Flip: Data-Centric Edge CGRA Accelerator

DAN WU, National University of Singapore, Singapore

PENG CHEN*, Chongqing University of Posts and Telecommunications, China

THILINI KAUSHALYA BANDARA, National University of Singapore, Singapore

ZHAOYING LI, National University of Singapore, Singapore

TULIKA MITRA, National University of Singapore, Singapore

Coarse-Grained Reconfigurable Arrays (CGRA) are promising edge accelerators due to the outstanding balance in flexibility, performance, and energy efficiency. Classic CGRAs statically map compute operations onto the processing elements (PE) and route the data dependencies among the operations through the Network-on-Chip. However, CGRAs are designed for fine-grained static instruction-level parallelism and struggle to accelerate applications with dynamic and irregular data-level parallelism, such as graph processing. To address this limitation, we present Flip, a novel accelerator that enhances traditional CGRA architectures to boost the performance of graph applications. Flip retains the classic CGRA execution model while introducing a special data-centric mode for efficient graph processing. Specifically, it leverages the inherent data parallelism of graph algorithms by mapping graph vertices onto PEs rather than the operations, and supporting dynamic routing of temporary data according to the runtime evolution of the graph frontier. Experimental results demonstrate that Flip achieves up to 36× speedup with merely 19% more area compared to classic CGRAs. Compared to state-of-the-art large-scale graph processors, Flip has similar energy efficiency and 2.2× better area efficiency at a much-reduced power/area budget.

CCS Concepts: • **Hardware → Hardware accelerators**; **Hardware-software codesign**.

Additional Key Words and Phrases: Coarse-grained Reconfigurable Array, Graph Processing, Accelerator

Coarse-Grained Reconfigurable Arrays (CGRA) are a prominent approach among high-performance low-power edge accelerators due to their excellent balance between flexibility, performance, and energy efficiency. Existing CGRAs employ a static *operation-centric* approach in which the operations within the dataflow graph (DFG) of the loop kernel are spatially mapped onto the processing element (PE) array, and the data dependencies among the operations are routed through network on chip (NoC) [28, 30, 33, 40]. In other words, CGRAs rely on instruction-level parallelism to accelerate loop kernels. Furthermore, the placement of operations and data routing paths are statically determined at

---

*Work conducted while at National University of Singapore. Currently affiliated with Chongqing University of Posts and Telecommunications.

Authors' addresses: Dan Wu, National University of Singapore, Singapore, Singapore, danwu20@comp.nus.edu.sg; Peng Chen, Chongqing University of Posts and Telecommunications, Chongqing, China, chenpeng@cqupt.edu.cn; Thilini Kaushalya Bandara, National University of Singapore, Singapore, Singapore, thilini@comp.nus.edu.sg; Zhaoying Li, National University of Singapore, Singapore, Singapore, zhaoying@comp.nus.edu.sg; Tulika Mitra, National University of Singapore, Singapore, Singapore, tulika@comp.nus.edu.sg.

compile time. The classic CGRA execution model exhibits the following key characteristics: (1) Mainly, the PE array exploits instruction-level parallelism, and it does not utilize data-level parallelism if there exist data dependencies in the input data; (2) Data is stored in the scratchpad memory (SPM), and each iteration necessitates regular SPM access; (3) The static nature of the fine-grained operation mapping requires the majority of instruction-level parallelism to be known at compile time to facilitate pipelining [15, 50]. Thus, CGRAs are suitable for kernels with high instruction-level parallelism and regular low-intensity memory access.

However, CGRAs are not well-suited for applications with irregular memory access, poor data locality, runtime control, and data dependencies that do not expose parallelism at compile time and inherently conflict with static mapping. Accelerating these irregular applications is crucial for the broader adoption of CGRA technology. Graph processing, a vital application at the edge (e.g., to support indoor or outdoor navigation), features intensive, irregular memory access and light computation with minimal instruction-level parallelism [16, 34]. Consequently, graph applications exhibit poor performance on CGRAs. Moreover, although CGRAs are capable of exploiting fine-grained instruction-level parallelism, the constraints of static mapping make it challenging to accelerate applications characterized by inherent coarse-grained dynamic data parallelism. Graph algorithms typically exhibit natural parallelism across vertices in the dynamically evolving graph frontier, i.e., vertices within the frontier can be processed in parallel. To harness this coarse-grained parallelism, graph accelerators iteratively dispatch tasks (e.g., active vertices) to different processing units, synchronize, and update the task queue (e.g., graph frontier). This complex schedule and dispatch mechanism is suitable for large graph processing accelerators [10, 36, 41, 44, 63] with significant area and power budgets. Although graph processing is widely used in edge devices, such as pathfinding in network devices [17] and navigation in small robots, tiny graph accelerators that can operate within limited power budgets are lacking.

We present FLIP, a tiny CGRA accelerator capable of supporting both the traditional *operation-centric* execution model for compute-heavy regular kernels and a novel *data-centric* paradigm that harnesses the natural coarse-grained parallelism of the graph algorithms. The data-centric paradigm employs a vertex-centric programming model, wherein the compute function of a vertex receives the updated vertex value from a neighbor, calculates a new vertex value, and distributes the updated value to neighboring vertices [34]. FLIP overturns the operation-centric paradigm by distributing data graph vertices to the PEs and routing the edges rather than mapping the operations corresponding to the loop kernel. As the vertex processing code is typically relatively short and the same for all the vertices, this code is stored in each PE and executed sequentially for a vertex. Once a vertex's processing is completed, the current PE sends the intermediate result to PEs where the neighboring vertices are mapped. The sent intermediate results then concurrently trigger the processing in different PEs. The execution thus follows the parallelism unearthed by the frontier of active vertices instead of struggling to find the meager parallelism among the operations within the computation for a single vertex.

Fig. 1 illustrates the differences between the operation-centric and data-centric execution models for the Breadth-First Search (BFS) algorithm. The data graph to be processed is shown in Fig. 1(a). Assuming BFS is invoked with V_1 as the source vertex, Fig. 1(b) displays the traditional operation-centric execution model in CGRA. In this model, operations to process one edge of a vertex are spatially mapped on the PE array, along with the routing of the data dependencies among the operations. Since concurrent processing of multiple vertices can potentially cause data conflict due to non-atomic read/write pairs, this approach can only process one vertex at a time but takes advantage of parallelism among the operations, which is quite limited in this scenario. In contrast, the data-centric execution model of FLIP, shown in Fig. 1(c), statically maps data vertices onto the PE array. Execution is triggered by the PE containing the source vertex V_1, where the code for processing the vertex is executed sequentially. The completion of processing V_1 triggers
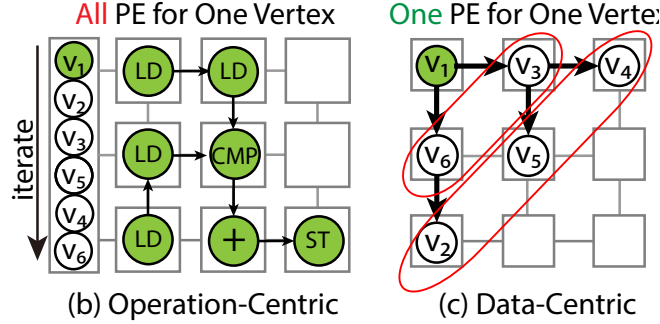
Fig. 1. Operation-centric vs. Data-centric execution modes. Operation-centric mode maps the operations on the PEs and processes the vertices stored in scratchpad memory iteratively. Our proposed data-centric mode maps the vertices on different PEs and processes them in parallel with identical code.

Table 1. Comparison among CGRA-based accelerators.

	PolyGraph [10]	Fifer [36]	HyCUBE [52]	RipTide [15]	FLIP (this work)
Graph Application Performance	✔	✔	✗	✗	✔
General Application Performance	✗	✔	✔	✗	✔
Power Efficient	✗	✗	✔	✔	✔
Area Efficient	✗	✗	✔	✔	✔
Number of PEs	16×5×4	16×16×5	4×4	6×6	8×8
CGRA mode	Op-Centric	Op-Centric	Op-Centric	Op-Centric	**Data-Centric&Op-Centric**

parallel processing of V_3 and V_6, and their completion, in turn, triggers parallel processing of the new frontier (V_2, V_4, V_5). Consequently, data-centric execution exploits coarse-grained dynamic data parallelism.

Achieving the data-centric execution model on a low-power CGRA accelerator is not straightforward. Different graph algorithms and even invocations of the same algorithm with different starting points lead to different parallelism and data routing patterns. In FLIP, we augment the underlying classic CGRA architecture to support dynamic routing of the intermediate data along the graph edges (e.g., V_1 to V_3 and V_6). We also design a compiler that can map the vertices onto the PE array to expose parallelism among the frontier vertices (e.g., V_2, V_4, V_5 should preferably be mapped to distinct PEs) while ensuring short routing lengths for the edges. Lastly, for larger graphs that exceed the capacity of the on-chip memory, we partition the graph and dynamically swap partitions in and out during runtime. To support dynamic graphs with changing vertex or edge attributes, we update these attributes while the corresponding partition is swapped out to the main memory, utilizing a general-purpose processor (e.g., micro-controller unit) for the updates.

To the best of our knowledge, FLIP is the only programmable accelerator under a limited power budget that supports graph processing at the edge while also accelerating traditional compute-intensive kernels that are a natural fit for CGRA architecture. Table 1 presents a qualitative comparison of FLIP with recent CGRA-based accelerators. Data-center scale advanced graph accelerators, PolyGraph [10] and Fifer [36] adopt multi-core graph processor design with complex components like scheduler, control core, caches, and a complete CGRA fabric within each core. In these hierarchical architectures, the CGRA serves only as a low-power substitute for the compute core, offering instruction-level parallelism. The higher-level core array captures coarse-grained data parallelism thanks to the complex control components. In the realm of low-power devices, such a complex system design becomes infeasible due to power constraints. With a

single and simple CGRA fabric, HyCUBE [52] and RipTide [15] adopt the operation-centric paradigm to support diverse applications and thus perform poorly on graph applications.

Experimental evaluation shows that: (1) The proposed data-centric model can achieve high data-level parallelism compared to the conventional operation-centric approach. (2) On graph processing tasks, FLIP achieves up to 36× performance speedup with only a 19% increase in area compared to the operation-centric CGRA accelerator. (3) Compared with an advanced graph processor, PolyGraph[10], FLIP achieves similar performance per watt and 2.2× performance per area at only 1.1% and 0.5% area and power budget, respectively.

1 BACKGROUND AND MOTIVATION

1.1 Graph Processing at the Edge

Unlike large-scale graphs, graphs at the edge exhibit distinct characteristics. We leverage these unique features of graph processing at the edge, as opposed to processing large graph data in the cloud, to achieve data-centric CGRA acceleration at the edge.

Graph algorithms can be expressed through vertex-centric programming models [16]. In this model, a concise function is repetitively executed for each vertex in the graph [6]. In other words, vertex data stored on different PEs can be processed with the same small code snippet, requiring minimal instruction memory per PE. The concept is akin to the SPMD (single program, multiple data) computing model employed by GPUs, except for the rich and dynamic dependencies among the vertices, contrasting with the massive regular data parallelism in most GPU kernels.

Graph processing at the edge typically employs small graphs, such as neighborhood road network for navigation [17]. In these cases, graph data can be stored entirely on-chip. The graphs usually have limited in and out degrees, resulting in a small and balanced memory requirement for maintaining the edge data corresponding to each vertex. Utilizing only the on-chip distributed memory within its PE array, FLIP can accommodate up to 256 vertices, roughly corresponding to a 2.5 km^2 area in downtown San Francisco [26]. With a centralized memory unit, like the main memory of a micro-controller unit or DRAM, FLIP can also handle larger problem sizes through data swapping from the centralized memory.

Most graph algorithms maintain a static graph structure, modifying only the attributes of vertices and edges. In this scenario, the routing configurations for most edges remain static. So, we can map a graph once and run multiple queries on the same data, or even run multiple applications on it by only replacing the code snippet on PEs. FLIP also supports efficient attribute changing for dynamic graphs. Without the need for recompilation or bulky data movement, only the graph partitions with the changed vertices/edges are swapped out and updated.

1.2 Motivating Example

Conventional operation-centric CGRAs accelerate application execution through instruction-level parallelism. Consider the example depicted in Fig. 2(a), where the inner loop kernel (line 5-7) of Dijkstra's algorithm for the single source shortest path (SSSP) is mapped onto a 2×2 conventional CGRA. The compiler first converts the loop kernel to DFG representation (①). A toy DFG is used for illustration. Then, it schedules the DFG onto the time-extended routing resource graph in a modulo fashion (②). The schedule repeats every two cycles, which is the Initiation Interval (II). This modulo schedule is loaded into the PE local configuration memory, and the graph data is loaded to the SPMs before execution. Detailed execution steps can be observed in Fig. 2(c). Each execution of the loop iteration needs to load (store) data from (to) SPM. For the sake of clarity and without loss of generality, in this example, we assume the edge weight to be 1, which means that each edge needs to be processed only once.
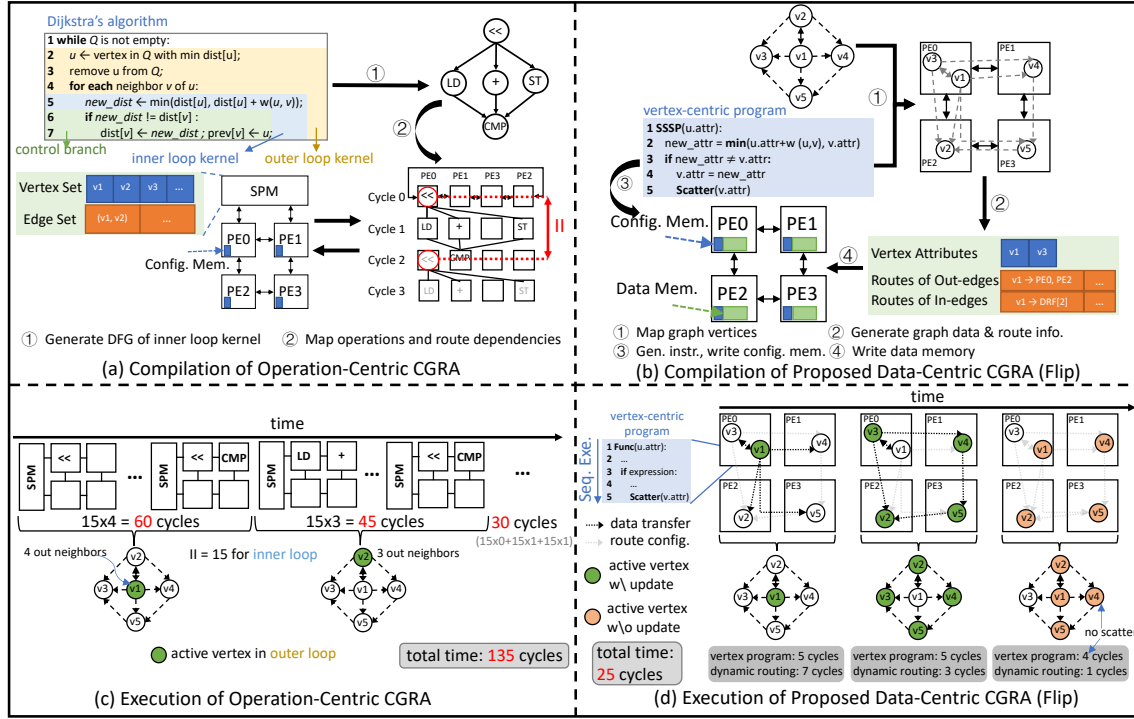
Fig. 2. Compilation and execution for operation-centric CGRA and data-centric CGRA, taking SSSP problem as an example. Data-centric CGRA benefits from data-level parallelism and outperforms operation-centric CGRA, which suffers from low instruction-level parallelism in graph algorithms.
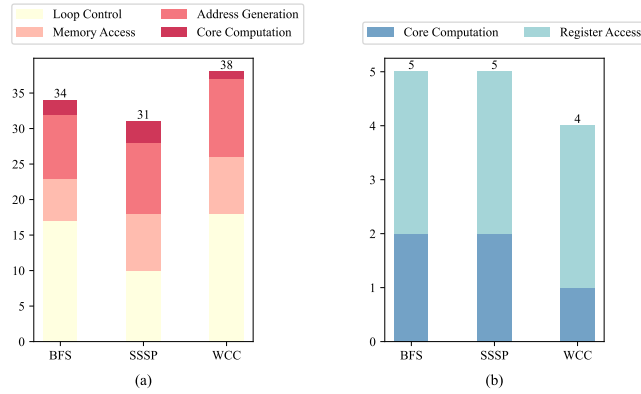
Fig. 3. Number of operations for (a) operation-centric and (b) data-centric CGRA. *Memory Access* counts only graph data access.

The operation-centric model encounters several limitations when applied to graph processing tasks. The intensive and irregular memory accesses demand considerable amount of operations for address calculation and data fetching. As shown in Fig. 3(a), 20% of the DFG operations are used to access graph data and 30% for memory address generation.

Fig. 4. Speed up with different degrees of unrolling on operation-centric CGRA for BFS on road networks. The execution time smoothes when the unroll degree reaches three and achieves only 1.3× speedup. Further unrolling will result in compilation failure because of the exponentially growing compilation time.

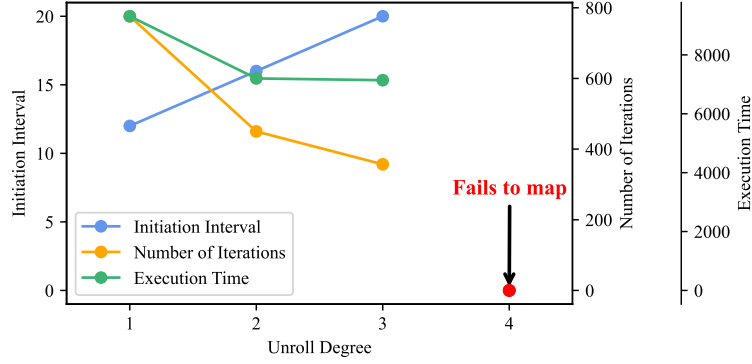Furthermore, the irregular memory access patterns result in a significant number of memory bank conflicts, further limiting memory-level parallelism. Additionally, without complicated hardware support, static scheduled CGRAs cannot support parallel graph applications [5, 7, 8] designed for multi-core processors with coarser parallelism (per-core graph partition) due to the cost of dynamic communications and synchronization. Additionally, fine-grained parallelization techniques, such as loop unrolling, fail to achieve the anticipated speedup due to the dependencies among the vertices and varying number of neighbors, as shown in Fig. 4. The increased size of DFGs after unrolling also leads to compiler crashes due to the exponentially increasing complexity of the mapping problem. Lastly, loop control operations identifying neighboring vertices constitute a substantial fraction of the total operations, as shown in Fig. 3(a). Overall, it takes 15×9=135 cycles to process all nine edges in this example, and only one vertex can be processed at a time.

FLIP effectively addresses these issues by adopting the data-centric paradigm. As shown in Fig. 2(b), instead of utilizing the entire PE array for inner loop computation, corresponding to processing one edge, we directly map the graph to the PE array. Each PE stores the vertex attributes mapped to it, attributes of the incoming edges, and the destination PEs of outgoing edges. By mapping graph data to PEs, the number of instructions is minimized, as shown in Fig. 3(b). Since data needed by one PE is stored locally or passed from other PEs through the NoC, address calculation is no longer required. Moreover, each PE executes the vertex-centric program independently, so instructions for loop control are eliminated. For larger graphs, different parts (multiple vertices) can be mapped onto the same PE through time-multiplexing and swapped in and out at runtime.

A PE executes the code snippet (5 cycles for SSSP) sequentially when triggered by an incoming edge, then routes the data (updated vertex's attribute) through the NoC. In the data-centric model, an application starts at a designated vertex (e.g., source V_1 in SSSP), which is unknown at compile time. The intermediate results (e.g., distance from V_1 to the current vertex) are sent to the neighboring vertices, which are then activated. In Fig. 2(d), the neighbors of vertex V_1 (V_2, V_3, V_4, V_5) are mapped to different PEs, enabling parallel processing. A PE only needs information from the incoming packet and the local data during the processing. In this example, it takes 5+7=12 cycles to process V_1 and scatter data to its neighbors. Subsequently, it takes another 7 cycles to concurrently process and scatter V_2, V_3, V_4, V_5. Finally, vertex V_1, V_2, V_4, and V_5 are activated, but the program stops early in 4 cycles as there is no attribute update

```
Apply(u.attr):
    return min(v.attr, u.attr + w(u,v))
SSSP(u.attr):
    new_attr = Apply(u.attr)
    if new_attr ≠ v.attr:
        v.attr = new_attr
        Scatter v.attr
```

Fig. 5. Implementation of SSSP in the vertex-centric programming model. Scatter instruction triggers the message passing to destination vertices.

and thus nothing to scatter. It takes a total of 25 cycles to process these 5 vertices, offering a substantial reduction in computational time compared to 135 cycles in the operation-centric model.

2 FLIP OVERVIEW

We design Flip to enable efficient graph processing on CGRA accelerators. *Note that the underlying architecture supports operation-centric execution mode, which is not discussed as it is similar to classic CGRA.* We detail the data-centric mode for graph processing. Flip uses the vertex-centric programming model where a vertex is processed following the general sequence of receiving updated value from its neighbor, calculating new values (*Apply()*), and distributing the updated values to neighboring vertices (*Scatter*)[34]. Fig. 5 shows the example code for SSSP written in the vertex-centric programming model. Flip takes the graph as input and maps the graph vertices onto the PE array in a many-to-one manner. Similar to some graph processors [2, 41, 60], Flip follows a graph frontier approach with the vertex-centric programming model. The key difference is that prior graph processors use programs to schedule and control the computation in each processing unit with data fetched from memory at runtime, while Flip statically places the data onto the PEs and uses the frontier data to trigger processing in each PE directly. This reduces control overhead tremendously and enables asynchronous distributed graph processing. The graph application execution is triggered from a specific vertex on a PE (e.g., the current location in navigation systems) and progresses to other vertices.

We also designed an efficient compiler to map the graph data onto the PE array and generate the routing information for graph edges. The compiler combines locality and parallelism to maximize graph processing performance. The graph is mapped offline. Before execution, a configuration is generated containing vertex properties, instructions, and routing information. This configuration is distributed to the PE array from the host. The user can specify the starting vertex at runtime, providing flexibility in execution. Flip also allows for updating graph attributes dynamically during execution through a data-swapping mechanism.

3 FLIP ARCHITECTURE

Fig. 6 shows the proposed Flip architecture. Flip fabric contains a PE array (8×8 in our prototype) connected by a mesh network. The complete Flip system also contains on-chip SPM (16KB in 8 banks), off-chip memory (256KB), and a scalar core as the host processor. To support data-centric mode, Flip uses on-chip cross-PE distributed memory (64×260B = 16KB) for graph data storage and packet buffering, and a NoC supporting dynamic routing.

In each cycle, a packet in one input buffer is selected by the arbiter. The offset subtractor and comparator then decide the routing direction for this packet. After a packet reaches the destination PE, the carried slice id is compared with the stored slice id in PE. If they are the same, the content carried by the packet will be later used for computation. Otherwise, this packet will be pushed to the memory buffer and sent to SPM. For an updated vertex to be scattered, the
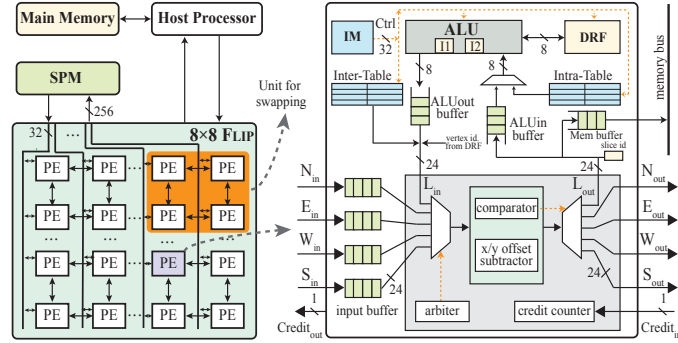
Fig. 6. FLIP: CGRA enhanced with *data-centric* mode. The arbiter, comparator, and x/y offset subtractor are used for dynamic routing. Once a packet reaches the destination PE, the Intra-Table is used to find the vertex location in the Data Register File (DRF) and the edge weight. To scatter a vertex's new attribute, the value is pushed to the ALUout buffer and will be later packed with the x/y offset of destination PE(s) in Inter-Table.

updated vertex attribute is pushed to the ALUout buffer, packed with the inter-PE routing information in Inter-Table. Finally, the packet is pushed to the NoC.

For handling larger problem sizes, FLIP provides efficient runtime data-swapping. To take advantage of data locality while avoiding straggler problems, we use non-overlapping 2×2 PE clusters as the basic units of data swapping. The graph partition mapped onto a PE cluster is called a *slice*. Multiple slices can be mapped onto the same PE cluster, identified by a unique id $slice_{id}$. FLIP employs the same Instruction Set Architecture (ISA) as classic CGRAs, allowing the underlying hardware to be repurposed for static mapping of non-graph-based applications by simply deactivating dynamic routing.

3.1 PE Memory

Distinct from conventional CGRAs, FLIP has larger memory per PE to handle contention at runtime and hold the data originally stored in the multi-bank SPM. One PE in FLIP consists of seven storage components for different functionalities. All buffers in FLIP are FIFO queues.

Instruction Memory (IM) is a register file holding the instruction set of the vertex-centric program. The code snippet is identical for all PEs.

Data Register File (DRF) stores the properties (e.g., vertex index, BFS level) of the vertices that are mapped onto the current PE. The properties of the vertices can be updated during execution. Since each DRF holds 4 registers, we allow at most 4 graph vertices to be stored in each PE.

Slice ID Register is an 8-bit register storing the $slice_{id}$ of currently loaded subgraph (slice).

ALUout Buffer stores packets in the format (id_u, $attribute_u$), where u is the current vertex with an updated attribute and needs to send the value to its neighboring vertices for the update. When the packet is sent out to the adjacent PE, we add an $offset_v$ item indicating the number of hops in the x and y axis to the PE that stores the destination vertex v.

Input Buffers cache the incoming packets from adjacent PEs. We have one input buffer per input port. Without

contention-free routing, we use larger input buffers to store multiple packets and avoid potential deadlock or packet loss. The packets in the input buffer have the format (id_u, $offset_v$, $attribute_u$, $slice_{id}^v$).

ALUin Buffer holds the packets forwarded by the routing logic till ALU is ready to receive new packets. Each incoming packet is processed and updated with edge attributes before being fed to ALU.

Memory Buffer caches packets whose destination vertex is currently off-chip. To check whether the destination vertex v is loaded, the $slice_{id}^v$ inside the packet is compared with the value in the Slice ID Register. Then, the packet will be pushed into either the ALUin buffer or the Memory buffer according to the comparison.

Inter-Table stores the *inter-PE* routing information, identifying the PEs that house the destination vertices for each vertex within this PE (Sec. 3.2.1).

Intra-Table maintains the *intra-PE* routing information and incorporates edge weights, specifying the target register(s) storing the destination vertex (vertices) for packets destined for this PE as well the weight for this edge (Sec. 3.2.2).

3.2 Dynamic and Contention-Aware Routing

The graphs we are processing at the edge have limited in-/out-degree, so the PEs have a balanced communication workload. Thus, there is limited benefit in supporting routes with an arbitrary number of turns that are useful to bypass the busy area. As a result, Flip employs YX dimension-ordered routing [13, 20], a scheme that minimizes routing complexity and conserves memory. We set two routing tables in each PE, storing the graph edge and routing information compactly, as shown in Fig. 7. With these two routing tables, a PE can dynamically select the corresponding routing configuration via finding offsets to destination PEs and identifying the address of the destination vertex inside the PE. The addressing process is thus split into two stages: inter-PE and intra-PE addressing. Inter-PE addressing aims to find the target PE where the neighboring vertices are located, while intra-PE addressing is to find the specific vertex among multiple vertices inside the current PE.

3.2.1 Inter-PE Addressing. An Inter-Table is established inside each PE for inter-PE addressing. As shown in Fig. 7, each Inter-Table entry indicates the offsets to the target PE, consisting of source vertex index src_{id}, x_offset in the x dimension, y_offset in the y dimension and $slice_{id}$ of the slice containing destination vertex. x_offset and y_offset have 4 bits each, of which the first bit indicates the transmission direction (i.e., 0 for negative direction, and 1 for positive direction), and the subsequent 3 bits (i.e., up to 7 hops) refer to the number of hops to the destination along that dimension. For example, in Inter-Table of Fig. 7, the first source vertex index is 3 and the corresponding $x\_offset = \{0, 100\}$ and $y\_offset = \{1, 010\}$. This means that the packet generated by vertex 3 will be sent out along the $+y$ dimension with 2 (=0b010) hops, then along the $-x$ dimension with 4 (=0b100) hops. The packet carries the routing information of x_offset and y_offset values to find the target PE. After each hop, the corresponding offset is decremented by one. The packet ultimately reaches the destination PE when the last three bits of x_offset and y_offset become zero.

To accelerate the search within Inter-Table, we organize the table entries with linked lists: all the entries with the same source vertex index are organized as a list. The head entry for a linked list can be found sequentially in the table head, which then points to the second entry with the same vertex index. The pointer field (i.e., next) of its last entry is NULL. The list length is equal to the outgoing degree of the current vertex, which is short since Flip is proposed for low-degree graphs. The four head entries, one for each vertex of current PE, with the unique source vertex index, are placed at the four headmost positions of Inter-Table for fast search.

3.2.2 Intra-PE Addressing. When a packet reaches the target PE and its $slice_{id}$ matches the value in the Slice ID Register, intra-PE addressing helps find the specific vertex among the multiple vertices inside the target PE. An Intra-Table entry

src$_{id}$	x_offset	y_offset	slice$_{id}$	next
3	0,100b	1,010b	7	•
4	0,110b	1,001b	9	•
...				
3	1,101b	0,101b	1	•

Inter-Table

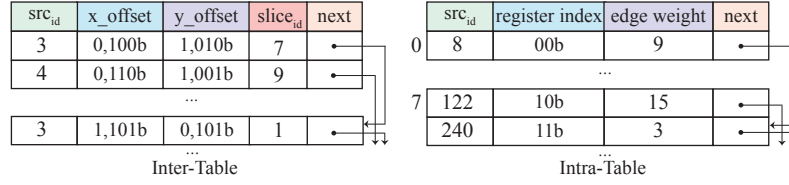|	src$_{id}$	register index	edge weight	next
0	8	00b	9	•
	...			
7	122	10b	15	•
	240	11b	3	•

Intra-Table

Fig. 7. Example of routing table in FLIP.

indicates destination vertex and incoming edge attributes, consisting of the index src_{id} of source vertex of the incoming edge, register index for destination vertex in DRF of current PE, and edge weight for the current edge. Entry of the source vertex index in Intra-Table has 8 bits, used to identify the destination vertex and the corresponding edge weight.

To accelerate the search within Intra-Table, all the entries are organized as multiple linked lists similar to Inter-Table. The source vertex index of entries in the same list has the same hash value returned by a hash function (i.e., $src_{id}\%8$). Before the table searching for incoming packets, the hash function is computed to find the head entry address of the corresponding list. The 8 head entries (i.e., hash value ranging from 0~7) are placed at the headmost position of Intra-Table. The entries inside the list are searched sequentially. In our experiment, it's important to note that the average length of each list is less than 2. This indicates that there is minimal overhead when searching through these lists.

3.2.3 Flow Control. Unlike conventional CGRAs that rely on static routing, FLIP dynamically generates packets. This introduces potential contentions for communication resources in the NoC, such as output ports. To effectively manage these contentions, we utilize a credit-based routing mechanism [13, 20].

3.3 Runtime Data Swapping

When the PE array cannot hold a graph due to capacity limit, the off-chip memory is used to store the whole graph, and graph partitions are brought to FLIP at runtime using SPM as a cache. When the underlying graph structure remains unchanged, data swapping can also be used to update the vertex/edge attributes by popping slices out, making modifications, and then swapping them in.

At runtime, a packet is first cached if it cannot find its destination vertex on-chip. It is later loaded and executed once the corresponding vertex is brought in. To be more specific, if the $slice_{id}$ does not match once a packet reaches the destination PE, the packet is pushed into *memory buffer* and sent to SPM. When a PE cluster becomes idle, the system initiates data swapping. We employ a cache-friendly priority scheme where the slice with the earliest pending task (a cached packet) is the first to be loaded among all slices mapped to the same PE cluster.

3.4 Mode Switching

FLIP architecture natively supports operation-centric computing. In this mode, Inter-Table and Intra-Table store the pre-determined Xbar configurations for static routing. Instructions are stored in *Instruction Memory* (IM) similar to data-centric mode. A global trigger initiates the execution of all PEs, and a global program counter is used to cycle through the configurations in all PEs in sync, as opposed to an independent program counter for each PE in data-centric mode. Components such as comparator and offset subtractor are disabled, and the SPM is used for centralized data storage.

4 FLIP COMPILER

In this section, we discuss the mapping of graph data onto the Flip PE array in the data-centric mode. We first explore the mapping algorithm when the entire graph can fit into the PE array, and subsequently introduce how to handle larger problem sizes.

4.1 Problem Formulation

The Flip compiler maps the graph vertices to PEs. Once the placement is ensured, a YX-dimension ordered route is generated for each graph edge. Since the Flip compiler assumes an unknown starting vertex with non-deterministic runtime contention, the final execution time cannot be derived during compilation. Therefore, we devise two mapping objectives that are closely correlated with the execution time: the *total routing length* and *sequentialization*.

Total Routing Length: As vertex updates are transmitted along the edges, and each edge is mapped onto a series of links on NoC, the total routing length of all graph edges determines the communication cost. The communication cost increases with longer routing lengths. Given that the latency of a single-hop routing is notably high in our contention-tolerant NoC—almost equivalent to the computation time of one packet—routing length becomes a critical factor in assessing the mapping quality.

Sequentialization: Simply clustering vertices for locality can hurt parallelism among the frontier vertices. Even with the flexibility of dynamic routing and the uncertainty of the starting vertex, there exists an inherent barrier to achieving full parallelism. Specifically, if two vertices mapped to the same PE receive incoming edges from the same vertex, they must be executed sequentially. Thus, a high-quality mapping should balance locality and parallelism.

Problem Definition: Given a graph $G(V, E)$ and a Flip instance with PE array P, the problem is to construct a many-to-one mapping $M = \{(v, p)\}$ of vertices in G to PEs in P.

- Each vertex $v \in V$ should be mapped to one PE $p \in P$.
- Each PE $p \in P$ can have at most n graph vertices mapped onto it. In the current design, $n = 4$.
- Minimize *total routing length* and *sequentialization*.

Algorithm 1 Flip Mapping Algorithm

Input: Graph $G(V, E)$ and PE array P
Output: Mapping $M = \{v : p\}$ for $v \in V, p \in P$
1: P = P.replicate($\lceil \frac{|V|}{P.capacity} \rceil$);
2: $M = \{v_c : p_c\}$;
3: M = Beam_Search(M);   ▷ Generate initial mapping
4: **while** M is not stable **do**   ▷ Local optimization
5:     p = random(P); P_p = Neighbors of p;
6:     V_p = vertices mapped on p; V_P = vertices mapped on P_p;
7:     ψ = combination(V_p, V_P);
8:     best_pair = M.**estimate**(ψ, t_h);
9:     If best_pair.cost>0, swap the placement of best_pair in M;
10: **end while**

4.2 Mapping Algorithm

Algorithm 1 presents the pseudo-code of Flip mapping algorithm. It adopts a two-phase strategy to generate high-quality mappings. The first phase is dedicated to producing an initial mapping that is solely optimized for routing length,

utilizing the beam search algorithm [42]. The second phase optimizes the mapping locally to reduce sequentialization and improve parallelism, guided by a *execution time estimation model* (Algorithm 2).

4.2.1 Initial Mapping Generation. In the first step, a mapping is generated using beam search with beam width k=10. Beam search serves as a heuristic method that navigates through the search tree. It sorts nodes at each level using a pre-defined function and explores only the k most promising nodes. In beam search, we only use the primary objective, *total routing length*, denoted as $f(M)$. For an incomplete mapping M' where just a subset of V is mapped, only edges with both ends mapped are evaluated in $f(M')$.

Initially, the vertex v_c, which serves as the graph center with minimum eccentricity, is positioned at p_c, the central point of the PE array. This constitutes the root of the search tree. Formally, we denote the root as $S(0, M, V_{can}, P_{can})$ containing: layer in the search tree which is equal to the number of vertices mapped minus 1, incomplete mapping M={v_c:p_c}, candidate vertices V_{can}={neighbors of v_c} to map for the next iteration, and candidate PEs P_{can}={p_c and PEs next to p_c} available for mapping vertices in the next iteration. At level i of the tree, it expands all nodes at the current level by generating their successors. A successor $S(i+1, M', V'_{can}, P'_{can})$ of a node $S(i, M, V_{can}, P_{can})$ is obtained by (a) assigning a vertex v in V_{can} to a PE p in P_{can}, (b) updating the frontier-like candidate vertex set and candidate PE set. In each layer of the search tree, only the top k nodes with the lowest $f(M)$ are retained for further exploration.

4.2.2 Local Optimization to Balance Locality and Parallelism. As beam search is a non-optimal heuristic method, to improve data locality further while avoiding sequentialization, we iteratively optimize the mapping by applying local changes guided by the *execution time estimation model* to reconcile the two conflicting goals (Algorithm 2). The model takes vertex pair s ψ and the whole graph mapping $M = \{v : p | v \in V, p \in P\}$ as its inputs. In each iteration, a vertex pair is selected, and we estimate the time to pass through the one-hop neighborhoods of these two vertices, called the *partial execution time*. Particularly for *sequentialization*, if vertices $(v_0, \ldots, v_n)$ in a PE p have incoming edges from the same vertex u, we denote $(v_0, \ldots, v_n)$ as the *collision set* and edges $\{(u,v)|v \in (v_0, \ldots, v_n)\}$ as *congested edges*.

In each iteration, a vertex pair (u, v) is evaluated for swapping (line 1). A vertex pair consists of a vertex in the current PE and another in a neighboring PE. To evaluate the benefit of swapping, we estimate partial execution time before and after swapping. The partial execution time is estimated by summing up the estimated execution time of each connected edge of these two vertices (line 2). For one edge, the execution time t^e is estimated by three parts: the intra-PE addressing time (packet transmission) t_{trans}, vertex program execution time t_{exe}, and the intra-PE addressing time (table searching) t_{tab}. The packet transmission time is calculated by multiplying the time per hop and the number of hops (line 3). When these two vertices are assigned to the same PE cluster but different slices, which means they cannot be loaded on-chip at the same time, an extra overhead ϵ is added (line 4). When the edge is one of the *congested edges*, vertices need to be sequentially executed, as shown in Fig. 8. In this case, we estimate the worst-case time, assuming that this vertex is the last one in sequential processing (line 6). Otherwise, the partial execution time is estimated by adding up the packet transmission time, table searching time, and vertex program execution time (line 8). The same process is repeated to get the partial execution time, assuming a swapped location of the vertex pair (line 10). Ultimately, we swap the vertex pair that yields the greatest reduction in partial execution time (line 13).
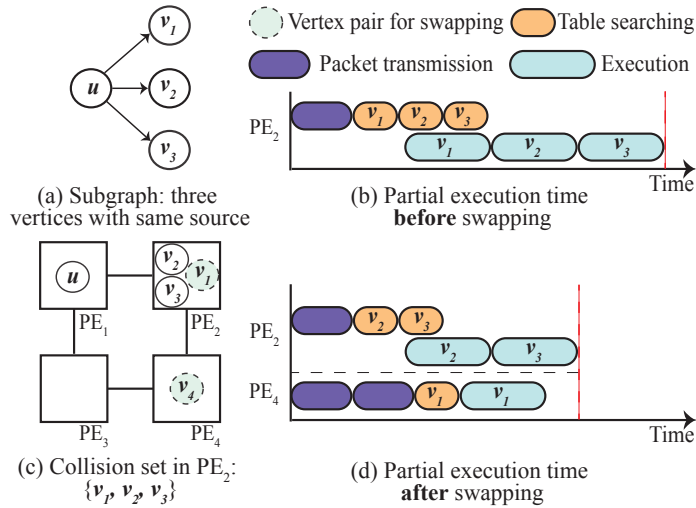
4.3 Data Layout

Besides mapping, an optimized data layout in Inter-Table can also reduce execution time by changing the data access pattern. The philosophy for efficient data layout is **"The Farthest, The First"**. Suppose one vertex u has three neighbors, each placed in a different PE, as illustrated in Fig. 9 (a, c). After u updates its value, multiple packets heading to different

---

Algorithm 2 FLIP Estimation Model

Input: Mapping $M=\{v:p\}$, vertices pairs ψ, est. time for 1 hop trans. t_h
Output: lowest-cost vertices pair (u',v') and its cost c' for swapping

 1: **for** (u,v) in ψ **do**
 2:    **for** e in u.edges $\cup$ v.edges **do**
 3:       t_{trans} = e.length_of_route $\times$ t_h;   ▷ Transmission time
 4:       t_{trans} += isSameClusterDiffChip(e.srcPE, e.destPE)? ϵ:0;
 5:       **if** e in *congested edges* **then**
 6:          t^e = maximumTime(*congested edges*)
 7:       **else**
 8:          t^e = t_{trans} + t^e_{tab} +t_{exe};   ▷ Add Table search and execution time
 9:       **end if**
10:       repeat to get t'^e, supposing swapped location of u and v.
11:       c = t^e -t'^e
12:    **end for**
13:    update (u',v') and c' if $c > c'$
14: **end for**

(a) Subgraph: three vertices with same source

(b) Partial execution time **before** swapping

(c) Collision set in PE$_2$: $\{v_1, v_2, v_3\}$

(d) Partial execution time **after** swapping

Fig. 8. Illustrative example of *partial execution time* estimation for PE with *collision*.

destinations are issued sequentially with the order of entries in Inter-Table. Without sorting the Inter-Table, the worst case may happen when the packet heading to the farthest destination is the last one generated (see Fig. 9 (b)). Meanwhile, with sorting, these three vertices can finish processing with less time (see Fig. 9 (d)). An inefficient data layout will increase the overall processing time for all neighbors of a vertex and lead to under-utilization of the PE array. Although asynchronous execution mitigates the straggler problem commonly seen in bulk synchronous parallel execution [9, 32], the routing to the farthest destination is highly likely to be a critical path in the whole execution, and thus optimization is necessary.
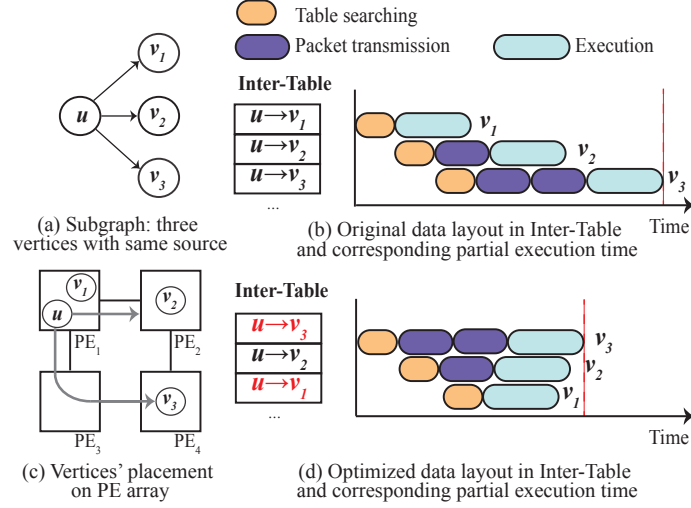
Fig. 9. Illustrative example of farthest-one-first strategy.

Table 2. Comparison of FLIP with CGRA-based accelerators.

	PolyGraph [10]	Fifer [36]	HyCUBE[*] [52]	RipTide [15]	FLIP (this work)
Goal	High Perf.	High Perf.	Low Pwr.	Ultra Low Pwr.	Low Pwr.
Target	Graph App.	Irreg. App.	Reg. Loop	General App.	Graph App. & Reg. Loop
On-Chip Memory	512MB	4.5MB	4KB	256KB	32KB
Frequency	1GHz	2GHz	488MHz	50MHz	100MHz
Technology (nm)	28	22	40	sub-28	22
Power (mW)	2292	N/A	140	0.5[+]	26
Area (mm^2)	73	21	3	0.3[+]	0.4

[*]Silicon implementation. [+]PE array only, w/o on-chip memory.

4.4 Support for Data Swapping

FLIP compiler is extended to support data swap by replicating the PE array into $\lceil \frac{|V|}{P.capacity} \rceil$ copies. As data swapping occurs in a basic unit of a 2×2 PE cluster, only edges whose end vertices are mapped to the same PE cluster but different PE array copies are estimated to have higher transmission time.

5 EXPERIMENTAL EVALUATION

We evaluated FLIP against a micro-controller unit (MCU) and a classic CGRA [22], achieving up to 403× and 36× speedup respectively. We also compare FLIP with PolyGraph [10], a state-of-the-art datacenter-scale dedicated graph processor based on CGRA, and demonstrate better performance per area. Table 2 compares FLIP with recent CGRA-based accelerators using reported numbers from the papers. PolyGraph and Fifer [36] are state-of-the-art datacenter-scale CGRA-based accelerators. PolyGraph is dedicated to graph processing while Fifer supports irregular applications. However, they exhibit significantly larger power and area compared to FLIP. Compared to the edge accelerators

Table 3. Workloads for Evaluation.

Workload	Description	Time Complexity
BFS	BFS level	$O(|E| + |V|)$
SSSP	shortest path	$O(|E| + |V|log|V|), O(|V|^2)$
WCC	connected component	$O(|E| + |V|)$

HyCUBE and RipTide, Flip has similar area/power but is more versatile, supporting graph processing in addition to regular kernels. Note that RipTide area and power are for the PE array only and exclude 256KB on-chip memory, whereas other accelerators include on-chip memory in the estimation.

5.1 Evaluation Methodology

Baseline Architectures. For detailed quantitative comparison using the same workload, we evaluated Flip against two edge architectures: <u>MCU</u>: a mature ARM Cortex-M4F core running at 64MHz, and <u>CGRA</u>: a classic static-scheduled CGRA similar to [52]. Both classic CGRA and Flip have 8×8 PE arrays running at 100MHz and 16KB on-chip SPM. Note that Flip has an additional distributed on-chip storage (DRF and Inter/Intra-Tables).

Implementation. Both classic CGRA and Flip were implemented in System Verilog RTL [1] and synthesized on 22nm process using *the Synopsys* toolchain to evaluate power and area. We implemented Flip compiler for graph data mapping. For the classic CGRA, we use the Morpher compiler [57] to perform DFG mapping. We built an in-house cycle-accurate simulator for the whole system to faithfully execute the applications on Flip architecture for faster evaluation of performance. Functional validation is performed through the simulator.

Workload. We chose the widely-used graph algorithms: Breadth-First Search (BFS), Single-Source Shortest Paths (SSSP), and Weakly Connected Components (WCC) for evaluation, as detailed in Table 3. For SSSP, the solution with optimal time complexity $O(|E| + |V|log|V|)$ needs to use an advanced data structure, the priority queue built on a binary search tree. Classic CGRAs designed for regular computation-intensive kernels cannot support such dynamic and complex data structures. Therefore, we implemented the algorithm with $O(|V|^2)$ time complexity for SSSP on classic CGRA, while for MCU, the optimal one is employed. For the classic CGRA, iterating over a single vertex requires 34 operations for BFS and 38 for WCC. In SSSP, two kernels with 10 and 31 operations will be mapped for vertex searching and updating. In the case of Flip, the instruction count for processing a single vertex is 4 for WCC, 5 for BFS, and 5 for SSSP when the vertex's properties are updated. If there is no update, only 2 instructions for WCC, 4 instructions for BFS, 4 instructions for SSSP are executed. In other words, the computation cost is minimized in Flip as we do not need to generate memory addresses for load/store instructions.

Datasets. Table 4 shows the graphs we used to evaluate Flip. The original road networks of California and San Francisco are retrieved from the Standard Large Network Dataset Collection (SNAP) [25]. As edge devices rarely use the entire road network of a whole state (city), we constructed our datasets with small subgraphs obtained by BFS on the original road networks with random seeds. Additionally, a data set of synthetic graphs with short diameters was built by randomly connecting two vertices. To investigate the performance of Flip for different kinds of graphs, we constructed five types of graphs: *small road networks* (SRN), *large road network*s (LRN), *tree*s (Tree), *synthetic graphs* (Syn) and extra large road networks (Ext. LRN). The first four graph types—SRN, LRN, Tree, and Syn—are designed to fit entirely on-chip for performance evaluation purposes. The last data set is only used for scalability evaluation, where runtime data swapping from off-chip memory is required.

Table 4. Graph characteristics for varying CGRA sizes.

Group	Type	Diameter	# Graphs	|V|	|E|
Tree	Directed	High	100	256	255
SRN	Undirected	High	100	[64,107]	[146,278]
LRN	Undirected	High	100	256	[584,898]
Syn.	Directed	Low	100	256	768
Ext. LRN	Undirected	High	10	16k	[44k,50k]

Table 5. Performance-power-area comparison.

	MTEPS	Power (mW)	Area (mm^2)	Power Efficiency MTEPS/mW	Area Efficiency MTEPS/mm^2	Technology nm
MCU (LRN)	1.1	0.78‡	0.03‡	1.41	39	22
CGRA (LRN)	7.1	17	0.32	0.43	22	22
Flip (LRN)	158	26	0.37	**6.12**	**424**	22
PolyGraph (from [10])	13,845	2292	72.56	6.04	191	28
Fifer [36] & RipTide [15]	MTEPS is not available from papers					

‡On-chip memory excluded.

Performance Evaluation Methodology. To evaluate the performance of Flip, we run all the applications against the graph datasets in our simulator. We report the number of cycles when the application terminates for each run. For all graphs except Tree, we select 100 random vertices as the source vertex in each run and report the average. For Tree, applications only start with the root vertex.

5.2 Experimental Results

5.2.1 **Performance.** Fig. 10(a) shows the performance normalized to MCU for three different graph applications on four data sets. Note the logarithmic scale for the Y-axis. Flip consistently outperforms the baseline architectures across all workloads and graph types. It achieves a 25×-393× speedup compared to MCU. With optimal algorithm implementation in the baseline, Flip achieves an 11×-36× speedup compared to classic CGRA on BFS and WCC. Classic CGRA cannot expose sufficient parallelism within each loop iteration, given the small code snippet for processing each vertex and complex dependencies among vertices. In contrast, Flip improves the performance by exploiting data parallelism at runtime and enabling multiple active vertices to execute in parallel.

Classic CGRA outperforms MCU in most cases because MCU is a 5-stage single-issue in-order core, while these CGRA can compute and access memory in parallel with multiple PEs. MCU performs better than CGRA on SSSP due to the non-optimal algorithm implemented on classic CGRA.

5.2.2 **Power, Area and Energy.** As shown in Table 5, with the same size of PE array, Flip has only 19% more area but 53% more power than classic CGRA due to extra logic on the NoC and buffer queues at the ports to support dynamic routing. While Flip does have a higher power consumption, its significantly enhanced performance results in requiring only 5%-82% and 3%-15% of the energy compared to MCU and classic CGRA, respectively. Note that this comparison is biased towards MCU, as MCU energy is for the core only and does not include on-chip memory, whereas Flip energy includes a total of 32KB on-chip memory (16KB SPM and 16KB distributed memory inside PE array). Table 6 shows the power and area breakdown of Flip. Memory components consume 92.76% of the power and 88.19% of the area.
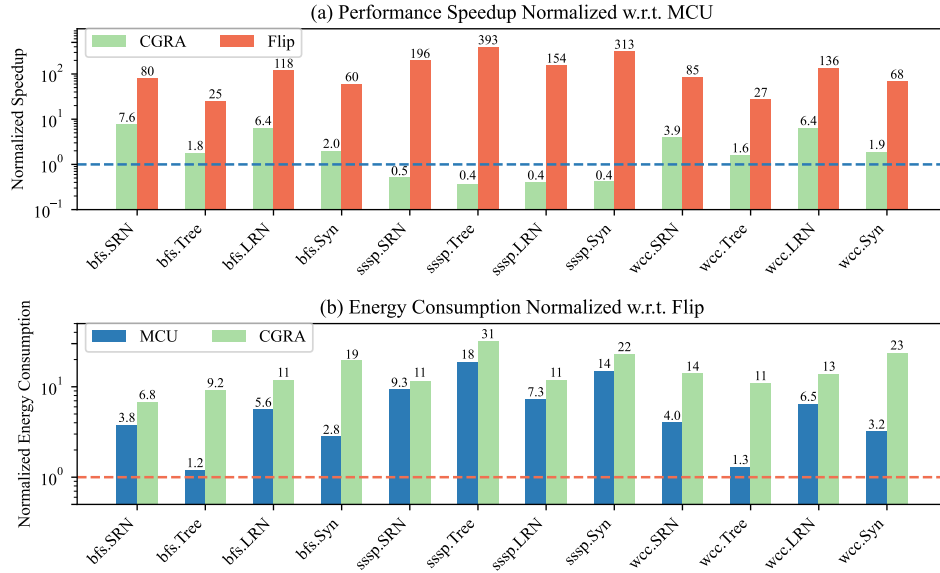
Fig. 10. Performance, energy comparison in *log* scale. Energy consumption for MCU is for the core only and excludes on-chip memory.

Moreover, Flip keeps a 32-entries instruction memory, taking 20% the power and area, to accommodate most kernels in operation-centric mode.

5.2.3 **Efficiency.** Table 5 also shows power and area efficiency of Flip compared with classic CGRA and a state-of-the-art CGRA-based large-scale graph accelerator, PolyGraph [10]. We use the absolute values mentioned in the original manuscript without scaling to a common setup. Because of the inconsistency in setup details, it is not feasible to scale to a common setup condition. Hence, we use this table to roughly place Flip among the current state of the art. The standard graph processing metric is used: Million Traversed Edges Per Second (MTEPS). The performance is evaluated on the WCC workload for all architectures. We report the average performance of PolyGraph running WCC on two unordered road networks, rdUSE and rdUSW, retrieved directly from [10]. Compared to PolyGraph, Flip achieves similar power efficiency and 2.2× area efficiency. This is because Flip compiler captures better data locality, reducing communication costs. Designed for edge-scale problems, we judiciously avoid complex components (e.g., scheduler and task coalescer) in Flip. PolyGraph uses complex controllers for data-center scale problems that are not feasible under stringent power/area constraints at the edge. Remarkably, Flip operates with just 1.1% and 0.5% of the area and power consumed by PolyGraph, underscoring its efficiency for edge-scale applications.

5.2.4 **Parallelism.** Fig. 11 shows the comparison of parallelism among classic CGRA and Flip. In this context, parallelism refers to the number of vertices that are being actively processed simultaneously. Operation-centric CGRAs were limited to processing fixed-size, tiny batches of vertices due to uncertain data dependencies, but they can exploit instruction parallelism within the computation for a vertex. As shown in the red area of Figure 11, the actual average parallelism is only 1 to 1.3 for different unroll degrees (1-4). In contrast, our approach accommodates variable-sized data batches, potentially handling considerably larger volumes of data.

Table 6. Power and Area Breakdown for Flip.

| Type	Component	Power (mW)	Area (mm^2)
Interconnect	Switch Allocator	0.08 (0.31%)	0.006 (1.60%)
Compute Unit	ALU	0.01 (0.04%)	0.004 (0.97%)
Memory	Inter-Table	5.91(22.90%)	0.073 (19.56%)
	Intra-Table	5.39 (20.89%)	0.065 (17.34%)
	ALUout Buffer	0.07 (0.27%)	0.021 (5.60%)
	ALUin Buffer	1.05 (4.07%)	0.011 (2.89%)
	Memory Buffer	0.75 (2.90%)	0.008 (2.90%)
	Input Buffer	4.02 (15.57%)	0.055 (14.74%)
	DRF	1.75 (6.77%)	0.021 (5.72%)
	Instruction Memory	4.89 (18.96%)	0.074 (19.74%)
	Slice ID Register	0.11 (0.42%)	0.001 (0.36%)
Additional Logic		1.78 (6.89%)	0.034 (9.24%)
Total		25.79	0.373
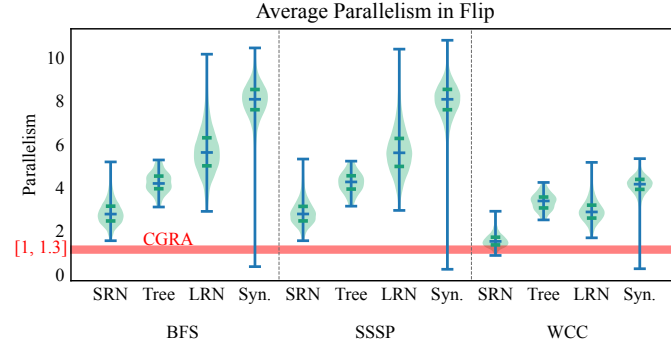
Fig. 11. Average parallelism for all runs on operation-centric CGRA and Flip.

As shown in Fig. 11, Flip achieves an average parallelism of 5.0-7.6 in most runs (25% quantile) when storage is fully utilized (LRN and Syn). Note that graph processing has naturally low parallelism at the beginning and end of execution. With a starting vertex in the center of the graph, the average parallelism can reach 10.4. However, the parallelism is fixed at a low level for classic CGRA. WCC performs worse than the other two applications on Flip due to the relatively minor computation demand, making the performance bounded by communication. For synthetic graphs, as not every vertex can reach all other vertices, the average parallelism in some rare runs could be low.

5.2.5 **Scalability.** Flip is capable of bringing in data at runtime from a 256KB off-chip main memory for larger problem sizes. Evaluated on Ext. LRN, the performance drops due to the overhead of data swapping at runtime but still greatly outperforms classic CGRA (5.7× throughput) and MCU (49.1× throughput). With a homogeneous PE array, Flip can be easily scaled up. However, as shown in Fig. 12, by linearly increasing PE array size and memory size, Flip shows some performance drop. This is because, for road networks, a larger graph denotes a longer diameter, requiring more hops to reach all vertices in the graph. As a result, even though the dataset and PE array are scaled proportionally, the total execution time inevitably increases, leading to some performance degradation.
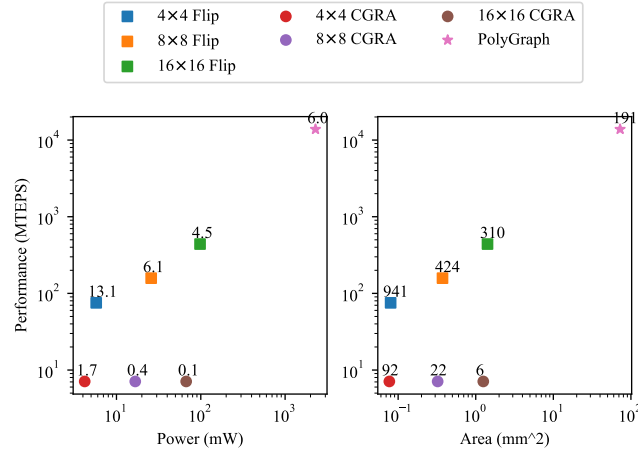
Fig. 12. Performance-power/-area comparison. Performance is evaluated for WCC with road networks that fully utilize the on-chip distributed memory. Memory size inside one PE remains constant during scaling. The number beside the node denotes the power/area efficiency.

5.2.6 **Compilation Time and Quality.** The compiler for classic CGRA generates mapping along the spatial and temporal dimensions and hence has a long compilation time. In contrast, the Flip compiler only needs to map the graph vertices on the PE array without any consideration of the temporal dimension. Due to the limited exploration space, the Flip compiler generates a valid mapping in less than 1% to 10% compilation time of conventional CGRAs, as shown in Fig. 13. The Y-axis in this graph has a logarithmic scale. However, compared with graph partition methods applied on data-center scale problems [2, 10, 62] where a maximum $O(|V| + |E|)$ complexity is allowed, Flip compiler provides a finer placement considering the routing distance as well as parallelism at the cost of time complexity. In the following, we dissect the time complexities associated with the two phases of the Flip compiler. The first phase is to use beam search with routing length as the cost to find an initial placement. For one node, the routing length is calculated by Manhattan distance for all connected edges with $O(\frac{|E|}{|V|})$ complexity, and the beam search has time complexity $O(k|V|)$ where k is set to 10 in Flip compiler. So, there is a total $O(|E|)$ complexity in the first phase. The second phase is to optimize the placement locally until stabilization. Consequently, the time complexity for a single iteration in this phase stands at $O(\frac{|V|+C|E|}{|P|})$, where $|P|$ is the number of PEs, C is the number of vertices a PE can accommodate. Table 7 shows the breakdown of the time complexity.

To evaluate the quality of mapping, we report statistics that indirectly capture the success of the compiler in minimizing routing length and contentions for resources in Table 8. The Flip compiler generates mappings with an average routing length of less than 1 for road networks. Recall that we might map neighboring vertices to the same PE (i.e., routing length 0) if it does not hurt parallelism.

The table also reports the contention for ALU when multiple active vertices are mapped to the same PE (ALUin Buffer Depth) and contention for routing resources (Pkt. Wait Time). The contention for ALU is negligible, while the waiting time for NoC resources is less than 10 cycles, even for large networks, demonstrating the efficacy of the compiler.

Table 7. Time Complexity Breakdown for FLIP Compiler.

Process	Time Complexity
Initial Mapping	$O(k\|V\|)$
Local Optimization (One Iteration)	$O(\frac{\|V\|+C\|E\|}{\|P\|})$
get neighboring PEs of a random PE	$O(\frac{\|V\|}{\|P\|C})$
get collision set	$O(C)$
get candidate vertices pairs	$O(C^2)$
time estimation for all edges of vertices pair	$O(\frac{\|E\|}{\|V\|})$

k: beam width in beam search. C: capacity of one PE. $\|P\|$: number of PEs.

Table 8. SSSP for different graph sizes on FLIP

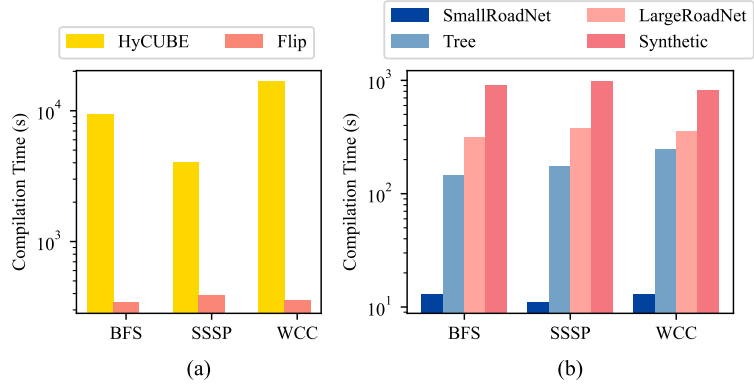| Graph Group	SRN	LRN	Tree	Syn.
Avg. Routing Length	0.63	0.76	0.55	2.46
Pkt. Wait Time (cycle)	7.8	9.6	5.1	7.9
ALUin Buffer Depth	0.04	0.08	0.03	0.14

Fig. 13. (a) Compilation time for three CGRAs. The average compilation time for all graphs is used for FLIP. (b) Compilation time of FLIP with different graph sizes.

6 RELATED WORK

We discuss related work from three perspectives: conventional operation-centric CGRAs, dedicated graph accelerators, and processing-in-memory acceleration. Three highly-correlated works are compared in detail. Note that the performance of a graph accelerator also depends on the parallelism available in the input graph data. The small-scale real-world road networks used in our experiment have much less parallelism compared to the large-scale synthetic graphs commonly benchmarked in other high-performance graph accelerators. Therefore, the power efficiency (MTEPS/Watt) should not be directly compared.

6.1 Conventional Operation-centric CGRAs

Conventional CGRAs exploit the Instruction Level Parallelism (ILP) to accelerate loop kernels. Many CGRAs also optimize performance through network-on-chip [52], concurrency [37], specialized memory access component [56], and predication for branch control [19, 23, 61], and compiler design [27, 29, 55, 58]. For irregular applications, efforts have been made on time-multiplexing [36] and heterogenous array [54]. More recent works [4, 31, 47, 49, 53] optimize the design of CGRA architecture for a specific set of applications through automatic design space exploration. OpenCGRA [46, 48] also provides an open-source full-stack unified framework to allow rapid design space exploration of CGRA. Despite all the efforts, all works still adhere to the conventional operation-centric approach. However, these methods still follow routine data-flow mapping with data streaming in exploiting ILP as well as regular independent data-level parallelism, which is not suitable for graph applications, while Flip directly maps the data and exploits the irregular data-level parallelism in the presence of rich dependencies. Furthermore, conventional CGRAs need to calculate data addresses for each access after loading graph data into on-chip memory. Although some CGRAs [10, 15, 37, 54, 56] introduce address generation units, the operation-centric design still leads to poor performance for graph applications with irregular memory access.

CGRAs with Dynamic Routing. Several CGRAs [35, 45, 51] are equipped with dynamic routing to support irregular and unpredictable communication. However, dynamic routing is only available at a top level among clusters of processing elements (at least 128 PEs in one cluster) [35, 45] or RISC-like processors (32KB memory in one tile) [51]. This is the same level as the whole Flip chip. Inside a PE cluster, [35, 45] still use static routing scheduled by the compiler. Therefore, when scaled down to the same range as Flip, these architectures can hardly benefit from dynamic routing.

6.2 Dedicated Graph Accelerators

A typical design for ASIC-based graph processors [3, 18, 38, 64] is to build hardware modules separately for the three phases in the *Gather-Apply-Scatter* model. A recent work by Luke et al. [14] proposes simulating the pathfinder on a mesh network with signal passing through NoC, but it is limited to mesh-like road networks. FPGA-based graph processors [21, 24, 65] achieve high throughput by exploiting the Multiple Instruction Single Data execution pattern and large accumulated bandwidth of on-chip memory. In the following, we differentiate Flip from two closely related works: PolyGraph [10] and GraphPulse [41].

Comparison between PolyGraph and Flip. PolyGraph [10, 11] is a state-of-the-art CGRA-based graph processor that allows modular integration of specialization features (e.g., synchronization frequency, scheduling strategy) for each graph processing variant (e.g., graph diameter, graph density, graph algorithm). It comprises a 4×4 core array connected by a mesh NoC. Each core has a control core for configuration and scheduling, a 5×4 CGRA array for computation, and a task management system for task caching, coalescing, and dispatching. It also features a comprehensive data access system for accurately retrieving data from local or other PEs without conflicts and offloading address computation to CGRA. PolyGraph can also be easily extended for general-purpose applications as it embeds classic operation-centric CGRA in each core. However, PolyGraph's design cannot be directly deployed at the edge due to its complex memory access and task scheduling systems. Remarkably, one single core of PolyGraph is 5.5× the power and 11.9× area of Flip. Additionally, PolyGraph uses CGRA in a classic way, while our approach innovatively alters the architecture to enable dual-mode execution. Finally, the execution inside each core in PolyGraph is managed by the task scheduler and control core. In other words, it is program-controlled. In contrast, Flip uses the data itself to trigger the subsequent execution in each PE, which is data-driven.

Comparison between GraphPulse and FLIP. GraphPulse [41] is an ASIC-based graph processor for efficient asynchronous graph processing. It bypasses the overhead of synchronization and frontier tracking in the Bulk Synchronous Parallel (BSP) processing model by employing an event-driven asynchronous processing model. In GraphPulse, each event encompasses the destination vertex ID and the changed vertex attribute, similar to FLIP. However, FLIP allows data to be sent directly to the destination PE rather than to a centralized event buffer and dispatched by a centralized scheduler. Although a centralized event coalescer helps reduce events heading to the same destination and subsequently decreases memory access, the coalescer is often area- and power-intensive, requiring a minimum size of the number of vertices in a graph multiplied by vertex attribute size for $O(1)$ access time. This makes GraphPulse unsuitable for edge devices with stringent power and area constraints.

6.3 Processing in Memory

Processing-in-Memory (PIM) is an effective technique that mitigates data movement costs by integrating processing units within memory. Graph processors enabled by PIM primarily utilize two technologies: ReRAM [59] and Hybrid Memory Cube (HMC) [39], which offer energy-efficient high-bandwidth memory access compared to conventional DRAM. GRAPHR [43], the first ReRAM-based graph processing accelerator, proposes to process the graphs in the SpMV format with crossbars. Tesseract [2] pioneered applying HMC technology for graph processing. Solutions such as GraphP [62], GraphQ, [66], and GraphH [12] optimize inter-cube communication through graph partition, reordering processing order, and efficient NoC.

Comparison between Tesseract and FLIP. Tesseract [2] constitutes a 4×4 HMC array with 4×8 vaults in each HMC. Each vault contains an in-order core for computation and task management, a data prefetcher, a data buffer, and a task queue, cumulatively facilitating up to 512 cores and 128GB of storage. Targeting large problem sizes, Tesseract utilizes a low-time-complexity simple index-based graph partitioning method at the expense of ignoring the inherent structure of the graph. In contrast, the FLIP compiler, tailored for smaller problem sizes coupled with limited memory in each PE, incorporates the graph structure into its partitioning and mapping algorithms. Moreover, FLIP compiler addresses the sequentialization problem, which is typically bypassed in PIM-enabled graph processors, where each vault can store a significant portion of the active frontier, minimizing the likelihood of staleness.

7 CONCLUSIONS

CGRAs are prominent accelerators but perform poorly on applications with rich control and data dependencies, especially graph processing, due to static scheduling. FLIP is a novel CGRA architecture that is good at graph processing at the edge while still supporting compute-intensive regular kernels. It adopts a novel data-centric model where the graph data (vertices) are mapped onto the PEs and the data dependencies (edges) are routed. FLIP achieves up to 393× speedup compared to micro-controller and 36× speedup compared to conventional CGRA accelerator on graph applications. Compared to large-scale graph processor[10], FLIP has similar energy efficiency and 2.2x better area efficiency at about 1% of the power/area cost.

REFERENCES

[1] 2018. IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)* (2018), 1–1315. https://doi.org/10.1109/IEEESTD.2018.8299595

[2] 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 42nd Annual International Symposium on Computer Architecture*. 105–117.

[3] Andrey Ayupov, Serif Yesil, Muhammet Mustafa Ozdal, Taemin Kim, Steven Burns, and Ozcan Ozturk. 2017. A template-based design methodology for graph-parallel hardware accelerators. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 2 (2017), 420–430.

[4] Thilini Kaushalya Bandara, Dhananjaya Wijerathne, Tulika Mitra, and Li-Shiuan Peh. 2022. REVAMP: a systematic framework for heterogeneous CGRA realization. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 918–932.

[5] Maciej Besta, Raghavendra Kanakagiri, Grzegorz Kwasniewski, Rachata Ausavarungnirun, Jakub Beránek, Konstantinos Kanellopoulos, Kacper Janda, Zur Vonarburg-Shmaria, Lukas Gianinazzi, Ioana Stefan, Juan Gómez Luna, Jakub Golinowski, Marcin Copik, Lukas Kapp-Schwoerer, Salvatore Di Girolamo, Nils Blach, Marek Konieczny, Onur Mutlu, and Torsten Hoefler. 2021. SISA: Set-Centric Instruction Set Architecture for Graph Mining on Processing-in-Memory Systems. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 282–297. https://doi.org/10.1145/3466752.3480133

[6] Maciej Besta, Dimitri Stanojevic, Johannes De Fine Licht, Tal Ben-Nun, and Torsten Hoefler. 2019. Graph processing on fpgas: Taxonomy, survey, challenges. *arXiv preprint arXiv:1903.06697* (2019).

[7] Mauro Bisson, Massimo Bernaschi, and Enrico Mastrostefano. 2016. Parallel Distributed Breadth First Search on the Kepler Architecture. *IEEE Transactions on Parallel and Distributed Systems* 27, 7 (2016), 2091–2102. https://doi.org/10.1109/TPDS.2015.2475270

[8] Aydin Buluc, Scott Beamer, Kamesh Madduri, Krste Asanovic, and David Patterson. 2017. Distributed-Memory Breadth-First Search on Massive Graphs. arXiv:1705.04590 [cs.DC]

[9] James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Gregory R Ganger, Garth Gibson, Kimberly Keeton, and Eric Xing. 2013. Solving the straggler problem with bounded staleness. In *14th Workshop on Hot Topics in Operating Systems (HotOS XIV)*.

[10] Vidushi Dadu, Sihao Liu, and Tony Nowatzki. 2021. Polygraph: Exposing the value of flexibility for graph processing accelerators. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 595–608.

[11] Vidushi Dadu, Sihao Liu, and Tony Nowatzki. 2022. Systematically understanding graph accelerator dimensions and the value of hardware flexibility. *IEEE Micro* 42, 4 (2022), 87–96.

[12] Guohao Dai, Tianhao Huang, Yuze Chi, Jishen Zhao, Guangyu Sun, Yongpan Liu, Yu Wang, Yuan Xie, and Huazhong Yang. 2019. GraphH: A Processing-in-Memory Architecture for Large-Scale Graph Processing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 4 (2019), 640–653. https://doi.org/10.1109/TCAD.2018.2821565

[13] William James Dally and Brian Patrick Towles. 2004. *Principles and practices of interconnection networks*. Elsevier.

[14] Luke R Everson, Sachin S Sapatnekar, and Chris H Kim. 2021. A Time-Based Intra-Memory Computing Graph Processor Featuring A* Wavefront Expansion and 2-D Gradient Control. *IEEE Journal of Solid-State Circuits* 56, 7 (2021), 2281–2290.

[15] Graham Gobieski, Souradip Ghosh, Marijn Heule, Todd Mowry, Tony Nowatzki, Nathan Beckmann, and Brandon Lucia. 2022. A programmable, energy-minimal dataflow compiler and architecture. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 546–564.

[16] Chuang-Yi Gui, Long Zheng, Bingsheng He, Cheng Liu, Xin-Yu Chen, Xiao-Fei Liao, and Hai Jin. 2019. A survey on graph processing accelerators: Challenges and opportunities. *Journal of Computer Science and Technology* 34, 2 (2019), 339–371.

[17] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the fourth annual IEEE international workshop on workload characterization. WWC-4 (Cat. No. 01EX538)*. IEEE, 3–14.

[18] 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 *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–13.

[19] Mahdi Hamzeh, Aviral Shrivastava, and Sarma Vrudhula. 2014. Branch-aware loop mapping on CGRAs. In *Proceedings of the 51st Annual Design Automation Conference*. 1–6.

[20] Natalie Enright Jerger, Tushar Krishna, and Li-Shiuan Peh. 2017. On-chip networks. *Synthesis Lectures on Computer Architecture* 12, 3 (2017), 1–210.

[21] Nachiket Kapre. 2015. Custom FPGA-based soft-processors for sparse graph acceleration. In *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 9–16.

[22] Manupa Karunaratne, Aditi Kulkarni Mohite, Tulika Mitra, and Li-Shiuan Peh. 2017. Hycube: A cgra with reconfigurable single-cycle multi-hop interconnect. In *Proceedings of the 54th Annual Design Automation Conference 2017*. 1–6.

[23] Manupa Karunaratne, Dhananjaya Wijerathne, Tulika Mitra, and Li-Shiuan Peh. 2019. 4D-CGRA: Introducing branch dimension to spatio-temporal application mapping on CGRAs. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–8.

[24] Guoqing Lei, Yong Dou, Rongchun Li, and Fei Xia. 2015. An FPGA implementation for solving the large single-source-shortest-path problem. *IEEE Transactions on Circuits and Systems II: Express Briefs* 63, 5 (2015), 473–477.

[25] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[26] Feifei Li, Dihan Cheng, Marios Hadjieleftheriou, George Kollios, and Shang-Hua Teng. 2005. On trip planning queries in spatial databases. In *International symposium on spatial and temporal databases*. Springer, 273–290.

[27] Zhaoying Li, Dhananjaya Wijerathne, Xianzhang Chen, Anuj Pathania, and Tulika Mitra. 2021. Chordmap: Automated mapping of streaming applications onto cgra. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 2 (2021), 306–319.

[28] Zhaoying Li, D Wijerathne, and Tulika Mitra. 2022. Coarse Grained Reconfigurable Array CGRA. *Book Chapter in Springer Handbook of Computer Architecture* (2022).

[29] Zhaoying Li, Dan Wu, Dhananjaya Wijerathne, and Tulika Mitra. 2022. Lisa: Graph neural network based portable mapping on spatial accelerators. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 444–459.

[30] Leibo Liu, Jianfeng Zhu, Zhaoshi Li, Yanan Lu, Yangdong Deng, Jie Han, Shouyi Yin, and Shaojun Wei. 2019. A survey of coarse-grained reconfigurable architecture and design: Taxonomy, challenges, and applications. *ACM Computing Surveys (CSUR)* 52, 6 (2019), 1–39.

[31] Sihao Liu, Jian Weng, Dylan Kupsh, Atefeh Sohrabizadeh, Zhengrong Wang, Licheng Guo, Jiuyang Liu, Maxim Zhulin, Rishabh Mani, Lucheng Zhang, Jason Cong, and Tony Nowatzki. 2022. OverGen: Improving FPGA usability through domain-specific overlay generation. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 35–56.

[32] Yucheng Low, Joseph E Gonzalez, Aapo Kyrola, Danny Bickson, Carlos E Guestrin, and Joseph Hellerstein. 2014. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041* (2014).

[33] Kevin JM Martin. 2022. Twenty Years of Automated Methods for Mapping Applications on CGRA. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 679–686.

[34] Robert Ryan McCune, Tim Weninger, and Greg Madey. 2015. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR)* 48, 2 (2015), 1–39.

[35] Mahim Mishra, Timothy J Callahan, Tiberiu Chelcea, Girish Venkataramani, Seth C Goldstein, and Mihai Budiu. 2006. Tartan: evaluating spatial computation for whole program execution. *ACM SIGARCH Computer Architecture News* 34, 5 (2006), 163–174.

[36] Quan M Nguyen and Daniel Sanchez. 2021. Fifer: Practical acceleration of irregular applications on reconfigurable architectures. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 1064–1077.

[37] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. 2017. Stream-dataflow acceleration. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 416–429.

[38] Muhammet Mustafa Ozdal, Serif Yesil, Taemin Kim, Andrey Ayupov, John Greth, Steven Burns, and Ozcan Ozturk. 2016. Energy efficient architecture for graph analytics accelerators. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 166–177.

[39] J Thomas Pawlowski. 2011. Hybrid memory cube (HMC). In *2011 IEEE Hot chips 23 symposium (HCS)*. IEEE, 1–24.

[40] Artur Podobas, Kentaro Sano, and Satoshi Matsuoka. 2020. A survey on coarse-grained reconfigurable architectures from a performance perspective. *IEEE Access* 8 (2020), 146719–146743.

[41] Shafiur Rahman, Nael Abu-Ghazaleh, and Rajiv Gupta. 2020. Graphpulse: An event-driven hardware accelerator for asynchronous graph processing. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 908–921.

[42] Steven M Rubin and Raj Reddy. 1977. The locus model of search and its use in image interpretation. *Cambridge, Massachusetts* (1977), 590–595.

[43] Linghao Song, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. 2018. GraphR: Accelerating graph processing using ReRAM. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 531–543.

[44] Jacob R Stevens, Dipankar Das, Sasikanth Avancha, Bharat Kaul, and Anand Raghunathan. 2021. GNNerator: A hardware/software framework for accelerating graph neural networks. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 955–960.

[45] Steven Swanson, Ken Michelson, Andrew Schwerin, and Mark Oskin. 2003. WaveScalar. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36*. IEEE, 291–302.

[46] Cheng Tan, Nicolas Bohm Agostini, Jeff Zhang, Marco Minutoli, Vito Giovanni Castellana, Chenhao Xie, Tong Geng, Ang Li, Kevin Barker, and Antonino Tumeo. 2021. Opencgra: Democratizing coarse-grained reconfigurable arrays. In *2021 IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 149–155.

[47] Cheng Tan, Thierry Tambe, Jeff Zhang, Bo Fang, Tong Geng, Gu-Yeon Wei, David Brooks, Antonino Tumeo, Ganesh Gopalakrishnan, and Ang Li. 2022. ASAP: automatic synthesis of area-efficient and precision-aware CGRAs. In *Proceedings of the 36th ACM International Conference on Supercomputing*. 1–13.

[48] Cheng Tan, Chenhao Xie, Ang Li, Kevin J Barker, and Antonino Tumeo. 2020. OpenCGRA: An open-source unified framework for modeling, testing, and evaluating CGRAs. In *2020 IEEE 38th International Conference on Computer Design (ICCD)*. IEEE, 381–388.

[49] Cheng Tan, Chenhao Xie, Ang Li, Kevin J Barker, and Antonino Tumeo. 2021. Aurora: Automated refinement of coarse-grained reconfigurable accelerators. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1388–1393.

[50] Dani Voitsechov and Yoav Etsion. 2014. Single-graph multiple flows: Energy efficient design alternative for GPGPUs. *ACM SIGARCH computer architecture news* 42, 3 (2014), 205–216.

[51] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. 1997. Baring it all to software: Raw machines. *Computer* 30, 9 (1997), 86–93.

[52] Bo Wang, Manupa Karunarathne, Aditi Kulkarni, Tulika Mitra, and Li-Shiuan Peh. 2019. Hycube: A 0.9 v 26.4 mops/mw, 290 pj/op, power efficient accelerator for iot applications. In *2019 IEEE Asian Solid-State Circuits Conference (A-SSCC)*. IEEE, 133–136.

[53] Jian Weng, Sihao Liu, Vidushi Dadu, Zhengrong Wang, Preyas Shah, and Tony Nowatzki. 2020. Dsagen: Synthesizing programmable spatial accelerators. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 268–281.

[54] Jian Weng, Sihao Liu, Zhengrong Wang, Vidushi Dadu, and Tony Nowatzki. 2020. A hybrid systolic-dataflow architecture for inductive matrix algorithms. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 703–716.

[55] Dhananjaya Wijerathne, Zhaoying Li, Thilini Kaushalya Bandara, and Tulika Mitra. 2022. PANORAMA: divide-and-conquer approach for mapping complex loop kernels on CGRA. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*. 127–132.

[56] Dhananjaya Wijerathne, Zhaoying Li, Manupa Karunarathne, Anuj Pathania, and Tulika Mitra. 2019. Cascade: High throughput data streaming via decoupled access-execute cgra. *ACM Transactions on Embedded Computing Systems (TECS)* 18, 5s (2019), 1–26.

[57] Dhananjaya Wijerathne, Zhaoying Li, Manupa Karunaratne, Li-Shiuan Peh, and Tulika Mitra. 2022. Morpher: An Open-Source Integrated Compilation and Simulation Framework for CGRA. In *Fifth Workshop on Open-Source EDA Technology (WOSET)*.

[58] Dhananjaya Wijerathne, Zhaoying Li, Anuj Pathania, Tulika Mitra, and Lothar Thiele. 2021. Himap: Fast and scalable high-quality mapping on cgra via hierarchical abstraction. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 10 (2021), 3290–3303.

[59] H-S Philip Wong, Heng-Yuan Lee, Shimeng Yu, Yu-Sheng Chen, Yi Wu, Pang-Shiu Chen, Byoungil Lee, Frederick T Chen, and Ming-Jinn Tsai. 2012. Metal–oxide RRAM. *Proc. IEEE* 100, 6 (2012), 1951–1970.

[60] Pengcheng Yao, Long Zheng, Yu Huang, Qinggang Wang, Chuangyi Gui, Zhen Zeng, Xiaofei Liao, Hai Jin, and Jingling Xue. 2022. Scalagraph: A scalable accelerator for massively parallel graph processing. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 199–212.

[61] Baofen Yuan, Jianfeng Zhu, Xingchen Man, Zijiao Ma, Shouyi Yin, Shaojun Wei, and Leibo Liu. 2021. Dynamic-II pipeline: compiling loops with irregular branches on static-scheduling CGRA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 9 (2021), 2929–2942.

[62] 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 *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 544–557.

[63] Yu Zhang, Xiaofei Liao, Hai Jin, Ligang He, Bingsheng He, Haikun Liu, and Lin Gu. 2021. Depgraph: A dependency-driven accelerator for efficient iterative graph processing. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 371–384.

[64] Jinhong Zhou, Shaoli Liu, Qi Guo, Xuda Zhou, Tian Zhi, Daofu Liu, Chao Wang, Xuehai Zhou, Yunji Chen, and Tianshi Chen. 2017. Tunao: A high-performance and energy-efficient reconfigurable accelerator for graph processing. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 731–734.

[65] Shijie Zhou, Rajgopal Kannan, Hanqing Zeng, and Viktor K Prasanna. 2018. An FPGA framework for edge-centric graph processing. In *Proceedings of the 15th ACM International Conference on Computing Frontiers*. 69–77.

[66] 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 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 712–725. https://doi.org/10.1145/3352460.3358256