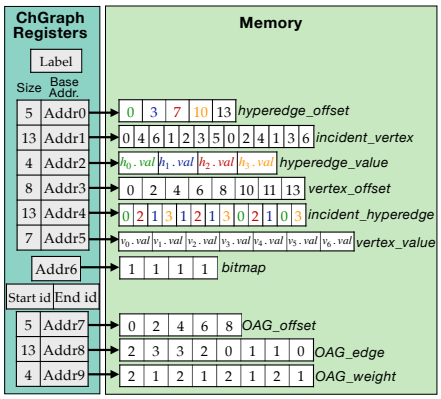


Figure 13. The information configuration

in the *bipartite edge FIFO* buffer. The general-purpose core can obtain the bipartite edge data from this buffer (❹).

Once all vertex chunks have been handled in the *hyperedge computation* phase, the *vertex computation* is then invoked. The iterative hypergraph algorithm is converged if no active data in all chunks are left.

The ChGraph-Core Interaction. Before a hypergraph chunk is processed, the core will use a CH_CONFIGURE instruction to configure ChGraph, which conveys the information to the memory-mapped registers accessible to the ChGraph engine. When generating the chain, ChGraph fills the *bipartite edge FIFO* buffer with bipartite edges for the processing of the core. We provide an ISA instruction CH_FETCH_BIPARTITE_EDGE to obtain these prefetched bipartite edges from the *bipartite edge FIFO* buffer. Thus, the conventional datapath of normal load instructions can be bypassed. To support this feature seamlessly, we expose the functionality of ChGraph to the upper hypergraph software framework through two simple low-level APIs: `ChGraph_Configure()` corresponding to the CH_CONFIGURE instruction, and `ChGraph_fetch_bipartite_edge()` corresponding to the CH_FETCH_BIPARTITE_EDGE instruction.

Generality. ChGraph is designed for generating a locality-aware scheduling sequence only and leaves algorithm-specific logic computation to general-purpose cores. Thus, ChGraph is transparent to the changes of hypergraph algorithm. ChGraph is currently built upon the general-purpose system, but its key designs, independent of a specific underlying platform, are general enough to be applied to any hypergraph-specific systems or accelerators to improve locality. Furthermore, since the conventional graph can be considered as a special case of the hypergraph, where each hyperedge is associated with only two vertices. Thus, ChGraph is also capable of handling ordinary graph applications, as witnessed in §VI-I.

B. Hardware Designs

Configuration. As shown in Figure 13, the general-purpose core uses the memory-mapped registers [34] to convey the following information: 1) A label with 1(0) indicating the phase of either *hyperedge computation* or *vertex computation*; 2) The sizes and base addresses of the `hyperedge_offset`, `incident_vertex`, `hyperedge_value`, `vertex_offset`, `incident_hyperedge`, and `vertex_value` arrays; 3) The base address of the `bitmap`; 4) The first and last indices of data in a chunk to be processed; and 5) The sizes and base addresses of the OAG data structures (i.e., `OAG_offset`, `OAG_edge`, and `OAG_weight`).

Chain Generation. As shown in Figure 12(b), the chain generator is equipped with a 4-stage pipeline: *root setting*, *offsets fetching*, *active neighbors fetching*, and *neighbor selection*. Specifically, the pipeline starts from an empty stack. The *root setting* stage chooses the active data with minimal index from the `bitmap` in turn. Then, its corresponding vertex in the OAG is taken as a root and pushed into the stack. Note that once the data is selected, it will be marked as inactive immediately for correctness. The *offsets fetching* stage gets the topmost vertex from the stack and obtains its first and last offsets from the `OAG_offset` array. The *active neighbors fetching* stage loads a cacheline of unvisited neighbor IDs from the `OAG_edge` array. The *neighbor selection* stage picks the neighbor with maximal edge weight, which is pushed into the stack and also outputted to the *chain FIFO* buffer.

Once no unvisited active neighbors of the root vertex can be obtained or the stack is full, all vertices will be popped out of the stack. A new round of exploration starts and repeats the above four stages. Note that, when the generator has visited all active vertex in the OAG, it will insert '-1' into the *chain FIFO* buffer to notify the completion of HCG.

Chain-Driven Prefetching. Whenever an entry is pushed into the *chain FIFO* buffer by the *neighbor selection* stage of HCG, CP prefetches its relevant bipartite edge data with the four stages (as shown in Figure 12). In the *element acquisition* stage, one element (e.g., h_0) will be popped out from the *chain FIFO* buffer. Then, the *offsets fetching* stage obtains the first and last offsets in the bipartite edges for this element from the `hyperedge_offset` or `vertex_offset`

Table I
CONFIGURATION OF THE SIMULATED SYSTEM

Structure	Configuration
Cores	16 cores, x86-64 ISA, 2.2GHz, Haswell-like OOO [40]
L1 Caches	32KB per-core, 8-way set-associative, split D/I, 3-cycle latency
L2 Cache	128KB per-core, 8-way set-associative, 6-cycle latency
L3 Cache	32MB shared, 16 banks, 16-way hashed set-associative, inclusive, 24-cycle bank latency, LRU replacement
Global NoC	4×4 mesh, 128-bit flits and links, X-Y routing, 1-cycle pipelined routers, 1-cycle links
Coherence	MESI, 64B lines, in-cache directory, no silent drops
Main Memory	4 controllers, DDR4 1600 (12.8 GB/s per controller)

array. In the *neighbors fetching* stage, the indices of this element's neighbors are loaded from the `incident_vertex` or `incident_hyperedge` array. The *values fetching* stage fetches the neighbor attributes from the *vertex_value* or *hyperedge_value* array, and packs each bipartite edge into a tuple. Once the *element acquisition* stage gets '-1', CP will insert a fake tuple $\{-1,-1,-1,-1\}$ into the *bipartite edge FIFO* buffer and is then stalled. The core will be suspended when such a fake tuple is detected.

Note that ChGraph is a synchronous hypergraph system like Hygra [41] in which updated properties in an iteration are prepared for being used only in the next iteration. Thus, new updated properties are not allowed to be used in the same iteration, without coherency issues.

VI. EXPERIMENTAL EVALUATION

This section evaluates the efficiency and effectiveness of ChGraph against the state-of-the-art.

A. Experimental Setup

Simulation Settings. We consider implementing ChGraph in an ASCI fashion, which allows having a handcrafted datapath for the functionality directly imprinted in hardware, so that costly instruction control overheads in other implementation fashions (e.g., RISC-V cores) can be eliminated.

We use ZSim [40] to simulate a 16-core system, whose parameters are shown in Table I as in previous work [34]. The energy consumption of the chip components and the main memory is obtained using McPAT [31] and Micron DDR3L datasheets [33], respectively. A cycle-accurate simulator is designed to model the microarchitecture behavior of ChGraph. We also implement each hardware module in Verilog RTL, then synthesize it with Synopsys toolchain using TMSC 65nm standard library. The power consumption is evaluated using Synopsys PrimeTime PX. The area, power, and latency for the on-chip buffers are estimated via CACTI 6.5 [27]. ChGraph runs at a frequency of 1GHz.

Hypergraph Datasets and Applications. As in the previous studies [18], [24], [41], we evaluate ChGraph on five real-world hypergraphs (shown in Table II). All these datasets are publicly available from the *Stanford Large Network Dataset Collection* (SNAP) [29] and *Koblenz Network Collection* (KONECT) [26]. "`#BEdges`" in Table II denotes the number of bipartite edges in the bipartite graph representation. ChGraph supports both directed and undirected hypergraphs. In our evaluation, all hypergraphs are considered undirected.

Table II
REAL-WORLD HYPERGRAPH DATASETS

Datasets	#Vertices	#Hyperedges	#BEdges	Size
Friendster (FS) [29]	7.94M	1.62M	23.48M	0.4GB
com-Orkut (OK) [29]	2.32M	15.30M	107.08M	1.8GB
LiveJournal (LJ) [26]	3.20M	7.49M	112.31M	1.5GB
Web-trackers (WEB) [26]	27.67M	12.77M	140.61M	2.2GB
Orkut-group (OG) [26]	2.78M	8.73M	327.03M	4.6GB

ChGraph is evaluated with six widely-used hypergraph applications: *Breadth First Search* (BFS), *PageRank* (PR), *maximal independent set* (MIS), *Betweenness Centrality* (BC), *Connected Components* (CC), and *k-core decomposition* (*k*-core). We benchmark PR within 10 iterations while others run until convergence.

Baseline. We compare ChGraph with a state-of-the-art CPU-based hypergraph processing system – Hygra [41]. Hygra is an in-memory single machine system that uses the typical index-ordered scheduling. We evaluate Hygra on a machine configured as in Table I.

B. Performance

We compare ChGraph with Hygra [41] with two options: **GLA** – a pure software-based GLA implementation, and **ChGraph** – a hardware-accelerated implementation. Figure 14 shows the results. On average, GLA is 1.51×, 1.13×, 1.30×, 1.62×, 1.56×, and 1.60× slower than Hygra for BFS, PageRank, MIS, BC, CC, and *k*-core respectively. As discussed in §I, the main reason lies in the fact that GLA suffers from excessively high runtime overheads for generating overlap-inducing chains. As a consequence, this overhead in almost all test cases negates the benefits achieved from the main memory access reduction. Compared to other hypergraph algorithms, PageRank shows the smallest performance difference of GLA against Hygra. The reason behind is simple. Since all hyperedges and vertices for PageRank are active, the per-iteration chain in this case will be the same without any changes. Thus, GLA only needs to generate the chains in the first (rather than every) iteration, yielding the minimal runtime overhead and performance loss.

We can also see that ChGraph outperforms Hygra by 3.39×∼4.73× (4.12× on average). The reasons are threefold. First, the overlap-inducing chain order applied in ChGraph is locality friendly and reduces excessive redundant main memory accesses in Hygra. Second, the overhead arising from the chain generation is suppressed successfully by our sophisticated hardware designs. Finally, ChGraph additionally uses a chain-guided prefetching mechanism that resolves the memory latency problem arising in Hygra. This prefetcher enables exploiting the information of upcoming tasks in advance so that their memory access latency can be hidden behind the normal computation of on-going tasks.

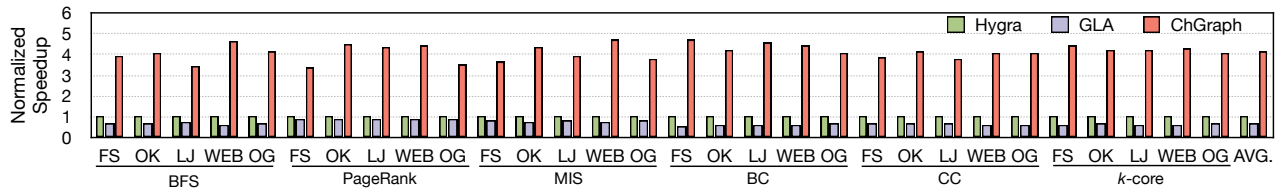
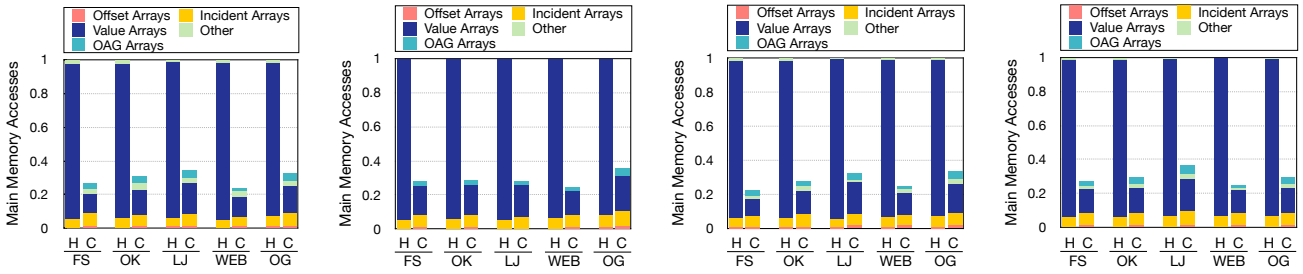


Figure 14. Performance of ChGraph against the pure software-based GLA solution (i.e., GLA) and a state-of-the-art hypergraph system Hygra [41]



(a) BFS (b) PageRank (c) BC (d) CC

Figure 15. The quantitative decomposition of main memory accesses to different data arrays (shown in Figure 13) between **H**ygra and **C**hGraph

C. Off-Chip Main Memory Accesses

Figure 15 compares ChGraph (*abbr.* **C**) with Hygra (*abbr.* **H**) in terms of main memory accesses. Each bar shows the breakdown of main memory accesses to 10 arrays in Figure 13. The **offset arrays** include the `hyperedge_offset` and `vertex_offset` arrays. The **incident arrays** include the `incident_vertex` and `incident_hyperedge` arrays. The **value arrays** include the `hyperedge_value` and `vertex_value` arrays. The **OAG arrays** include `OAG_offset`, `OAG_edge`, and `OAG_weight` arrays. **Other** indicates the accesses to the `bitmap`. Overall, ChGraph reduces main memory accesses by 2.77×∼4.56× (3.51× on average) against Hygra. These benefits mainly come from the chain-driven execution model applied, which regularizes the processing order of hypergraph data with better locality.

The reduced number of main memory accesses varies for different hypergraphs. Compared with Hygra, ChGraph reduces the total number of main memory accesses by 3.88×, 3.43×, 3.07×, 4.09×, and 3.06× on average for FS, OK, LJ, WEB, and OG, respectively. To understand this, let us recall Figure 8, in which 71.31%∼82.03% of vertices can be shared by seven hyperedges in OG, LJ, and OK while FS and WEB take only 8.26%∼13.27%. In this case, even if the original index-ordered scheduling is applied, these frequently-used vertices existing in OG, LJ, and OK naturally fit the LRU cache replacement strategy [4], showing lots of locality that can be exploited both by general-purpose cores and ChGraph, therefore incurring a relatively-small difference.

We also see that ChGraph significantly improves the cache locality for the **value arrays**, but increases slightly cache misses for the **incident arrays**. Consider all cases together, ChGraph improves the overall cache locality significantly. For instance, consider BFS on WEB, the memory access proportion to the **value arrays** for Hygra takes 93.01%, which can be reduced to 12.30% by ChGraph with 4.22× overall main memory access reduction. As for OAG arrays, since OAG is not unchanged for a given hypergraph, the entries of the OAGs can be dropped rather than written back to the main memory when they are evicted from the cache. This reduces the number of memory accesses to OAG arrays, taking only 6.86%∼12.08% of the total amount. Since all data are always active for PageRank, there is no need to access the `bitmap`, yielding negligible `bitmap` accesses.

D. Effectiveness of Hardware Designs

Figure 16 investigates the benefits breakdown for two core hardware of ChGraph: *hardware-accelerated chain generator* (HCG) and *chain-driven prefetcher* (CP). The baseline represents our pure software-based GLA implementation.

HCG. With the hardware support of accelerating the chain generation, runtime generation cost is greatly reduced. Therefore, we see that HCG offers 4.42× performance improvement over the baseline, occupying 92.09% of the overall benefit. In particular, for the all-active PageRank algorithm, it yields the smallest speedup (3.42×) over the baseline compared to other algorithms. The reason is that chains are generated only once so that in this case there incurs relatively little runtime generation overhead that can be reduced by HCG.

CP. CP hides the memory access latency behind the normal computation. We see that CP further improves the performance of hypergraph processing by 1.37× on average against HCG, taking 7.91% of the overall benefit.

E. Area and Power Evaluation

We store the hypergraph and the OAGs in the existing memory subsystem. The area of ChGraph contains only its hardware logic (i.e., HCG and CP) and a few buffers. Specifically, the stack is set with a depth of 16, in which each level contains a vertex index (4 bytes), the beginning offset (4 bytes), the end offset (4 bytes), and a cacheline of

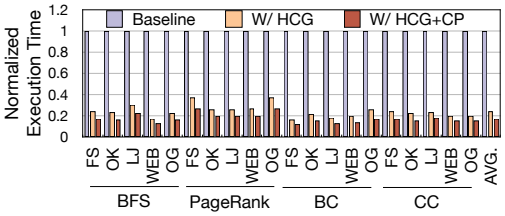


Figure 16. Performance of ChGraph with and without *hardware-accelerated chain generator* (HCG) and *chain-driven prefetcher* (CP)

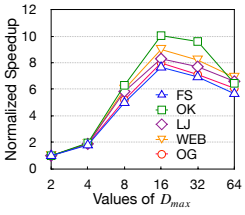


Figure 17. Performance of ChGraph on PageRank with varying D_{max}

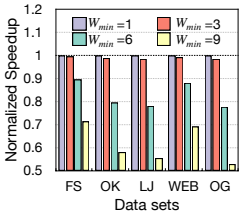


Figure 18. Performance of ChGraph on PR with varying W_{min}

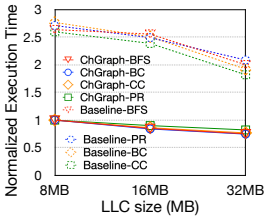


Figure 19. Execution time of ChGraph on WEB with different LLC sizes

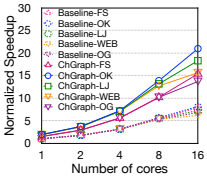


Figure 20. Performance of ChGraph on PR with different number of cores

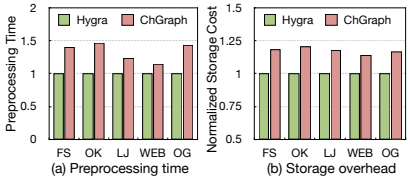


Figure 21. Preprocessing overhead of ChGraph against Hygra in terms of (a) preprocessing time and (b) storage overhead

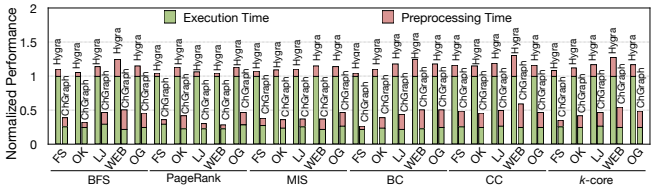


Figure 22. The total running time (including preprocessing time) of ChGraph against Hygra. All results are normalized to the execution time of Hygra.

neighbor indices (64 bytes). Thus, the stack incurs 1.19KB storage. The *chain FIFO* buffer is configured with a size of 32, incurring just 0.13KB storage. Since each *tuple* is set to be 24 bytes and sized of 32, the *bipartite edge FIFO* buffer takes 0.75KB storage space. The registers shown in Figure 13 are with only 84 bytes. In addition, ChGraph eliminates expensive instruction control overheads by having a handcrafted datapath for the functionality directly imprinted in hardware. Therefore, ChGraph is rather cheap, requiring only 0.094mm^2 area that is only 0.26% of a core in the Intel Core2 E6750 processor manufactured in 65nm process [12]. The power consumption of ChGraph is 61mW, which takes about 0.19% of core TDP.

F. Sensitivity Study

We next show the sensitivity analysis of ChGraph with different configuration parameters.

Sensitivity to D_{max}. Figure 17 depicts the performance of ChGraph on PageRank with different maximum exploration depths (D_{max} discussed in §IV-B) ranging from 2 to 64. When D_{max} is less than 16, the larger D_{max} is, the better the performance is. This is because a larger D_{max} implies more overlapped data that can be reused. However, when D_{max} exceeds 16, it may enforce to generate more short chains, leading to fewer overlapped data that can be reused.

Sensitivity to W_{min}. Figure 18 characterizes the performance of ChGraph on PageRank with different W_{min} (discussed in §IV-A) from 1 to 9. All results are normalized to W_{min}=1. Overall, the larger W_{min} is, the worse the performance is. The reason behind is that a large W_{min} may filter out some crucial edges that may be useful for generating the expected overlap-maximal chains. In addition, as W_{min} increases from 1 to 3, the performance decreases slightly from 100% to 98.74%. This is because the missed information represents only one or two overlaps with few data reuse chances, leading to little impact on the locality.

Sensitivity to LLC Size. Figure 19 plots the execution time of ChGraph on WEB with the LLC size ranging from 8MB to 32MB. All results are normalized to the execution time of using an 8MB LLC. As the LLC size increases, the performance of ChGraph is improved by 1.30×. The reason is that a larger LLC can cache more reusable hypergraph and OAG data. Compared with the baseline, the LLC size does not affect the performance of ChGraph significantly due to the improved cache hit rate.

Sensitivity to Core Number. Figure 20 further depicts the performance of ChGraph on PageRank with different core numbers. Overall, as the number of cores increases, the performance has been improved while the growth rate gradually decreases. The reason is simple that more cores imply more serious data access contention. However, ChGraph alleviates this situation with fewer memory requests with our chain-driven scheduling. Thus, ChGraph achieves better scalability than the baseline.

G. Preprocessing

Both Hygra and ChGraph need to preprocess a hypergraph for generating their bipartite graph formats, but ChGraph introduces extra OAG graph preprocessing overhead. Figure 21(a) depicts the normalized preprocessing time of ChGraph against Hygra. For FS, OK, LJ, WEB, and OG, ChGraph increases the preprocessing time over Hygra by 39.42%, 46.07%, 23.86%, 13.60%, and 43.06%, respectively. In particular, compared to other hypergraphs, WEB incurs the smallest preprocessing overhead with 13.60% only. It is because that the overlapping inclusion number of many hyperedges in WEB is no more than three vertices. Thus, the creation operation for many edges can be skipped and saved during the OAG construction.

Figure 22 shows the total running time with the preprocessing time included. ChGraph still runs 2.20×∼3.89× faster than Hygra because of the substantial performance

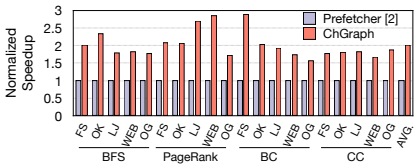


Figure 23. Overall performance (including preprocessing time) of ChGraph against the event-triggered programmable prefetcher [2]

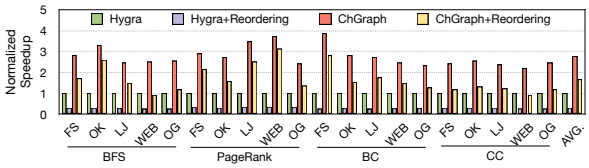


Figure 24. Overall performance comparison (including reordering and preprocessing overheads) of ChGraph with Hygra, Hygra+Reordering, ChGraph+Reordering

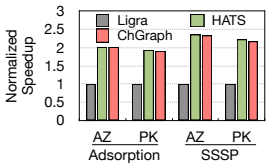


Figure 25. Performance comparison of ChGraph with Ligra and HATS for graph applications

gains reaped from the chain-driven scheduling. Yet, a preprocessed hypergraph can be used for any hypergraph algorithm so that preprocessing overheads incurred can be amortized by multiple executions of a variety of hypergraph algorithms.

Figure 21(b) further shows the extra OAG storage overhead of ChGraph agasint Hygra. Compared to Hygra, ChGraph introduces extra space overhead of 18.19%, 20.41%, 17.48%, 13.93%, and 16.73% for FS, OK, LJ, WEB, and OG, respectively. In spite of some extra spaces required, the OAG incurs negligible memory access overhead because of its read-only attribute as discussed in Figure 15.

H. Compared with Alternatives

Compared with Hardware Prefetcher [2]. Figure 23 shows the results. Overall, ChGraph outperforms the event-driven hardware prefetcher [2] by 1.56×∼2.88×. The reason is simple. The hardware prefetcher [2] aims to hide the access latency for saturating memory bandwidth. ChGraph focuses instead on utilizing bandwidth fully without prefetching too much noisy data.

Compared with Reordering Technique. ChGraph improves only temporal locality related inefficiencies. To improve spatial locality further, we can use a reordering technique that assigns incident vertices of each hyperedge with close-by IDs. We then run Hygra and ChGraph on the reordered hypergraph, respectively. Figure 24 shows the results. We see that the reordering technique does not improve the overall performance due to its high reordering overhead that significantly offsets the benefits achieved.

I. Generality

We also investigate the performance of ChGraph for handling graph applications. Figure 25 shows the overall performance of ChGraph (with the preprocessing time included) against a state-of-the-art graph system Ligra [42] and a state-of-the-art graph accelerator HATS [34] using graph applications Adsorption and SSSP on *com-Amazon* (AZ) and *soc-Pokec* (PK) [29]. Compared to Ligra, ChGraph offers 2.13×, on average. For graph applications, the built OAG would be the same as the input graph. In this case, HATS can be understood as a special version of ChGraph. Thus, we see that ChGraph achieves similar results against HATS.

VII. Related Work

In the past decades, there have emerged a large number of software and hardware solutions for ordinary graph processing in releasing programming efforts [39], [42], [45], reducing communication overhead [9], [10], improving load balance [49], [52], and optimizing memory efficiency [16], [37], [46]. There are also some hardware accelerators adopting the decoupled access-execute design [43] to accelerate graph analytics. Minnow [51] augments general-purpose cores by privileging to handle the high-priority vertex at a low cost. HATS [34], the most related accelerator work to ChGraph, presents a hardware scheduler to improve the locality of graph applications with locality-aware scheduling. In contrast to the conventional graph processing, which aims at analyzing pairwise relationships between vertices, hypergraph processing is more general to analyze complex relationships beyond pairwise ones. This motivates us to devise ChGraph, the first hypergraph-specific hardware solution, to tackle different irregular memory access behaviors in hypergraph applications from that in graph applications, as discussed in §II-C. There still exist hardware prefetchers [1], [2], [50] in improving memory latency of graph processing. GRASP [14] employs specialized cache management to tackle cache thrashing of hot vertices. PHI [35] proposes a cache hierarchy for efficient commutative scatter updates. All these earlier efforts are focused on ordinary graph processing and inadequate to solve challenges in hypergraph processing for the differences in not only graph structure but also architectural behaviors.

Early research of hypergraph processing focuses on algorithmic optimizations for a specific hypergraph application. Ducournau et al. [13] tailor the random walk algorithm in directed hypergraphs to solve the image segmentation problem. Nielsen et al. [36] propose a shortest hyperpath algorithm for the network routing problem. HyperBC [38] presents a betweenness centrality algorithm on hypergraphs. There are also studies for finding maximal independent sets in hypergraphs [5], [23]. Unlike these algorithm-by-algorithm optimizations, ChGraph emphasizes on the general-purpose hypergraph processing problem.

HyperX [24] and MESH [18] are the first generation of general-purpose hypergraph systems in a distributed environment. HyperX [24] provides a ”think like a vertex or hyperedge” model to uniformly express various hypergraph algorithms. MESH [18] proposes to build a hypergraph system quickly from existing mature ordinary graph systems. A symmetry-aware hypergraph partition mechanism is developed to improve load balance of distributed hypergraph processing [15]. CHGL [22] provides high-level operators to improve programming productivity. Hygra [41] is the

first in-memory hypergraph system that makes full of the runtime information of hypergraph processing for improving performance significantly. However, the efficiency of these earlier hypergraph systems is still greatly limited to the underlying architecture, inspiring us to develop a brand new hardware-accelerated hypergraph engine in this work.

VIII. Conclusion

Hypergraph processing applications are typically memory bound. We have observed that the structural overlapping of hypergraphs can be leveraged to improve locality in hypergraph processing significantly. We propose a novel chain-driven execution model that enables exposing locality effectively, and a hardware-accelerated hypergraph engine that allows exploiting locality efficiently. Evaluation on a simulated 16-core system shows that our hardware solution outperforms a state-of-the-art hypergraph system Hygra by up to 4.56× main memory access reduction and by up to 4.73× performance improvement, while introducing only 0.26% area overhead.

Acknowledgment

We thank the anonymous reviewers and the anonymous shepherd for their insightful comments and suggestions. This work is supported by the National Natural Science Foundation of China under Grant No. 61825202, 62072195, and 61832006. The correspondence of this paper should be addressed to Long Zheng.

References

[1] S. Ainsworth and T. M. Jones, "Graph prefetching using data structure knowledge," in *Proceedings of the 2016 International Conference on Supercomputing (ICS'16)*, 2016, pp. 39:1–39:11.

[2] S. Ainsworth and T. M. Jones, "An event-triggered programmable prefetcher for irregular workloads," in *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18)*, 2018, pp. 578–592.

[3] J. Bai, B. Gong, Y. Zhao, F. Lei, C. Yan, and Y. Gao, "Multi-scale representation learning on hypergraph for 3D shape retrieval and recognition," *IEEE Transactions on Image Processing*, vol. 30, no. 1, pp. 5327–5338, 2021.

[4] N. Beckmann and D. Sánchez, "Talus: A simple way to remove cliffs in cache performance," in *Proceedings of the 21st IEEE International Symposium on High Performance Computer Architecture (HPCA'15)*, 2015, pp. 64–75.

[5] I. O. Bercea, N. Goyal, D. G. Harris, and A. Srinivasan, "On computing maximal independent sets of hypergraphs in parallel," *ACM Transactions on Parallel Computing*, vol. 3, no. 1, pp. 5:1–5:13, 2016.

[6] C. Berge, *Hypergraphs*, 1st ed. North Holland, Amsterdam: Elsevier Science Ltd, 2013.

[7] A. E. Caldwell, A. B. Kahng, A. A. Kennings, and I. L. Markov, "Hypergraph partitioning for VLSI CAD: methodology for heuristic development, experimentation and reporting," in *Proceedings of the 36th Conference on Design Automation (DAC'99)*, 1999, pp. 349–354.

[8] L. Chen, Y. Gao, Y. Zhang, S. Wang, and B. Zheng, "Scalable hypergraph-based image retrieval and tagging system," in *Proceedings of the 34th IEEE International Conference on Data Engineering (ICDE'18)*, 2018, pp. 257–268.

[9] R. Chen, J. Shi, Y. Chen, and H. Chen, "Powerlyra: differentiated graph computation and partitioning on skewed graphs," in *Proceedings of the Tenth European Conference on Computer Systems (EuroSys'15)*, 2015, pp. 1:1–1:15.

[10] R. Dathathri, G. Gill, L. Hoang, H.-V. Dang, A. Brooks, N. Dryden, M. Snir, and K. Pingali, "Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18)*, 2018, pp. 752–768.

[11] M. T. Do, S. Yoon, B. Hooi, and K. Shin, "Structural patterns and generative models of real-world hypergraphs," in *Proceedings of the 26th ACM SIGKDD Conference on Knowledge Discovery & Data Mining (KDD'20)*, 2020, pp. 176–186.

[12] J. Doweck, "Inside intel core microarchitecture," in *Proceedings of the 2006 IEEE Hot Chips 18 Symposium (HCS'06)*, 2006, pp. 1–35.

[13] A. Ducournau and A. Bretto, "Random walks in directed hypergraphs and application to semi-supervised image segmentation," *Computer Vision and Image Understanding*, vol. 120, no. 1, pp. 91–102, 2014.

[14] P. Faldu, J. Diamond, and B. Grot, "Domain-specialized cache management for graph analytics," in *Proceedings of the 26th IEEE International Symposium on High Performance Computer Architecture (HPCA'20)*, 2020, pp. 234–248.

[15] Y. Gu, K. Yu, Z. Song, J. Qi, Z. Wang, G. Yu, and R. Zhang, "Distributed hypergraph processing using intersection graphs," *IEEE Transactions on Knowledge and Data Engineering*, vol. 1, no. 1, pp. 1–14, 2020.

[16] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*, 2016, pp. 1–13.

[17] B. Heintz and A. Chandra, "Beyond graphs: toward scalable hypergraph analysis systems," *ACM SIGMETRICS Performance Evaluation Review*, vol. 41, no. 4, pp. 94–97, 2014.

[18] B. Heintz, R. Hong, S. Singh, G. Khandelwal, C. Tesdahl, and A. Chandra, "MESH: A flexible distributed hypergraph processing system," in *Proceedings of the 2019 IEEE International Conference on Cloud Engineering (IC2E'19)*, 2019, pp. 12–22.

[19] T. Heuer, P. Sanders, and S. Schlag, "Network flow-based refinement for multilevel hypergraph partitioning," *ACM Journal of Experimental Algorithmics*, vol. 24, no. 1, pp. 2.3:1–2.3:36, 2019.

[20] W. Hsiao, J. Liu, Y. Yeh, and Y. Yang, "Compound word transformer: Learning to compose full-song music over dynamic directed hypergraphs," in *Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI'21)*, 2021, pp. 178–186.

[21] Intel, "Intel vtune profiler," https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/vtune-profiler.html#gs.59xb4m.

[22] L. Jenkins, T. H. Bhuiyan, S. Harun, C. Lightsey, D. Mentgen, S. G. Aksoy, T. Stavcnger, M. Zalewski, H. R. Medal, and C. A. Joslyn, "Chapel hypergraph library (CHGL)," in

Proceedings of the 2018 IEEE High Performance Extreme Computing Conference (HPEC'19), 2018, pp. 1–6.

[23] J. Jiang, M. Mitzenmacher, and J. Thaler, "Parallel peeling algorithms," *ACM Transactions on Parallel Computing*, vol. 3, no. 1, pp. 7:1–7:27, 2016.

[24] W. Jiang, J. Qi, J. X. Yu, J. Huang, and R. Zhang, "Hyperx: A scalable hypergraph framework," *IEEE Transactions on Knowledge and Data Engineering*, vol. 31, no. 5, pp. 909–922, 2019.

[25] I. Kabiljo, B. Karrer, M. Pundir, S. Pupyrev, A. Shalita, Y. Akhremtsev, and A. Presta, "Social hash partitioner: A scalable distributed hypergraph partitioner," *Proc. VLDB Endow.*, vol. 10, no. 11, pp. 1418–1429, 2017.

[26] J. Kunegis, "KONECT: the koblenz network collection," in *Proceedings of the 22nd International World Wide Web Conference (WWW'13)*, 2013, pp. 1343–1350.

[27] H. Labs, "Cacti," http://www.hpl.hp.com/research/cacti/.

[28] G. Lee, J. Ko, and K. Shin, "Hypergraph motifs: Concepts, algorithms, and discoveries," *Proc. VLDB Endow.*, vol. 13, no. 11, pp. 2256–2269, 2020.

[29] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data.

[30] J. Li, J. He, and Y. Zhu, "E-tail product return prediction via hypergraph-based local graph cut," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD'18)*, 2018, pp. 519–527.

[31] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'09)*, 2009, pp. 469–480.

[32] S. Maleki, U. Agarwal, M. Burtscher, and K. Pingali, "Bipart: a parallel and deterministic hypergraph partitioner," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'21)*, 2021, pp. 161–174.

[33] Micron, "DDR3L," https://www.micron.com/products/dram/.

[34] A. Mukkara, N. Beckmann, M. Abeydeera, X. Ma, and D. Sánchez, "Exploiting locality in graph analytics through hardware-accelerated traversal scheduling," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'18)*, 2018, pp. 1–14.

[35] A. Mukkara, N. Beckmann, and D. Sánchez, "PHI: architectural support for synchronization- and bandwidth-efficient commutative scatter updates," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'19)*, 2019, pp. 1009–1022.

[36] L. R. Nielsen, K. A. Andersen, and D. Pretolani, "Finding the K shortest hyperpaths," *Computers and Operations Research*, vol. 32, pp. 1477–1497, 2005.

[37] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk, "Energy efficient architecture for graph analytics accelerators," in *Proceedings of the 43rd ACM/IEEE Annual International Symposium on Computer Architecture (ISCA'16)*, 2016, pp. 166–177.

[38] R. Puzis, M. Purohit, and V. S. Subrahmanian, "Betweenness computation in the single graph representation of hypergraphs," *Social Networks*, vol. 35, no. 4, pp. 561–572, 2013.

[39] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*, 2013, pp. 472–488.

[40] D. Sánchez and C. Kozyrakis, "Zsim: fast and accurate microarchitectural simulation of thousand-core systems," in *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA'13)*, 2013, pp. 475–486.

[41] J. Shun, "Practical parallel hypergraph algorithms," in *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'20)*, 2020, pp. 232–249.

[42] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP'13)*, 2013, pp. 135–146.

[43] J. E. Smith, "Decoupled access/execute computer architectures," in *Proceedings of the 9th International Symposium on Computer Architecture (ISCA'82)*, 1982, pp. 112–119.

[44] Y. Takai, A. Miyauchi, M. Ikeda, and Y. Yoshida, "Hypergraph clustering based on pagerank," in *Proceedings of the 26th ACM SIGKDD Conference on Knowledge Discovery & Data Mining (KDD'20)*, 2020, pp. 1970–1978.

[45] Q. Wang, L. Zheng, Y. Huang, P. Yao, C. Gui, X. Liao, H. Jin, W. Jiang, and F. Mao, "GraSU: A fast graph update library for FPGA-based dynamic graph processing," in *Proceedings of the 2021 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA'21)*, 2021, pp. 149–159.

[46] Q. Wang, L. Zheng, J. Zhao, X. Liao, H. Jin, and J. Xue, "A conflict-free scheduler for high-performance graph processing on multi-pipeline fpgas," *ACM Transactions on Architecture and Code Optimization*, vol. 17, no. 2, pp. 14:1–14:26, 2020.

[47] X. Xia, H. Yin, J. Yu, Q. Wang, L. Cui, and X. Zhang, "Self-supervised hypergraph convolutional networks for session-based recommendation," in *Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI'21)*, 2021, pp. 4503–4511.

[48] J. Z. Yan, C. Chu, and W. Mak, "Safechoice: A novel approach to hypergraph clustering for wirelength-driven placement," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 7, pp. 1020–1033, 2011.

[49] M. Yan, X. Hu, S. Li, A. Basak, H. Li, X. Ma, I. Akgun, Y. Feng, P. Gu, L. Deng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, "Alleviating irregularity in graph analytics acceleration: a hardware/software co-design approach," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'19)*, 2019, pp. 615–628.

[50] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, "IMP: indirect memory prefetcher," in *Proceedings of the 48th International Symposium on Microarchitecture (MICRO'15)*, 2015, pp. 178–190.

[51] D. Zhang, X. Ma, M. Thomson, and D. Chiou, "Minnow: Lightweight offload engines for worklist management and worklist-directed prefetching," in *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18)*, 2018, pp. 593–607.

[52] X. Zhu, W. Chen, W. Zheng, and X. Ma, "Gemini: A computation-centric distributed graph processing system," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, 2016, pp. 301–316.