2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO) | 978-1-6654-6272-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/MICRO56248.2022.00083

# An architecture interface and offload model for low-overhead, near-data, distributed accelerators

Saambhavi Baskaran, Mahmut Taylan Kandemir, Jack Sampson
*Pennsylvania State University*
*sxv49@psu.edu, mtk2@psu.edu, jms1257@psu.edu*

***Abstract*—The performance and energy costs of coordinating and performing data movement have led to proposals adding compute units and/or specialized access units to the memory hierarchy. However, current on-chip offload models are restricted to fixed compute and access pattern types, which limits software-driven optimizations and the applicability of such an offload interface to heterogeneous accelerator resources. This paper presents a computation offload interface for multi-core systems augmented with distributed on-chip accelerators. With energy-efficiency as the primary goal, we define mechanisms to identify offload partitioning, create a low-overhead execution model to sequence these fine-grained operations, and evaluate a set of workloads to identify the complexity needed to achieve distributed near-data execution.**

**We demonstrate that our model and interface, combining features of dataflow in parallel with near-data processing engines, can be profitably applied to memory hierarchies augmented with either specialized compute substrates or lightweight near-memory cores. We differentiate the benefits stemming from each of elevating data access semantics, near-data computation, inter-accelerator coordination, and compute/access logic specialization. Experimental results indicate a geometric mean (energy efficiency improvement; speedup; data movement reduction) of (3.3; 1.59; 2.4)×, (2.46; 1.43; 3.5)× and (1.46; 1.65; 1.48)× compared to an out-of-order processor, monolithic accelerator with centralized accesses and monolithic accelerator with decentralized accesses, respectively. Evaluating both lightweight core and CGRA fabric implementations highlights model flexibility and quantifies the benefits of compute specialization for energy efficiency and speedup at 1.23× and 1.43×, respectively.**

***Keywords*-distributed accelerator; near-data offload; energy efficiency; heterogeneous architecture interface**

## I. Introduction

Between the end of Dennard Scaling [1] and the spiraling divergence in the energy costs of communication and computation, increasing integration merely by raising the number of general-purpose cores (GPP) and cache resources has limited appeal [2], [3]. An alternative, explored by several recent efforts, focuses on augmenting the cache and memory hierarchy with specialized computational units to overcome data movement limitations, and thereby achieve better energy efficiency [4], [5], [6], [7], [8], [9], [10]. The design space of future energy-efficient architectures is poised to become extremely diverse owing to the heterogeneity in computation and non-uniformity in data access costs brought on by these efforts, further amplifying the heterogeneity already being embraced by commercial designs [11], [12], [13]. Effectively managing this diversity will require adopting new abstractions and developing new optimizations appropriate to these platforms for lower time-to-market and better accelerator-level parallelism (ALP) [14].

The traditional primacy of active, centralized compute units within a hierarchy of mostly passive memory units creates an overhead in the movement of both control and data. Fixed-granularity, demand-based load and store interfaces exhibit control inefficiencies in scenarios where data accesses are highly structured and data movement inefficiencies when only a subset of the accessed data, or a collective property of said data, is desired by the computation (e.g., searching, filtering, reductions, etc). Furthermore, given the scale of modern memory systems, there is no single location within the memory hierarchy that can be plausibly *near-data* for all data. To achieve peak efficiency, near-data approaches must distribute compute resources proportionately to the physical scale of the memory system.

Prior works have addressed portions of these concerns by specializing streaming memory access patterns [8], [9] and offloading computations near data [4], [7], [15], [16]. A traditional, A-to-B offload model, where a GPP offloads a part of the computation to a *monolithic*, on-chip accelerator resource may still exhibit data movement overheads, as operands are often spread *throughout* the cache hierarchy. Systems where recurrent *data access functionality* is *decoupled and decentralized* from the host/accelerator [8], [17], [18], still require data to move to a central location for computation, causing a bandwidth bottleneck. Some near-data offload models [10], [19] provide support for fixed compute patterns and/or data access functionalities. Extending the processor ISAs with *binding* accelerator functionality restricts the applicability of the offload interface to other application and access characteristics. This also constrains the scope of software optimizations that can be applied in such an offload model. In other words, a desirable offload model for designs with increasingly diverse resources should be extensible to heterogeneous accelerator resources with arbitrary functionality. While the software optimizations can help localize computations to data in such offload models, distributed compute units require *specialized, yet flexible,* interface definitions to move control and data efficiently among various compute units. In this work, we show that generalizing the offload of *both* distributed near-data computations and common communication/access patterns among the distributed accelerators allows the exploitation of greater ALP in architectures while managing the increasingly divergent spatial access costs and asymmetric compute-

978-1-6654-6272-3/22/$31.00 ©2022 IEEE
DOI 10.1109/MICRO56248.2022.00083

memory costs effectively.

Our proposed offload model answers the following questions– **What to offload?** We exploit compiler mechanisms to extract key application semantics such as interrelationship between multiple data structures in the application, their respective data access patterns and associated instruction chains that are profitable to offload; **How to be near-data?** We use the above information to partition the application code, and identify placement of partitioned computations near-data in the cache hierarchy to achieve better energy efficiency; and **how to interface with and among offloads?** We utilize a generic offload model encompassing an architecture interface and compiler-generated offload configurations to enable distributed and concurrent compute units to co-exist as peers for operand communication.

The major contributions of this work include:

(1) We propose a **novel offload interface with minimal constraints for architecture models with *both* distributed computation and distributed access capabilities.** The proposed interface is designed in a generic manner for offloading *arbitrary functionalities* to heterogeneous accelerator resources, irrespective of their substrate implementation. Further, the interface offers flexible communication mechanisms for energy-efficient orchestration of both control and data.

(2) We demonstrate a **compiler-automated framework to exploit the proposed offload interface with an optimization goal of reducing data movements.** Given an application coded in an imperative programming language, our proposed compiler support automatically analyzes and partitions the code to map onto distributed accelerator resources employing the proposed interface.

(3) We **validate the offload interface for various architecture models**. Our experimental evaluation shows that the architecture models with distributed compute/access capabilities employing the proposed interface have, for the application benchmarks considered, a geometric mean energy efficiency of 3.3×, 2.46× and 1.46×, and data movement reduction of 2.4×, 3.5× and 1.48×, while also having a speedup of 1.59×, 1.43× and 1.65× compared to an out-of-order processor, a monolithic accelerator with centralized accesses on L3 bus and monolithic accelerators with decentralized accesses respectively.

(4) We further **differentiate the energy and performance benefits arising along two axes –** near-data computation specialization and data access/communication specialization.

## II. Background

We classify our design space along two major specialization trends that aim to reduce data and control movements: (1) ***monolithic* vs. *distributed* computation–** distributing accelerators (termed *compute nodes*) through the memory hierarchy, and (2) ***centralized* vs. *decentralized* accesses–**decoupling memory accesses from the program and distributing the hardware orchestrating the accesses (termed *access nodes*) through the memory hierarchy. The *physical locality* of compute nodes and their decentralization reduces the control and data movement overheads through the system hierarchy, whereas increasing the *specialization of data access* minimizes the control overhead required to move data to compute nodes. Figure 1a-d shows a code snippet and how it can be executed in the design space with the architecture models under consideration.

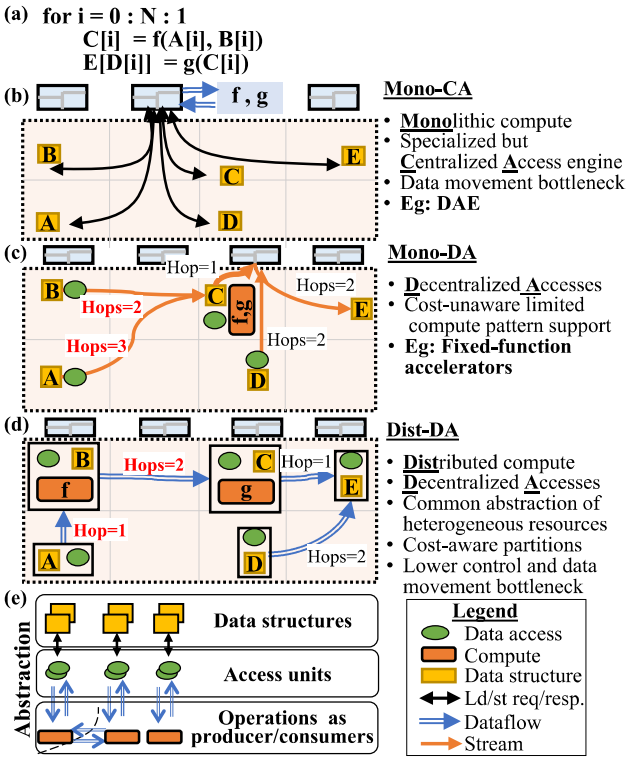



Figure 1. Design space for compute and memory access specialization approaches showing the dataflow for the example code snippet.

**Monolithic-Compute Centralized Accesses (Mono-CA):** General-purpose out-of-order (OoO) cores, on-chip accelerators [20] and decoupled access-execute architectures have *monolithic* execution semantics and/or *specialized* accesses (e.g., access pattern-specific prefetches). However, as shown in Figure 1b these require its operands to be *moved* to a *centralized* location, where f and g are executed. This causes data and control movement bottlenecks.

**Monolithic-Compute Decentralized Accesses (Mono-DA):** In a basic Mono-DA architecture model, there is only one place where an offloaded computation happens. However, there are distributed access points from where the data are forwarded. Recent works [8], [9], [21] offload the *functionality of accessing* data to the memory system. This facilitates the memory hierarchy to supply data related to

the configured access patterns (streams, e.g., for i=0:N; A[i] or indirect streams, e.g., for i=0:N;B[A[i]]) faster and in an energy-efficient fashion with reduced *control* overheads. Besides this, we also classify architectures as Mono-DA if a given offloaded computation is mapped in a monolithic fashion even if *different computations* can be mapped concurrently to various locations in the hierarchy. While these architectures offer more distributed computation options than a basic Mono-DA architecture, near-data architectures with fixed compute pattern support in/near-cache do not support breaking apart the compute pattern to reduce data movement (i.e., the particular computation is mapped *monolithic*ally), and hence lose opportunities to perform "sub-computation" [22] placement. Figure 1c shows an example of this architecture model, where the multi-input operands are forwarded to a single location to compute and write data in-place. *A and B* are forwarded to *C*, where the *functions-f and g* are concurrently executed. In the example, it might be energy-efficient to forward *A* to *B*, and forward only the result of *function-f* to *C*. In other words, *the offload interface constrains further placement optimizations* that are essential for reducing data movements. The complex indirect accesses such as *E[D[i]]* in the figure do not get offloaded to an accelerator, since the architecture paradigm does not associate the indirect memory references to application data structures and does not allow representing the interdependence of various data structures following arbitrary semantics.

**Distributed-Compute Decentralized Accesses (Dist-DA):** Architectures in this space should provide both distributed and flexible execution semantics along with decentralized accesses. In Figure 1d, multiple read dataflows *A and B* are associated together, where operation *f* happens. The result, an intermediate dataflow, is then forwarded to the location of *C*. Dist-DA requires only three hops, *collectively*, for the result of function-*f* to reach the location of *C*, compared to Mono-DA. Such a model first requires a *common abstraction for the offloads and for interfacing among the various concurrently-active nodes*, so that the compiler/programmer can use these resources efficiently. This further allows compiler partitioning optimizations to be applied *independently* of the interface definition. Secondly, Figure 1d also shows how the interrelationship among data structures (*A, B,* etc.) is used to create a natural flow of control and data among various computation nodes through a *low-overhead operand communication interface*. The dataflow from location-*B* updates *C* and gets forwarded to be consumed by a compute node at location-*E*. Similarly, the compute node at location-*D* can be configured as another producer for the node at location-*E*. The nodes at locations *A - E* can be *independently* mapped onto *any accelerator substrate* available near data.

Note that an effective offload model targeting this design space has the following requirements:

**R1 Flexible co-location of compute and data for reduced data movement**: The model should support distributed arbitrary compute functions (including sub-computations) to enable static/runtime optimizations, such as cost-aware partitioning to reduce data movements.

**R2 Low-overhead operand communication**: The model requires interface mechanisms for inter-accelerator communications to happen without incurring the high control overheads of host orchestrations.

**R3 Heterogeneous accelerator implementation**: The model and the interface should ***not*** dictate the type of execution substrate at the distributed locations. In other words, the model should permit heterogeneous accelerator resources to use the common interface for inter-accelerator or host-accelerator communication, with minimal micro-architectural requirements in accelerator/accessor design.

### A. Related Work

Our work relates to four primary areas of research as follows.

**Specialization/decentralization of data accesses:** Decoupled access-execute [18], [23], [24], [25] architectures decouple data accesses from other computations for performance speedup. StreamISA [9] and stream floating [8] adds the notion of "streams" to processor ISAs and the cache hierarchy, to enable caches to proactively generate and forward access requests without requiring host intervention for specific memory access patterns. These approaches help to improve performance speed-up, as long-latency accesses can be hidden and can also reduce the energy-inefficiencies of control overhead for fetching the data.

**Near-data architectures:** There have been numerous proposals for designing data-centric architectures [4], [5], [7], [15]. One common approach involves augmenting cache with logic circuits for in-place computation and is driven by technology advancements [5], [6], [16]. Reconfigurable arrays in the last-level cache executing coarse granularity offloads [4] have also been investigated. Our focus is to explore the offload model and architecture interface for better exploiting the benefits of such in/near-data technology improvements.

**Near-data offload models:** Near-stream computing (NSC) [10] extends the host ISA and coherence protocols to support sixteen offload functions near stream-based (strided/indirect/pointer-chasing) memory accesses. For complex operation offloads, it relies on the nearby core. Additional microarchitecture modifications in the host processor can provide benefits such as tight-coupling with the processor and fine-grained range management within a data structure (may-aliasing streams). However, the stream ISA abstraction binds the offloadable function and access patterns. There is also less flexibility to exercise different code partition strategies for data movement optimizations or to make use of on-chip heterogeneous accelerator resources. Livia [7] proposes *memory services*, a task-based

programming model and architecture that automatically schedules tasks based on data locality. However, it is limited to irregular workloads with only single kernels operating on a single cacheline. Table I differentiates our approach from these works. NSC considers augmenting the host processor ISA and micro-architecture capabilities for near-stream abstraction. On the other hand, our work proposes a *generic* interface for near-data accelerators to communicate with each other. The higher-level philosophies of these works are different, and the implications of these approaches are significant. We believe that the remote accelerator definitions and their micro-architectures should not be bound by the host ISA. The accelerators in the system should be able to communicate as first-class citizens without any host-side overhead. Whereas NSC extracts *streams* and reduces vertical data movement through the cache hierarchy, we associate code with data structures and allow software optimizations to reduce on-chip data movement including inter-tile movement. NSC augments the host with multiple stream-engine cores that monitor the address ranges of may-alias streams before offloading to remote accelerators. While we do not handle aliasing (false positive) streams, the cases where code regions do not know which data structures they are handling are rare for the workloads considered. Hence, in this work, we only rely on compiler optimizations and a simple accelerator scheduler to enable multi-access combining and distribution within a data structure (Section IV Figure 2). Further, the proposed interface does not preclude compiler passes from adding software runtime checks for may-alias pointers, and dynamically let the host decide to offload.

Recent work [26] has targeted finer-grained near-data offloads by decentralizing the accelerator offload mechanism, while still employing a monolithic view of memory, and like Livia [7], does not make use of the concurrency enabled by distributed computation substrates. Both these works start and finish an offloaded function to completion, and then pass on the control to another accelerator. On the other hand, PEI [15] and GraphPIM [27] extend the processor ISA to interface with other in-memory co-processors. In this work, we leave the distributed offload functionality to be defined by the software. We adhere to energy-efficient dataflow for recurring patterns of communication between the distributed accelerators using software-configured interface mechanisms. Our Dist-DA offload interface enables a variable degree of coupling between the distributed accelerator resources, which in turn facilitates the applicability of additional optimizations such as, data movement reduction and further computation specialization of the distributed application code.

**On-chip accelerator specialization and multi-accelerator coordination:** Charm [28], [29] proposes a multi-core architecture with distributed accelerator resources with DMA and scratchpad, and an ability to pipeline the output of one accelerator to another for exploiting pipeline parallelism. Stitch [30] proposes a many-core architecture with distributed *patches* of fixed computation patterns communicating over a static NoC. However, the offload interfaces in both these works are not generic, do not allow various degrees of offload granularity, and require a specific implementation substrate for accelerators. Pipette [31] and Fifer [32] architectures propose techniques to exploit fine-grained pipeline-parallelism in irregular applications for general-purpose processors and large-scale CGRAs (coarse-grained reconfigurable arrays). While the focus of these works is different from ours, the techniques to decouple compute partitions and the principles to augment producer-consumer partitions with control flow semantics are similar to our work. However, our interface mechanisms to support irregular control flow are more generic in its applicability since each accelerator can choose to define the response functions to control flow indications.

Table I
COMPARISON WITH RELATED WORKS.

| | **Livia [7]** | **NSC [10]** | **This work** |
|---|---|---|---|
| **Access:** #concurrent data structures | One | Many | Many |
| **Compute:** Offload function | Software-defined; One task at a time | ISA-defined 16 compute patterns | Software-defined accelerator functions |
| Offload substrate | Programmable unit-in-order core/FPGA | Scalar unit/remote SMT core | Programmable unit-in-order core/CGRA |
| Abstraction | Data structure/ Memory service | Stream & coupled compute pattern | Access, compute, data structures (memory objects) |
| **Communication** | N/A | Look-up & forward | Peer-like |
| Data movement-aware partitioning | N/A | Partial (fixed patterns limit optimizations) | Yes |
| **Target appls** | Irregular | Stream (strided/indirect /pointer chase) with no reuse | Both |
| **Additional impl. overhead** | High (new programming model) | High (ISA, coherence protocol) | Low (compiler intrinsics) |

## III. OUR APPROACH

Our goal is to define an offload model for a Dist-DA target architecture, shown in Figure 2a, while adhering to the requirements listed in Section II. We make the following design choices to build an offload interface that allows distributing code regions flexibly and at low overhead, for mapping on heterogeneous accelerator micro-architectures.

• **Configurable semantics of offload function**: We target partitioning at sub-computation granularity which is *necessitated* for reducing data movement by R1 and as has also been established by prior work [19], [22]. Therefore, the functionality of the distributed accelerators should not be bound by the host ISA or its micro-architecture. With this in mind, we define an abstraction (**Section IV-A**) for the compiler to extract such sub-computations along with application data structure semantics. Associating code regions to application data structures helps to reduce inter-data structure traffic. We then build an automated compiler approach for programmer-transparency, and to define communication-aware distributed offload functions.

• **Decoupled producer-consumer communication**: We observe that, among the considered workloads, loops performing recurrent functions dominate the dynamic code coverage between 70-99%. This results in the communication between

distributed offload functions to have a "recurrent" producer-consumer relationship. We exploit this behavior by adopting a dataflow communication model for energy-efficiency (**Section IV-B**). We configure the accelerator definitions with details of its producer/consumer to enable implicit low overhead communications between the offload functions as required by R2. This reduces the control overhead of recurrent communications compared to a two-way request-response mechanism. Note that this implicit dataflow communication mechanism can be further generalized to accelerator pipelines beyond inner-loop scope, but that our current compiler flow does not *automate* exploitation of these scopes.

• **Accelerator architecture-agnostic**: The rising heterogeneity of emerging systems requires the interfaces between different compute units to be as general as possible to ensure easier portability, maintenance, and future-proofing. To achieve this R3, our architecture interface does *not* impose a specific accelerator architecture beyond requiring a generic interface model for configuration-related and operand communication (**Section IV-C**). In other words, our offload model permits the use of heterogeneous accelerator resources by employing software intrinsics for inter-accelerator or host-accelerator communication. In addition, we use a buffer within the access units to decouple distributed offloads. The buffers are also used to capture reuse and to provide latency-insensitivity of memory accesses and operand communication.

Our main contribution is defining a *generic offload interface* that allows a better exploitation of advantageous features of the Dist-DA architectural model compared to prior works. Since the offload interface allows flexible offload function definitions, it permits software-driven fine-grained optimizations to be applicable for on-chip data movement reductions. Further, heterogeneous accelerator architectures can conform to the proposed offload interface without requiring intrusive design changes.

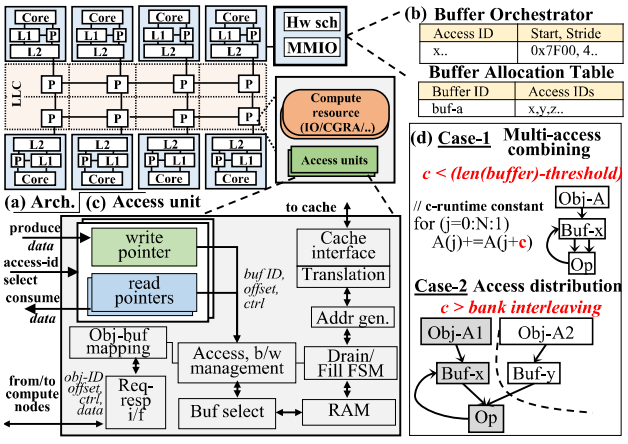



Figure 2. (a) Target architecture, (b) Buffer management, (c) Access unit design and (d) Access mappings.

## IV. The Dist-DA Offload Interface

This section describes how we abstract the target offloadable code and how we map the interface between distributed offloads, along with the necessary architecture and software support. An example of how a code snippet gets mapped using the proposed interface is shown in Figure 3.

### A. Software Support

**Offload abstraction:** Offloadable code regions are abstracted as dataflow graphs (DFGs) of primitive units (see Figure 1e), namely, (i) application memory objects/data structures, (ii) access instructions for required memory transactions, and (iii) operations represented as a set of producers and consumers. The compiler converts the example code snippet shown in Figure 3-1 to a DFG representation as in Figure 3-2 for further analysis and optimizations. The DFG shows the natural dataflow relationship between the data structures involved in the region of offload. Each data structure is assigned a distinct *virtual* object ID, representing a logical memory object. An object can be resident at various levels in the memory hierarchy. The compiler groups the accessors based on the underlying memory object that it can access. This ensures object-level memory access ordering.

**Offload extraction:** The compiler partitions the DFG so that each partition has at most one memory object to localize computation to data, as shown in Figure 3-3. Further, the sum of the communication costs across multiple partitions is minimized by employing graph partitioning algorithms [33]. To enable distributed execution with low control overheads, each partition has a co-located control logic that orchestrates its execution. The orchestrator contains the necessary conditions to iterate a given offload function, based on loop induction variables or the presence of an input value. The communication between the partitioned offload functions is mapped using the interface mechanisms (described next) to enable decoupled producer-consumer execution. Finally, the compiler generates *distributed accelerator definitions*, shown in Figure 3-4 (see Section V).

### B. Interface mechanisms for Dist-DA

To facilitate communication of various kinds of data or control operands among distributed accelerator definitions, we define memory-mapped I/O (MMIO)-based interface mechanisms implemented as software intrinsics, as summarized in Table II. We categorize the mechanisms into four classes.

• **Host-initiated mechanisms**: The host uses these mechanisms to allocate and free a set of accelerator resources at runtime based on the requirements in the application binary. The *cp_config* transfers the offload configurations to the hardware accelerator scheduler. *cp_config_stream/random* allocates an access unit for strided memory accesses or random access, respectively. Additional arguments specify the size and type of accessed data. If the allocation is successful, the hardware scheduler returns the allocated buffer, uniquely

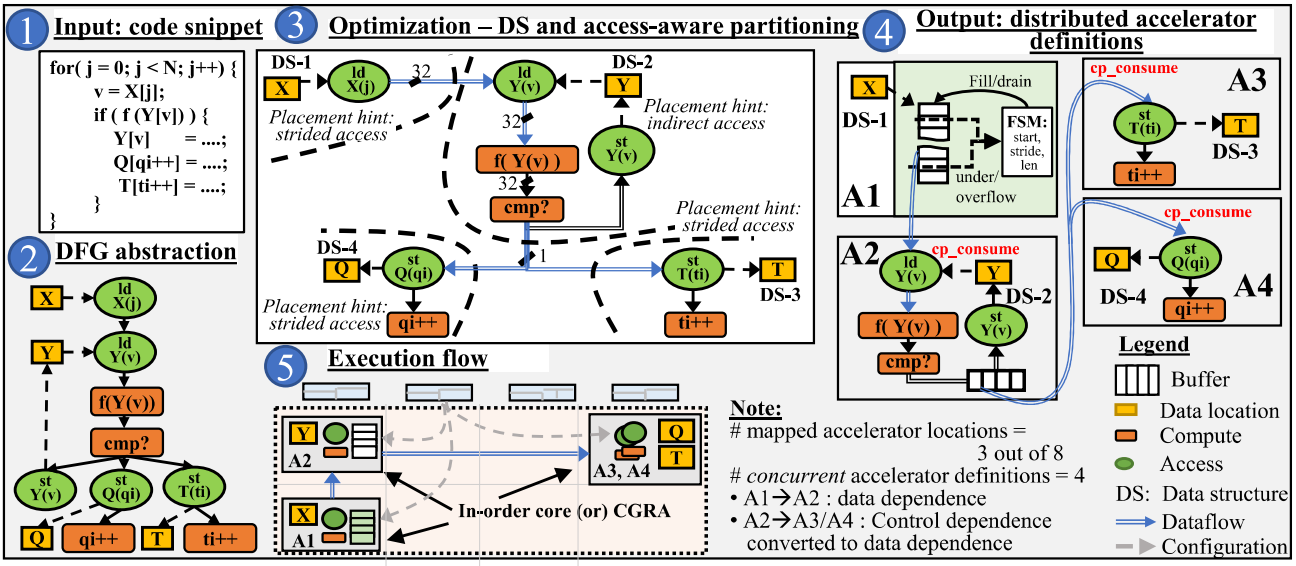



Figure 3. Example of Dist-DA offload model with proposed interface. (1) The input code is abstracted as a DFG of memory object locations, accesses and computations, in (2). (3) Compiler optimizations are done. (4) and (5) show distributed accelerator definitions and execution flow in Dist-DA architecture.

identified by a *buf-id*. The hardware scheduler maintains the *access-id* to *buf-id* mappings per application context in a buffer allocation table, shown in Figure 2b.

Table II
INTERFACE MECHANISMS. THE FOLLOWING MECHANISMS ARE IMPLEMENTED THROUGH MMIO-BASED SOFTWARE INTRINSICS.

| | Mechanism | Operands/Return |
|---|---|---|
| Host | cp_config | offload-id, [args] |
| | cp_config_stream | access-id, start, stride, length, [args]/ buf-id |
| | cp_config_random | access-id, start, end, [args] / buf-id |
| Dataflow | cp_produce | access-id, data |
| | cp_consume | access-id / data |
| | cp_step | access-id, [N] |
| | cp_fill_buf | access-id, [#elements] |
| | cp_drain_buf | access-id, [#elements] |
| Random | cp_write | obj-id, obj-offset, data |
| | cp_read | obj-id, obj-offset/ data |
| | cp_fill_ra | buf-id, [addr], [#elements] |
| | cp_drain_ra | buf-id, [addr], [#elements] |
| Ctrl | cp_set_rf | reg-id, data |
| | cp_load_rf | reg-id / data |
| | cp_run | offload-id |

• **Dataflow mechanisms**: The dataflow-based producer-consumer mechanisms allow memory accesses and communication of intermittent values to be mapped with the same semantics. *cp_produce/consume* writes/reads data from the *access-id*. Spatially-mapped producer-consumer nodes (e.g., as in CGRA) may have an implicit access-id. These enable repetitive transfer of operands between accelerators in a decoupled fashion. Decoupling the units helps for a part of a DFG to run ahead of the rest. Dataflow control *cp_step* skips its read/write pointers through N elements (e.g., a consumer can consume from the buffer corresponding to a given *access-id*, and can optionally *step* its access pointer by one or *N* times). *cp_fill/drain_buf* mechanisms interface with the hardware module (if present) to fill/drain the buffers as per the configured pattern for its associated access-id.

• **Random access mechanisms**: *cp_fill/drain_ra* mechanisms are used to *explicitly* fill/drain the local buffers, analogous to DMA operand transfers. Our automated compiler passes do not currently utilize the *cp_fill/drain_ra* mechanisms. However, Figure 5 and Section VI-D explore examples where these mechanisms are employed via user-specified schedules. *cp_write/read* mechanisms are useful to transfer intermittent data between co-processors with indirect/data-dependent access patterns. These accesses are based on the data structure/memory object ID and its offset. They get translated by the object-buffer mapping module to identify if the data is present in local buffers. If not, these accesses go through the cache interface block to the external memory system.

• **Accelerator control mechanisms**: The register-file access mechanisms *cp_set/load_rf* are used (typically by the host) to transfer scalar values to/from an accelerator register, *reg-id*. The *cp_run* command starts the execution of an offload.

**Interface vs. implementations**: Note that the interface mechanisms separate out the semantics of the offload function from the communication aspects as per requirement R1. This allows other offload models to be designed on top of this interface. For example, Livia's migration scheme can be implemented by invoking *cp_set_rf and cp_run* to transfer operands and invoke an already configured accelerator based on a coin flip, or based on looking up the location of the data as in NSC [10].

In addition to the interfaces shown in Table II, which are evaluated in this work, we also propose that our interface would benefit from a *cp_transfer* intrinsic to transfer values

from a source to destination buffer in batches. However, the expected use cases for *cp_transfer* imply algorithmic changes to explicitly perform these batch transfers, and our compiler passes therefore neither currently use this mechanism, nor is the mapping between additional pragma support and entailed compiler extensions sufficiently obvious that a manual approximation clearly reflects future implementations. We therefore exclude *cp_transfer* from the scope of this work except to note it as a desirable feature of an offload interface.

While this work assumes the target architecture in Figure 2a for the purposes of concrete evaluation, the proposed interface would be equally applicable to any distributed collection of accelerator resources with the following features: A light-weight in-order (IO) processor or configurable processing element (PE), capable of executing a DFG of primitive operations and data access units with local buffering to enable reuse and support latency-insensitive dataflow communication between producers and consumers.

### C. Architectural Support in Accelerators

**Access units:** Figure 2c shows the design of our access unit, the primary functionality of which is to decouple the distributed partitions by acting as an intermediate data-store to provide latency-insensitivity. Each access unit has four main components. (1) The SRAM buffers enable data reuse and decoupling. The accelerators deal only with data structures identified by an *obj-id* and its offset. Every accelerator has a translation block that converts the *obj-id* and offset to its respective physical address. (2) The registers store the current read and write pointers. This helps multiple consumers to access the buffers regulated by the bandwidth management unit. (3) The cache interface block handles memory requests/responses. Credit-based backwards flow-control protocol support is assumed in the network-on-chip (NoC). The communication model can be adapted to support *multi-casting*, depending on the underlying NoC design, by enabling each producer to send values only when there is space in all consuming buffers. (4) Additionally, we implement hardware support for one-dimensional strided patterns in the access unit, given their ubiquity. The finite state machine logic (FSM) tracks the currently accessed data structure offsets, and prefetches or drains data based on the configured stride and buffer occupancy. The address generation units can be combined with the co-located compute resources to map more complex access patterns, such as hashing or N-dimensional strides.

**Reuse:** While our primary aim is to support *distributed, mostly-in-place computation*, it is also important to *reuse* already fetched data. We use a simple hardware scheduler at allocation time to detect buffer reuse among access nodes. Multiple accessors are combined to be co-located along with a local storage unit for temporal and spatial reuse across accesses. Access instructions with overlapping strides are mapped to a single buffer. Figure 2d shows an example, where $c$ is the distance between the accesses on data structure A, along with two examples of access specialization, based on the runtime value of $c$. When the accelerator resources are configured at allocation time, if there is a runtime-determined overlap of access windows and the access distance is less than the buffer overflow limit, these accesses are *combined* by the runtime (Section V-B) to be mapped to the same buffer. However, if the runtime finds that two accesses do not overlap, they can be distributed upon the conditions that the compiler did not identify any dependency violations and the partitioning is profitable. This enables concurrency among non-overlapping accesses and reduces coherence traffic.

### D. System Support

**Translation and Coherence:** The proposed accelerators need to be virtual-memory compatible and fine-grained translation imposes latency overheads. Hence, we map a large contiguous memory space for accelerator-accessible data structures that is managed with a slab allocator for allocations/frees. This mitigates the number of allocations and page translation requests between the host and the accelerator. The accelerator uses offsets into the data structure to access the local buffer and memory. The runtime manages the translations per memory object.

Within the boundaries of the offloaded code, the accesses to data structures are localized to the home bank where they are anchored. However, if the accesses fall outside the current compiler analysis scope for offloadable code, the data will need to be invalidated if the scope of access changes between processor/accelerator domain. This can also be accomplished by flushing the data as conventional NDP designs do [34]. For ease of integration with the rest of the system, each accelerator has an accelerator coherency port (ACP) [35]. All memory requests from the accelerators pass through its *local* ACP. We minimize generating cross-cluster coherence traffic by explicitly managing the accelerator-visible data structures in software and not participating in hardware coherence protocols for co-located accesses. This is possible since we have one serializing point per memory object.

### E. Use cases

**Flexible offload functions:** In Figure 3-4, data movement-aware partitioning of the code region produces flexible accelerator definitions *A1-A4*, to be co-located at the four data structures: X, Y, Q, and T. Our offload interface helps in transferring the control and data dependencies across distributed compute/access nodes in an energy-efficient dataflow model shown in Figure 3-5. Note that the data and control dependencies in the original code region still exist: A2 is data-dependent on A1, whereas both A3 and A4 are control-dependent on A2. The mapped accelerators execute *concurrently* (pipeline-parallelism across *A1-A4*), but these do not parallelize beyond the sequential dependencies that already exist in the application. While our focus is *not* to

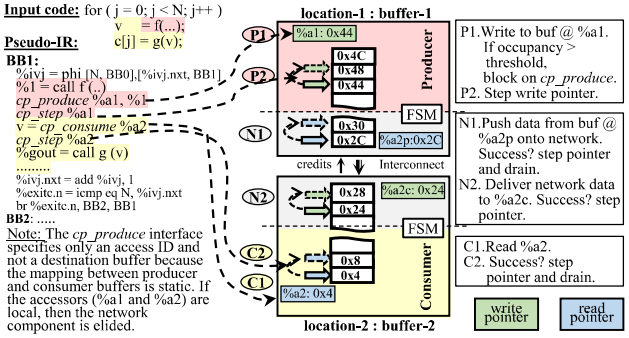



Figure 4. Cross-partition communication mapping.

auto-parallelize applications in this work, our approach does not preclude parallel speed-ups. If we have a task-parallel application, it can be adapted to use the proposed interface.

**Cross-partition communication:** Figure 4 details how a communication edge (intermediate value *v* in the input code) between distributed accelerators at locations-1 and 2 get mapped on the access units for decoupling. *cp_produce* and *cp_step* on access ID *%a1* in the pseudocode writes into the local buffer-1, say at address 0x44 in the example, and steps the pointer by 1 as shown in the figure. This is followed by a read operation at access ID *%a2p* for remote consumer. Although the consumer is remote, the proxy pointer in *%a2p* is maintained locally for memory object-specific access ordering. A buffer is allocated at the consumer for performance decoupling. The data from the network gets written at *%a2c* on buffer-2. The consumer at location-2 uses *cp_consume* and *cp_step* on *%a2* to read data from its local buffer.

**Control communication:** Figure 5a shows how communication can happen across different control domains in a loop nest, assuming the loop nest in the input code gets partitioned into two, one with the outermost loop control and other with innermost loop. The pseudocodes show how the host configures the write (*acc1*) and read (*acc2*) accesses with *cp_config_random* mechanisms, and how the innermost loop bounds (*Ap[i] and Ap[i+1]*) are transferred between partitioned offloads. Partition-1 reads the loop bounds from memory and *produces* onto the buffer, which are later *consumed* by the partition-2 executing the innermost loop.

**Access/communication scheduling:** Figure 5b shows an example for an in-order core with manual access scheduling with *cp_config_random* and *cp_fill/drain* mechanisms, in the absence of a fill/drain FSM hardware block. The host configures two access-IDs to be mapped on the same buffer *s/sr* at the source – a forward-stepping write access *accW* and a reverse-stepping read access *accR*. The third access *accD* at the destination writes into the target buffer, *d*. The partition-1 initially fills the buffer *s* with data from memory object-*src*. This is followed by repeated *consume* and *step* on the filled buffer at partition-1. Concurrently, the partition-2 receives data from the network and writes into the buffer *d*. The buffer *d* is finally drained once the loop in partition-2 finishes.

## V. Compiler Support

This section details our compiler implementation for automating the identification and extraction of offloads. We implement the compiler passes on the LLVM framework [36] to automate the entire compilation flow from partitioning offloadable code regions to extraction of distributed accelerator definitions from application hot-spots. The implemented passes rely on existing LLVM analysis passes, namely, static single assignment (SSA), scalar evolution [37], and memory analyses. Current memory-pattern specialization approaches [23], [24] specialize a memory access pattern in isolation from the rest of the application accesses. As a result, the application characteristics, including its execution and data semantics, are lost in the process of mapping to the underlying architecture. Each memory access gets transformed to an abstract load or store to a virtual address, and loops get transformed into repetitive execution of instructions. However, we note that some of the algorithm and data semantics are visible to the compiler. In fact, current compiler systems try to efficiently represent every memory access and compute instruction in loops as recurrent expressions, or as a function of other static single assignment (SSA) expressions [37]. While the aim of these compiler techniques is to enable efficient mapping of computations by exposing how the variables evolve in a loop, in this work, we leverage this representation to enable efficient decoupling of distributed communicating offload units. Each unit is "self-contained" in terms of control, and connecting these units in a producer-consumer fashion reduces the external control overhead of sequencing multiple accelerator resources. Figure 6 visualizes the compiler support detailed below.

### A. Compilation Steps

All the following steps are *automated* on the LLVM framework with support from the Metis graph partitioning library [38] and CGRA-Mapper [39] for CGRA-based spatial accelerators.

1) **Profiling:** Profiling is used (only) to identify key code regions that contribute to high dynamic instruction coverage. We profile on train/small inputs that are different from the actual datasets used for evaluation in Section VI. While we could apply our technique without this step to any code region whose semantics support offload, not all offloads are likely to be profitable, and hot-code profiling assists our profitability analysis. Further, a more dynamic approach could use run-time hotness, rather than offline code profiling, to determine whether to offload a given computation graph in whole or in part. We consider the former to be an interesting topic for sensitivity studies and workload characterization of offloading under our proposed model and the latter to be

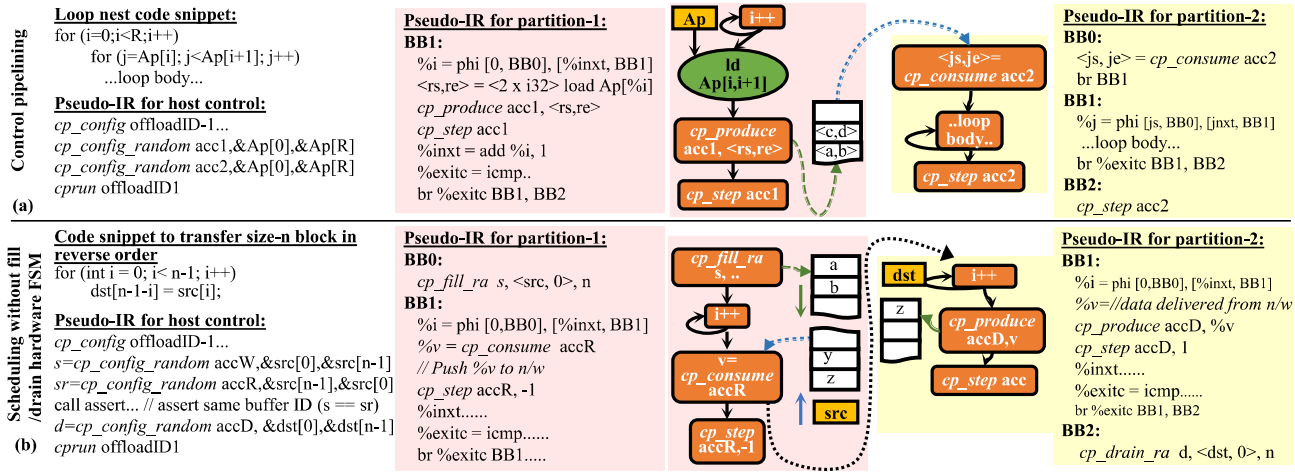



Figure 5. Characteristic use cases of proposed interface mechanisms.

a route to further optimization of our system, but defer the exploration of both to future works.

2) **DFG classification:** The compiler analyzes all loop regions in the target program to identify all basic blocks without high control complexity. The compiler skips considering basic blocks requiring frequent, non-pipelinable host synchronizations for offload. The final set of candidate code regions are used for identifying acceleratable regions.

The compiler abstracts groups of offloadable instructions as accessor and computation nodes in a DFG. All the address computation instructions leading to load or store instruction are grouped together as accessors. The accessors are grouped based on pointer analysis by marking each node with a distinct pointer ID. The rest of the instructions are classified as computation nodes. Additional representative nodes per memory data structure are added to the DFG to indicate the data structure-accessor mapping for further partitioning and placement steps. These three types of nodes are shown in Figure 3-(2). Compiler-based memory alias and dependence analyses [40] are used to identify the underlying pointers that each memory access instruction points to. Access nodes that do not have known memory pointers at compile time are marked with unknown memory object IDs. Control-dependencies in the DFG are converted to data dependencies by predication.

We leverage Scalar Evolution analysis to identify add-recurrent (streaming) patterns of address computations. Static memory dependence analysis of the compiler [40] is further leveraged to conservatively classify each DFG into three cases for partitioning: (1) partitionable accesses and computations, with no memory dependence cycles across loop iterations (parallelizable offload); (2) non-partitionable DFG because of unresolved memory pointers or presence of memory dependence cycles across loop iterations; and (3) partitionable accesses and computations that are non-parallelizable due

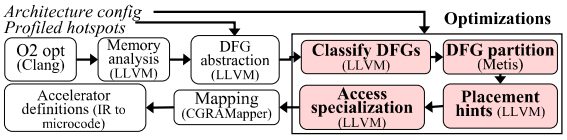



Figure 6. Compilation flow in the LLVM tool-chain (the shaded blocks represent the added optimizations).

to the presence of irregular write accesses (pipelinable). In cases (1) and (3), the computations can overlap with multiple memory accesses and the offload can further be analyzed for partitioning. In case (2) however, the execution needs to be serialized to ensure correctness and, hence our compiler conservatively avoids partitioning. Note that the compiler can add pointer checks and decide whether to offload at runtime if such cases are prevalent in a particular target workload.

3) **DFG partitioning:** The DFG edges connecting various accesses and compute instructions in the abstracted DFG are annotated with communication bit-widths. All nodes are marked with the type of necessary compute resources. The partitionable DFGs are input to Metis [38] following conventional graph partitioning techniques [33] to reduce inter-partition communication cost *while keeping the number of accessed data structures per partition low*. The compiler iterates graph partitioning with increasing number of partitions as a criterion. The pass maintains a history of inter-partition communication cost for various solutions and the number of distinct data structures accessed within a partition. The iterations are performed for a maximum of the number of nodes in the DFG or until there is only one data structure per partition. At the end, the compiler outputs the solution with the lowest inter-partition communication cost and least number of data structures per partition. Non-partitionable DFGs bypass the partitioning step. Partitioning involves clustering various accesses with computations to reduce communication costs, clustering multiple accessors for

exploiting spatial reuse and decoupling long-latency memory accesses by mapping onto a buffer interface as shown in the example on Figure 3-④). The compiler assigns a distinct *access-ID* for such specialized accesses.

4) **Access node placement:** The compiler marks the access nodes with preferred (**vertical**) placement in the memory hierarchy. The preferred location is decided based on memory access pattern characteristics. In general, long strided access are marked to be placed at L3, whereas irregular accesses to shorter sequences are placed closer to the host. Offloading shorter irregular accesses requires more control data to be transferred, and this is not amortized at the LLC. In the example, the compiler pass marks the X, Q, T accesses as strided and Y as indirect accesses.

A DFG partition with both stream and indirect accesses to the same memory object are marked to be co-placed, and the indirect accesses will expend one cycle to check the local buffer before forwarding a request through the memory interface. We adopt a *greedy approach* for the (**horizontal**) placement of access nodes on an LLC cluster. At allocation time, the access nodes are assigned a home LLC cluster based on the address of its first access. However, the placement technique can further be improved in the future with additional runtime support to compare the migration cost with cross-cluster access costs: Then, the partitions with access units placed at L3 can further be decoupled by dynamically migrating the access nodes to banks with data like prior work [10].

5) **Access specialization:** The compiler specializes the edges connecting different partitions with the help of *dataflow* mechanisms in Table II. An example of the mapping is shown in Figure 4. The cross-edges between the partitions are specialized to be mapped onto buffers in the access units for enabling decoupled communication between producers and consumers. The decoupled partitions can execute multiple iterations of the loop ahead of the other partitions. In Figure 3-①, the indirect accesses to Y require ordering, which is already ensured by the clustering of all accesses per data structure. The compiler further specializes the strided access nodes by mapping on access units (Section-IV-C, Figure 3-④-A1), while also inserting the associated configuration intrinsics *cp_config_stream*. Nodes with determinable constant access distance are grouped together as candidates for spatial reuse. As an optional specialization for in-order accelerators, the pass inserts software prefetches for any irregular access without dependency violations.

6) **Offload configurations:** Finally, the compiler outputs accelerator definitions for the partitions based on the available accelerator resources in the target system. In the case of CGRAs, larger DFGs are partitioned further and mapped across multiple tiles. After successful mapping, the offloaded code region is outlined, and *cp_set/load_rf* instructions are inserted for transfer of any scalar operands to/from the accelerators, respectively. The final co-processor(s) configurations are bundled with the application binary.

### B. Execution Flow

- At runtime, depending on the placement hints from the compiler, home nodes get identified as per the greedy approach mentioned in Section V-A-4. The host allocates and configures the accelerator resources for the associated partition identified in the *cp_config* instructions (Figure 3-⑤). This includes loading the translation block for all *objectIDs* used within the partition.
- Each access node with an *access-ID* gets mapped to hardware address generation units (*buf-id*), and each producer/consumer mapped over the interface mechanism gets assigned a write/read pointer interface (see Figure 2c) that is mapped to MMIO space.
- The buffer orchestrator, in Figure 2b, is configured with *start* and *stride* values of the stream to let the buffer hold a window of elements for *reuse* among any co-placed accessors. Accessor-to-memory-object mappings are stored in a runtime data structure managed by the orchestrator and are used to perform multi-access combining as shown in the Figure 2d (case-1). If the access distance is within the address range of the local cache, then these accessors are mapped on distributed accelerator resources. Note that the compiler ensures that only accessors with *constant access distances* between them are identified as target candidates for multi-access combining.
- During execution, accelerators communicate through access buffers, and the access management unit ensures that, if there are consumers with back-pressure, the fill/drain FSM is throttled accordingly. The offload model allows *concurrent execution* of the host and *multiple* accelerators.
- *cp_consume* operations on an empty buffer block until the producer writes a value. This is used for cases where the host waits until the offloaded execution finishes. After execution finishes, the runtime deallocates accelerator resources, if there is no reuse across multiple outer loop iterations.

Table III
SIMULATED PARAMETERS

| | |
|---|---|
| OoO Core | 2GHz, 2x4, X86 ISA, 5-way Ice Lake [41] |
| L1 D/I cache | 8-way 32KB, MSHR-8, latency 2 |
| L2 cache | 128KB, 16-way, MSHR-16, latency 4, stride prefetcher |
| L3 cache | 2MB static NUCA (256KB per cluster) [41] 8 clusters (4 banks per cluster) on mesh NoC, 16-way, MSHRs-64, latency 10 |
| Memory | LPDDR 2GB |
| Accelerator | CGRA @ 1GHz or 1-issue in-order @ 2GHz, 4KB buffer per L3 cluster, ACP - 1-way 1KB |

## VI. EXPERIMENTAL EVALUATION

We use gem5 [48], a cycle-level simulator to simulate the system with the parameters shown in Table III. Table IV lists key characteristics of the evaluated single-threaded workloads. To test the effectiveness of our approach, we choose

Table IV
WORKLOADS.

| Benchmark | Input dataset |
|---|---|
| **Disp**arity [42] | 288x352 images |
| **Trac**king [42] | 288x352 image |
| **FDTD**-2D [43] | 5.8MB |
| **Cho**lesky [43] | 1MB |
| **adi** [43] | 1024x1024 matrix (24MB working set) |
| **Sei**del [43] | 1000x1000 matrix (3.8MB working set) |
| **Path**finder [44] | 6MB matrix |
| **Nw** [44] | 4MB matrix |
| **BFS** [45] | scale-12,edge factor-32 |
| **Page r**ank [46] | 12MB working set |
| **Pointer ch**ase | 8MB uniform distribution |
| **PCA** [47] | 1.2MB working set |

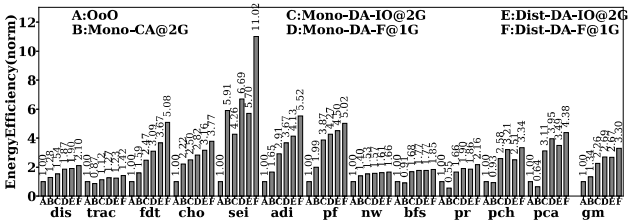


Figure 7. Normalized energy efficiency.

streaming, graph, dynamic programming, dense linear algebra, and data mining applications from Cortex/SDVBS [47] [42], MachSuite [45], Polybench [43] and Rodinia [44]. The accelerator code regions are identified and extracted using compiler passes built on top of the LLVM compiler infrastructure 11.0 [36]. All applications are compiled with Clang and LLVM Opt -O2. The O3 LLVM-Opt passes aggressively unroll the loops with multiple runtime checks by default. This increases the number of configurations of an offloadable code region and the binary size. Fine-tuning the sequence of compiler passes to avoid this issue is ongoing work. Metis [38] is configured for partitioning the offloads. For configurations with CGRA, we use the open-source CGRA Mapper tool [49], [50] to map the offloads. A slab allocator handles memory allocations for the evaluated configurations with accelerators. A cycle-accurate model of a statically-mapped CGRA is implemented in the gem5 simulator. For configurations with in-order (IO) cores, we use an in-order core pipeline based on the gem5 simple cpu model executing custom 64-bit microcodes generated by the compiler. All our applications with accelerator offloads are validated by execution until program completion. We model the dynamic energy for all system components including processor, caches, interconnect, accelerators, access buffers and memory. We use McPAT [51] and Cacti [52] configured for parameters shown in Table III for 32nm technology.

### A. Tested Configurations

The following configurations are evaluated. ① **OoO** – out-of-order core running single-threaded workloads. ② **Mono-CA@L3@2GHz without area constraints** – OoO processor with a monolithic accelerator on the L3 bus with centralized, but stream-specialized, accesses and an 8KB private cache. ③ **Mono-DA-IO @2GHz** and ④ **Mono-DA-F@1GHz without area constraints** – OoO processor with decentralized accesses but without computation partitioning. Both the configurations allow the *reuse* of the locally buffered data. Accelerator resources are implemented with either an IO core@2GHz or CGRA fabric@1GHz, respectively. The Mono-DA-F configuration has a 8x8 CGRA fabric to support larger near-data offloads. ⑤ **Dist-DA-IO@2GHz** and ⑥ **Dist-DA-F@1GHz** – OoO processor with decentralized accesses and distributed computations. Accelerators are based on IO cores@2GHz and 5x5 CGRA fabric@1GHz, respectively (see Section VI-E for area overheads). The results are *normalized* against the OoO configuration, unless explicitly specified.

### B. Energy Efficiency

We illustrate the potential of the offload interface in (1) distributing both compute and accesses and (2) being adaptable to various optimizations. Figure 7 plots the energy efficiency of the configurations for various benchmarks. Distributing and specializing the computations along with decentralizing the accesses (Dist-DA-F) shows 3.3× geometric mean (GM) energy efficiency over a OoO core execution. Because of reduced inter-accelerator traffic and flexibility of the offload interface, Dist-DA-F provides 1.46× GM energy efficiency compared to Mono-DA-IO baseline. Further, the Dist-DA-IO configuration provides a GM energy efficiency of 2.67× over the baseline. It highlights the flexibility of our offload interface to support various accelerator implementations.

**Effect of decentralizing accesses:** Overall, Dist-DA-F@1GHz provides 2.4×, 3.5× and 1.48× GM reduction in data movement (bytes) compared to OoO, Mono-CA@2GHz and Mono-DA-IO@2GHz, respectively. Decentralizing accesses reduces the traffic through the cache hierarchy for all the benchmarks as shown in Figure 8 (remains the same for all DA configurations). The distribution of dynamic accesses between accelerator resources is shown in Figure 9. Whereas the component *intra* measures the traffic internal to an accelerator's local buffers in bytes, *D-A*/*A-A* measures the external traffic between an accelerator and cache hierarchy/remote accelerator, respectively. We note that all applications with good spatial locality have a higher percentage of *intra*, which are more energy-efficient than cache accesses and the associated implicit data movement overheads (*D-A* in Figure 9).
**Effect of sub-computation partitioning:** The Dist-DA configuration associates arbitrary sub-computations with data structures based on compiler hints discussed in Section V-A. As a result, better data movement-aware partitioning reduces inter-accelerator traffic (*A-A* in Figure 9) further compared to Mono-DA configurations. We further show the breakdown of traffic through the NoC in Figure 10. The four components are host-initiated request/response control (*ctrl*) and data traffic (*data*), and inter-accelerator control (*acc_ctrl*) and data overheads (*acc_data*). While specializing access nodes can reduce data traffic across the cache hierarchy, as shown in Figure 8, further partitioning and placement optimization has

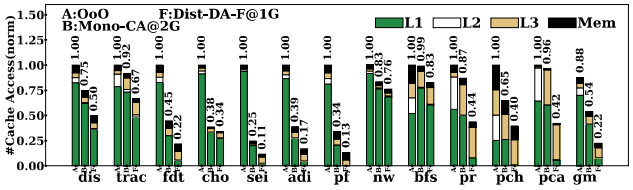



Figure 8. # normalized cache accesses.

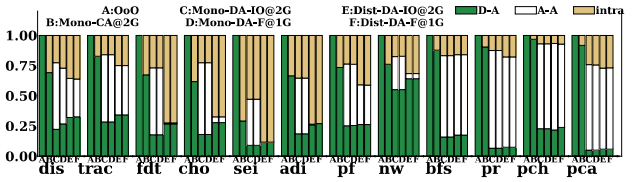



Figure 9. Dynamic access distribution

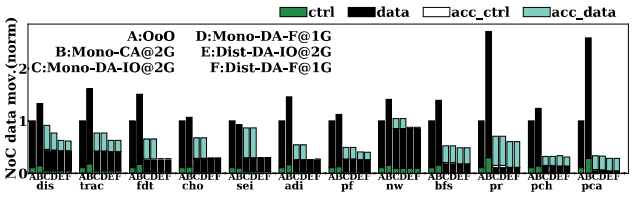
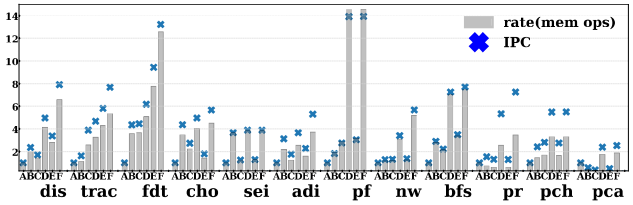
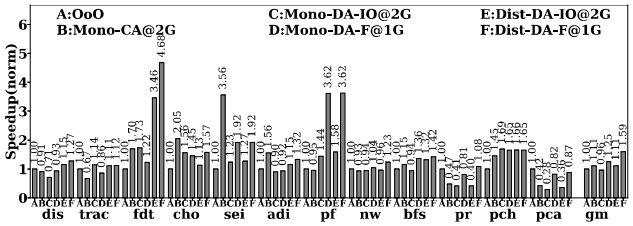



Figure 10. Data transferred through NoC (norm).




(a) Normalized rate of memory operations and IPC.




(b) Normalized speedup

Figure 11. Performance numbers for all configurations.

enabled the computations to take place closer to the cache cluster, as indicated by the reduced inter-compute costs in Dist-DA compared to Mono-DA (*acc_ctrl* and *acc_data*) in Figure 10. Our offload model enables *better* reductions in data movement for workloads where a complex computation requires multiple read operands and where sub-compute partitioning is advantageous (*disparity, tracking, fdtd-2d, cholesky, seidel-2d, adi* and *nw*).

**Effect of compute specialization:** Dist-DA-F@1GHz provides 1.23× GM energy efficiency compared to Dist-DA-IO@2GHz, as besides *near-data accesses and better compute parallelism arising from distribution*, the CGRA improves the instructions per cycle (*IPC*) within each partition as seen in Figure 11a.

### C. Performance

Figure 11b plots the speedup normalized to OoO configuration for the applications. Overall, Dist-DA-F configuration provides a 1.59× and 1.65× speedup over OoO and Mono-DA-IO models. *IPC:* The benchmarks *disparity, tracking, fdtd-2d, cholesky, adi, seidel, serial-implementation of pagerank and pathfinder* have better cache line spatial reuse with all accelerator configurations, which gets reflected in the cache access counts in Figure 8 and hence better IPC in Figure 11a. *Access bandwidth: PCA* has column-major traversals, and the increased access latency with shallow cache hierarchy is in the critical path (note however that the additional L1-L2 traffic is avoided). Since Mono-DA distributes data accesses without data structure grouping, it achieves better access bandwidth per data structure. This is seen in the *tracking* benchmark. Similarly, the larger access bandwidth of a private cache in Mono-CA gives better speedup for *cholesky*. However, as we will show in Section VI-E, users can manually optimize data structure allocation for bank-level parallelism. All the workloads with irregular memory accesses (*bfs, pointer chase*) show better performance in DA configurations, owing to better access locality and bandwidth, whereas OoO and Mono-CA must wait for the data to go through the cache hierarchy. *Clocking:* Benchmarks *adi* and *seidel*, which feature a larger number of complex arithmetic operations, perform better on a faster accelerator Mono-CA@2GHz. Similarly, higher clocking rate in the Mono-DA-IO configuration provides speedup for benchmarks *tracking, fdtd-2d and cholesky*. Dist-DA provides lower NoC traffic, reduced communication latency and better instruction-level parallelism (ILP). All of these play a role in improving the rate of memory operations and IPC for the Dist-DA configurations, as shown in Figure 11a.

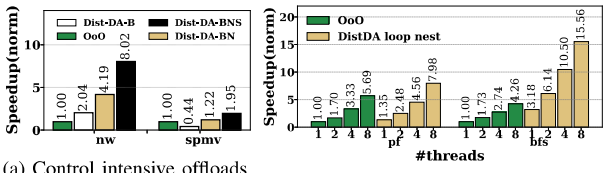
Our evaluation focuses to showcase the wider applicability of the proposed offload model to near-data accelerator resources for energy efficiency, rather than to deeply mine the finely-tuned performance capabilities of a single implementation of the model. Performance speedup can be further improved with more aggressive techniques in unrolling, hardware synthesis, mapping, increased buffer size in access units, and access scheduling (e.g., migration [10]).

### D. Case studies

While the focus of this work has been to evaluate the benefits of distributing computation and accesses within single-threaded applications, we also perform case studies for multi-threaded workloads to show how thread-level parallelism synergizes with the Dist-DA model within a thread. We also provide a case study that exhibits the utility of our non-automated interface mechanisms in offloading control-intensive codes. Table V shows the coverage of interface mechanisms in Section IV-B, differentiating the use of automated interface features in our core evaluation

Table V
COVERAGE OF INTERFACE MECHANISMS USED IN BENCHMARKS AND CASE STUDIES. C: COMPILER AUTOMATED. U: USER ANNOTATED.

| Benchmark / *Case study* | cp_produce | cp_consume | cp_write | cp_read | cp_step | cp_fill_buf | cp_drain_buf | cp_fill_ra | cp_drain_ra | cp_config | cp_config_str | cp_config_ra | cp_set_rf | cp_load_rf | cp_run |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Access unit supports? | | | | | | Y | Y | Y | Y | | | | | | |
| disp | C | C | C | C | C | C | C | | | C | C | | C | C | C |
| trac | C | C | C | C | C | C | C | | | C | C | | C | C | C |
| fdt | C | C | | | C | C | C | | | C | C | | C | C | C |
| cho | | C | | | C | C | C | | | C | C | | C | C | C |
| sei | C | C | | | C | C | C | | | C | C | | C | C | C |
| adi | C | C | C | C | C | C | C | | | C | C | | C | C | C |
| pf | C | C | | | C | C | C | | | C | C | | C | C | C |
| nw | C | C | | | C | C | C | | | C | C | | C | C | C |
| bfs | | C | C | C | C | C | C | | | C | C | | C | C | C |
| pr | C | C | | C | C | C | C | | | C | C | | C | C | C |
| pch | | | | C | | | | | | C | | | C | C | C |
| pca | C | C | C | C | C | C | C | | | C | C | | C | C | C |
| *spmv (annotated)* | U | U | | U | U | U | U | U | U | U | | U | U | U | U |
| *nw (annotated)* | U | U | U | U | U | | | U | U | U | | U | U | U | U |
| *bfs (multi-thread)* | U | U | U | U | U | | | U | U | U | | U | U | U | U |
| *pf (multi-thread)* | U | U | U | U | U | | | U | U | U | | U | U | U | U |




(a) Control intensive offloads.

(b) Multithreaded workloads.

Figure 12. Case studies: Speedup for control-intensive offloads and multithreading workloads.

and programmer-annotation-required features for our control-intensive and multi-threading case studies.

**Interface mechanisms** Although our compiler automation is currently limited to extracting accelerator definitions from the innermost loops of workloads, the proposed interface mechanisms can be employed to larger scopes. We evaluate the use of the interface for workloads with irregular control flow (the *sparse matrix vector multiplication* benchmark [53] with randomly generated dataset – sixteen equally-sized 2D tiles [54] of dimension 4096x4096 with $5 \times 10^{-3}$ sparsity and 2.048 standard deviation in CSR format) and irregular data access patterns (*nw* [44]) by modifying the codes with additional user annotations to exercise specific portions of our proposed interface. We consider three Dist-DA configurations- (1) Dist-DA-B: compiler automated Dist-DA configuration with the application written with loop-*b*locking, (2) Dist-DA-BN: user-identified code regions with *b*locked loop-*n*ests for offloads, and (3) Dist-DA-BNS: user-identified *s*chedule for fetching data *b*locks or for pipelining the control flow in a loop *n*est. Figure 12a shows the speedup of all the configurations normalized to OoO. Whereas loop-blocked implementation of *nw* improves parallelism by reducing data dependencies, Dist-DA-BN localizes the control of the nested loop and pipelines a larger number of loop iterations. On the other hand, Dist-DA-BNS configuration allows multiple stages of scheduling by filling in (*cp_fill_ra*) the data block for computation followed by draining of the output results (*cp_drain_ra*). Offloading of the shorter innermost loops in *spmv* does not amortize the distributed offload as seen by the 0.44× speedup of Dist-DA-B configuration. Offloading a two-deep loop nest with localized control reduces the host-accelerator offload overheads while also providing a 1.22× speedup. Additionally, Dist-DA-BNS shows a 1.95× speedup since it decouples the loop nest control by producing (*cp_produce*) the loop bounds for the innermost loop. The Dist-DA offload interface allows for defining flexible offload functions, which helps in pipelining across multiple invocations of the innermost loop offload. Whereas recent works enhance the processor architecture and/or interface [10], [17], [55] for improved data access efficiency with limited support for offloading computation, our proposed offload interface permits software-directed mapping of both computations and data accesses near-data without complex core modifications.

**Multithreading workloads:** To understand the behavior of our proposed offload model on multithreaded programs, we evaluate two multi-threaded applications (*pathfinder* and *BFS*) from the rodinia benchmark suite [44] by scaling the number of threads (1, 2, 4 and 8 threads). Whereas the proposed compiler flow automatically partitions the application code region subject to the data dependencies within a thread, we currently rely on programmer annotations to identify parallel code regions without read-write synchronizations between concurrent threads. Figure 12b shows the execution speedup for various configurations. All values are normalized to single-threaded OoO configuration. The execution time reduces as the number of threads is increased from 1 to 8 for both benchmarks. Current framework limitations require the parallel iterations of a loop (or loop nest) to be individually scheduled to different threads. Hence, the stream-based access specialization step in Figure 6 is skipped. This prevents even better speedup in *pathfinder* as the spatial locality of accesses across multiple parallel iterations are not exploited. Better pragmas to indicate scheduling *chunks* of loop iterations can improve the speedup. On the other hand, the outer-loop parallelism in *BFS* exposes opportunities to exploit pipelining across innermost loop iterations, and therefore provides consistent speedups with increasing number of concurrent threads.

### *E. Sensitivity analysis*

**Offload characteristics:** Table VI shows various characteristics of the offloaded code regions for all applications. The second column (%cc) indicates the dynamic fraction of instructions (code coverage) in the baseline that get specialized, and the third column (%dc) displays the dynamic memory access coverage capturing the fraction of memory accesses that are offloaded. We note that the identified offloads within the innermost loops have a reasonable code coverage and cover a majority of application memory accesses for most of the benchmarks. The control overhead required to

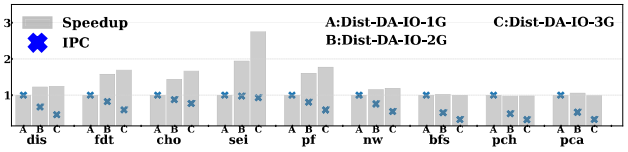



Figure 13. Clocking sensitivity– speedup and IPC norm. to Dist-DA-IO-1G.

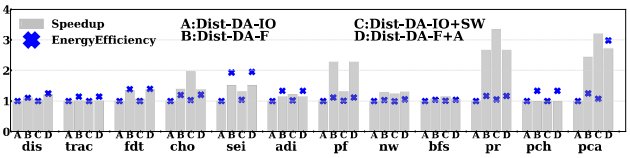



Figure 14. Effect of software-prefetching and allocation optimizations– speedup and energy efficiency norm. to Dist-DA-IO.

initialize and offload these code regions through MMIO accesses is shown as a fraction of total memory accesses in the original application, which forms a small percentage (%init in column-4). Immediate operands configured based on the *cp_config/cp_set_rf* mechanisms also contribute to the MMIO overheads. The average number of local buffers allocated per partitioned offload (#buf in column-5) is at most three across all benchmarks. Table VI also shows the maximum size of the output microcode per offload produced by our compiler pass for the Dist-DA-IO configuration in column-7.

Table VI
OFFLOAD CHARACTERISTICS FOR DIST-DA. COLUMNS: BENCHMARK, %CODE COVERAGE, DATA COVERAGE, INITIALIZATION OVERHEAD, AVERAGE #BUFFERS, MAXIMUM STATIC INSTRUCTIONS & DFG DIMENSIONS AND STATIC INSTRUCTIONS SIZE FOR IN-ORDER CORE.

| | %cc | %dc | %init | #buf | #insts,DFG dim | insts(B) |
|---|---|---|---|---|---|---|
| dis | 95 | 99.55 | 0.57 | 2 | 27, 5x8 | 216 |
| tra | 82 | 99.97 | 0.37 | 3 | 55, 8x8 | 440 |
| adi | 98 | 99.98 | 0.1 | 3 | 28, 8x5 | 224 |
| fdt | 98 | 99.97 | 0.05 | 3 | 35, 9x7 | 280 |
| cho | 95 | 98.54 | 1.73 | 2 | 11, 4x5 | 88 |
| sei | 99 | 99.83 | 0.25 | 3 | 18, 9x5 | 144 |
| pf | 99 | 99.9 | 0 | 3 | 27, 6x7 | 216 |
| nw | 86 | 59.97 | 0.05 | 3 | 9 , 5x8 | 72 |
| bfs | 74 | 80.27 | 1.1 | 1 | 14, 4x7 | 112 |
| pr | 97 | 99.89 | 0.31 | 1 | 28, 6x6 | 224 |
| pch | 99 | 99.9 | 0.78 | 0 | 4 , 2x4 | 32 |
| pca | 98 | 99.94 | 0.66 | 1 | 17, 4x7 | 136 |

**Area:** We use McPAT, Yosys [56], FreePDK45 [57], CGRA-Mapper [49], [50] and scaling equations [58] to estimate the area overhead of the accelerator resources. The Dist-DA configuration with light-weight single-issue in-order cores with multi-threading capability, two complex and two floating point ALUs contributes to an area overhead of 1.9% of one L3 cache cluster at 32nm (0.3% of the entire chip). For the CGRA-based configurations, we assume a *statically-mapped* CGRA architecture as in prior work [39], [50]. The processing units are *heterogeneously* distributed for area efficiency. Employing dynamic per-cycle CGRA scheduling that tolerates flexible memory access latency and better mapping techniques can reduce area requirements further [59], [60]. As shown in Table VI on column-6, the *maximum* number of instructions per input DFG is 55, with a two-dimensional DFG span of 8x8 when ordered topologically (before partitioning). An upper bound provisioning of 5x5 CGRA tile (four float, complex and fifteen integer ALUs) *per* L3 cluster along with buffers and accelerator coherence ports translates to an area overhead of 2.9% per cluster (0.48% of the entire chip). Further performance-area tradeoffs (e.g., increasing vectorization and unrolling factors of loops) have *not* been our focus. We conservatively model the accelerator resource showing the upper bounds of area limitations at a nominal 1GHz timing requirement, assuming the default vector width of 4 and no explicit loop unrolling pragmas.

**Clocking:** We vary the accelerator clocking rate from 1GHz to 3GHz and find that, although the speedup improves for most of the benchmarks, as seen in Figure 13, the IPC reduces prominently for the access-dominated benchmarks. *Seidel-2D*, on the other hand, has higher #arithmetic operations compared to accesses, hence the rate of IPC reduction is low. This further supports our offload model, since we gain more from increasing distributed accelerator-level parallelism (see Figure 11a) than just increasing the clock rate, and exploiting this requires the offload model to support flexible accelerator definitions and energy-efficient communication between distributed accelerators.

**Working set size:** We increase the working set size of the fdtd-2d kernel from 5.8MB to 1.11GB with LLC capacity of 2MB. We find that the performance delay and energy numbers are dominated by the latency and energy of memory accesses. However, the Dist-DA configuration still reduces the *on-chip* data movement by 2.5× to yield an overall energy efficiency improvement of 9.5% compared to the Mono-DA baseline.

**Software optimizations:** To highlight that many optimizations are applicable on our Dist-DA model, we further evaluate two configurations shown in Figure 14. The configurations are normalized to Dist-DA-IO. ① Dist-DA-IO+SW with an issue width of 4 and additional software prefetches inserted in the offloaded code: software prefetching helps to hide the longer L3 latency in most benchmarks with indirect accesses, most prominently for *pca* and *pr*. Higher clocking rate (2GHz) and issue width helps better the performance compared to Dist-DA-F@1GHz, as in *cholesky* with multi-stream reduction pattern and spatial reuse. ② Dist-DA-F+A: We test the sensitivity to allocation by manually customizing the data structure allocations for intra-cluster locality. We find minor improvements in speedup and energy efficiency. This is because our offload scope is limited to the innermost loop that already has intra-cluster locality most of the time. For every outer loop iteration, the home node placement decision is repeated, and this helps keep the computations near data. For any longer offloads, our offload model is extensible to incorporate runtime migration. This can be done at compile-time by incorporating additional logic in the offload code to raise a control request to the host to calibrate home node placement. Otherwise, additional micro-architecture support in the host can help monitor address ranges at sub-data

structure granularity for better runtime placement decisions and handle may-alias address ranges within a data structure, similar to prior work [10].

## VII. Discussion

**Application affinity for Dist-DA:** Of the benchmarks we studied, applications consisting of streaming or irregular memory accesses to a large data structure which are recurrent and/or which can be pipelined across multiple inner-loop iterations are profitable to be mapped on the proposed architecture. Shorter code regions requiring increased host communication and synchronizations, which cannot be pipelined over multiple offload calls, would translate to reduced performance and energy benefits, just as with other offload models. Exploring performance-driven trade-offs for domain-specific customization of access units is one of our future directions.

**Extending the interface to off-chip data residence:** In this work, we evaluate only on-chip offload models to distribute computations for reducing on-chip data movement. However, if the data is resident off-chip, off-chip localization of compute may be preferable. Since the proposed offload interface uses MMIO-based intrinsics to communicate across distributed accelerators, extending it to off-chip memories is possible. Off-chip offloads can follow similar producer-consumer semantics. However, the scale of amortization and offload granularity for which an offload becomes profitable would vary. Such an off-chip offload model should handle the challenges arising when a data structure is spread out across multiple DRAM modules and when the larger off-chip latency overheads are difficult to amortize over fine-grained compute distribution and communication. Further, we expect that many applications with off-chip resident data structures will also have some on-chip resident data structures as well. Our interface should allow both on-chip and off-chip offloads to communicate among each other as appropriate for the given application mapping.

**Utilization:** This work considers mapping of single-threaded workloads with inherent control and data dependencies to distributed accelerators. As a result, although the distributed accelerators are concurrent, the dependencies exerted by the accelerators bring down the effective hardware utilization. To improve utilization and to exploit "accelerator-reuse" [14], a core focus of our future work is exploring accelerator micro-architectures specifically optimized for multi-workload sharing of accelerator resources.

**Accelerator sharing and multi-threading:** Our current evaluation focuses on single-threaded executions, but many multi-threaded applications or multi-process workloads are also sensible for the Dist-DA model. While evaluating the quantitative trade-offs in supporting these scenarios is a subject of ongoing work, two relevant issues are clear: 1) atomicity support and interface generality and 2) resource sharing. Our existing offload semantics already ensure ordering of accesses to an offload-accessed data structure. Mapping synchronization data and operations as an offload task, combined with our support for accelerator-to-accelerator control transfer, should be sufficient to embody necessary atomic operations, although how efficiently this performs is not yet known. For verifying interface generality, we are exploring ways to work alongside the OpenMP runtime, where Dist-DA helps in energy-efficient communication/synchronization within the concurrent but distributed partitions of a software thread, and OpenMP helps manage multiple software threads.

Finally, prior work [61], [62], [63], [64] has noted that specialized logic often shares many common sub-patterns and access patterns. Our future work will investigate how best to exploit this, especially in multi-process workloads, to generate offload descriptions that can perform the union of multiple common patterns with much lower resource entailment than provisioning for all offloads individually.

## VIII. Conclusion

This paper proposes an offload model to enable efficient data and control communication among the various distributed and heterogeneous compute resources that are increasingly becoming common in system hierarchies. Further, we identify a distributed offload abstraction of an input code in a given sequential C/C++ application for efficient near-data partitioning by the compiler. The partitioned offload definitions along with the architecture interface and compiler support help us implement an energy-efficient near-data system. Our near-data accelerator abstractions and offload models are adaptable to various architecture configurations. We show that distributed computation offloading along with decentralizing accesses (Dist-DA-F) provides a geometric mean energy efficiency of 3.3×, 2.46× and 1.46×, data movement reduction of 2.4×, 3.5× and 1.48×, and a speedup of 1.59×, 1.43× and 1.65× compared to an OoO 5-way processor, a monolithic accelerator @ L3 (Mono-CA@L3) and monolithic offload but with decentralized accesses (Mono-DA-IO) for various application benchmarks, respectively.

## Acknowledgment

The authors would like to thank the anonymous reviewers and shepherd for their insightful comments and suggestions. We also thank Adithya Kumar and Dr. Kanchana Bhaaskaran for providing useful feedback on early drafts. This work was funded in part by NSF awards #1822923, #1763681, #2211018, #1931531, #2028929 and #2008398.

## References

[1] R. H. Dennard, F. H. Gaensslen, H.-N. Yu, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted mosfet's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974.

[2] M. B. Taylor, "Is dark silicon useful? harnessing the four horsemen of the coming dark silicon apocalypse," in *Design Automation Conference*. NY, USA: IEEE, 2012, pp. 1131–1136.

[3] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation cores: reducing the energy of mature computations," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. NY, USA: ACM, 2010, pp. 205–218.

[4] A. Dhar, X. Wang, H. Franke, J. Xiong, J. Huang, W.-m. Hwu, N. S. Kim, and D. Chen, "Freac cache: folded-logic reconfigurable computing in the last level cache," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 102–117.

[5] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das, "Compute caches," in *International Symposium on High Performance Computer Architecture*. NY, USA: IEEE, 2017, pp. 481–492.

[6] O. Mutlu, S. Ghose, J. Gómez-Luna, and R. Ausavarungnirun, "Processing data where it makes sense: Enabling in-memory computation," *Microprocessors and Microsystems*, vol. 67, pp. 28–41, 2019.

[7] E. Lockerman, A. Feldmann, M. Bakhshalipour, A. Stanescu, S. Gupta, D. Sanchez, and N. Beckmann, "Livia: Data-centric computing throughout the memory hierarchy," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. NY, USA: ACM, 2020, p. 417–433.

[8] Z. Wang, J. Weng, J. Lowe-Power, J. Gaur, and T. Nowatzki, "Stream floating: Enabling proactive and decentralized cache optimizations," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 640–653.

[9] Z. Wang and T. Nowatzki, "Stream-based memory access specialization for general purpose processors," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, 2019, pp. 736–749.

[10] Z. Wang, J. Weng, S. Liu, and T. Nowatzki, "Near-stream computing: General and transparent near-cache acceleration," *HPCA. https://seanzw. github. io/pub/hpca2022-near-stream-computing. pdf*, 2022.

[11] Intel, "Intel 12th gen hybrid performance and efficiency-cores." [Online]. Available: https://www.intel.com/content/www/us/en/gaming/resources/how-hybrid-design-works.html

[12] ARM, "Arm big.little technology." [Online]. Available: https://www.arm.com/technologies/big-little

[13] A. Branover, D. Foley, and M. Steinman, "Amd fusion apu: Llano," *IEEE Micro*, vol. 32, no. 2, pp. 28–37, 2012.

[14] M. D. Hill and V. J. Reddi, "Accelerator-level parallelism," *Communications of the ACM*, vol. 64, no. 12, pp. 36–38, 2021.

[15] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "Pim-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture," in *Proceedings of the International Symposium on Computer Architecture*. NY, USA: IEEE, 2015, pp. 336–348.

[16] C. Eckert, X. Wang, J. Wang, A. Subramaniyan, R. Iyer, D. Sylvester, D. Blaaauw, and R. Das, "Neural cache: Bit-serial in-cache acceleration of deep neural networks," in *Proceedings of the International Symposium on Computer Architecture*. NY, USA: IEEE, 2018, pp. 383–396.

[17] B. C. Schwedock, P. Yoovidhya, J. Seibert, and N. Beckmann, "Täkō: A polymorphic cache hierarchy for general-purpose optimization of data movement," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 42–58. [Online]. Available: https://doi.org/10.1145/3470496.3527379

[18] M. Orenes-Vera, A. Manocha, J. Balkind, F. Gao, J. L. Aragón, D. Wentzlaff, and M. Martonosi, "Tiny but mighty: Designing and realizing scalable latency tolerance for manycore socs," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 817–830. [Online]. Available: https://doi.org/10.1145/3470496.3527400

[19] A. Pattnaik, X. Tang, O. Kayiran, A. Jog, A. Mishra, M. T. Kandemir, A. Sivasubramaniam, and C. R. Das, "Opportunistic computing in gpu architectures," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2019, pp. 210–223.

[20] Y. S. Shao, S. L. Xi, V. Srinivasan, G.-Y. Wei, and D. Brooks, "Co-designing accelerators and soc interfaces using gem5-aladdin," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.

[21] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam, "Stream-dataflow acceleration," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 416–429.

[22] X. Tang, O. Kislal, M. Kandemir, and M. Karakoy, "Data movement aware computation partitioning," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017, pp. 730–744.

[23] J. E. Smith, "Decoupled access/execute computer architectures," *ACM SIGARCH Computer Architecture News*, vol. 10, no. 3, pp. 112–119, 1982.

[24] T. J. Ham, J. L. Aragón, and M. Martonosi, "Decoupling data supply from computation for latency-tolerant communication in heterogeneous architectures," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 2, pp. 1–27, 2017.

[25] ——, "Efficient data supply for parallel heterogeneous architectures," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 16, no. 2, pp. 1–23, 2019.

[26] S. Baskaran and J. Sampson, "Decentralized offload-based execution on memory-centric compute cores," in *The International Symposium on Memory Systems*, 2020, pp. 61–76.

[27] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "Graphpim: Enabling instruction-level pim offloading in graph computing frameworks," in *International symposium on high performance computer architecture*. NY, USA: IEEE, 2017, pp. 457–468.

[28] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, and G. Reinman, "Charm: A composable heterogeneous accelerator-rich microprocessor," in *Proceedings of the International Symposium on Low Power Electronics and Design*. NY, USA: ACM, 2012, p. 379–384.

[29] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, H. Huang, and G. Reinman, "Composable accelerator-rich microprocessor enhanced for adaptivity and longevity," in *International Symposium on Low Power Electronics and Design*. NY, USA: IEEE, 2013, pp. 305–310.

[30] C. Tan, M. Karunaratne, T. Mitra, and L.-S. Peh, "Stitch: Fusible heterogeneous accelerators enmeshed with many-core architecture for wearables," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 575–587.

[31] Q. M. Nguyen and D. Sanchez, "Pipette: Improving core utilization on irregular applications through intra-core pipeline parallelism," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 596–608.

[32] ——, "Fifer: Practical acceleration of irregular applications on reconfigurable architectures," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 1064–1077.

[33] I. Moulitsas and G. Karypis, "Architecture aware partitioning algorithms," in *International Conference on Algorithms and Architectures for Parallel Processing*. Springer, 2008, pp. 42–53.

[34] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, J. Brockman, A. Srivastava *et al.*, "Mapping irregular applications to diva, a pim-based data-intensive architecture," in *SC'99: Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*. IEEE, 1999, pp. 57–57.

[35] Y. S. Shao and D. Brooks, "Research infrastructures for hardware accelerators," *Synthesis Lectures on Computer Architecture*, vol. 10, no. 4, pp. 1–99, 2015.

[36] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization*. NY, USA: IEEE, 2004, pp. 75–86.

[37] O. Bachmann, P. S. Wang, and E. V. Zima, "Chains of recurrences—a method to expedite the evaluation of closed-form functions," in *Proceedings of the International Symposium on Symbolic and Algebraic Computation*, ser. ISSAC '94. New York, NY, USA: Association for Computing Machinery, 1994, p. 242–249. [Online]. Available: https://doi.org/10.1145/190347.190423

[38] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[39] G. repository, 2022. [Online]. Available: https://github.com/tancheng/CGRA-Mapper

[40] V. Paisante, M. Maalej, L. Barbosa, L. Gonnord, and F. M. Q. Pereira, "Symbolic range analysis of pointers," in *2016 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2016, pp. 171–181.

[41] Agner, "The microarchitecture of intel, amd, and via cpus: An optimization guide for assembly programmers and compiler makers." [Online]. Available: https://www.agner.org/optimize/microarchitecture.pdf

[42] S. K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. B. Taylor, "Sd-vbs: The san diego vision benchmark suite," in *IEEE International Symposium on Workload Characterization*. NY, USA: IEEE, 2009, pp. 55–64.

[43] L.-N. Pouchet and S. Grauer-Gray, "Polybench: The polyhedral benchmark suite.(2012)," *URL http://www-roc. inria. fr/~ pouchet/software/polybench*, 2012.

[44] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IEEE international symposium on workload characterization*. NY, USA: IEEE, 2009, pp. 44–54.

[45] B. Reagen, R. Adolf, Y. S. Shao, G.-Y. Wei, and D. Brooks, "Machsuite: Benchmarks for accelerator design and customized architectures," in *IEEE International Symposium on Workload Characterization*. NY, USA: IEEE, 2014, pp. 110–119.

[46] S. R. Group, "Page rank serial implementation." [Online]. Available: https://github.com/Sable/pagerank-benchmark

[47] S. Thomas, C. Gohkale, E. Tanuwidjaja, T. Chong, D. Lau, S. Garcia, and M. Bedford Taylor, "Cortexsuite: A synthetic brain benchmark suite," in *2014 IEEE International Symposium on Workload Characterization (IISWC)*, 2014, pp. 76–79.

[48] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news,*, vol. 39, no. 2, pp. 1–7, 2011.

[49] C. Torng, P. Pan, Y. Ou, C. Tan, and C. Batten, "Ultra-elastic cgras for irregular loop specialization," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 412–425.

[50] C. Tan, C. Xie, A. Li, K. J. Barker, and A. Tumeo, "Opencgra: An open-source unified framework for modeling, testing, and evaluating cgras," in *2020 IEEE 38th International Conference on Computer Design (ICCD)*. IEEE, 2020, pp. 381–388.

[51] 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 International Symposium on Microarchitecture*. NY, USA: ACM, 2009, p. 469–480.

[52] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "Cacti 7: New tools for interconnect exploration in innovative off-chip memories," *ACM Transactions on Architecture and Code Optimization,*, vol. 14, no. 2, Jun. 2017.

[53] S. R. Group, "Spmv benchmark." [Online]. Available: https://github.com/Sable/spmv-benchmark

[54] C. Giannoula, I. Fernandez, J. G. Luna, N. Koziris, G. Goumas, and O. Mutlu, "Sparsep: Towards efficient sparse matrix vector multiplication on real processing-in-memory architectures," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 6, no. 1, pp. 1–49, 2022.

[55] N. Talati, K. May, A. Behroozi, Y. Yang, K. Kaszyk, C. Vasiladiotis, T. Verma, L. Li, B. Nguyen, J. Sun *et al.*, "Prodigy: Improving the memory latency of data-indirect irregular workloads using hardware-software co-design," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 654–667.

[56] C. Wolf, "Yosys open synthesis suite," 2016.

[57] FreePDK45. [Online]. Available: https://www.eda.ncsu.edu/

[58] A. Stillmaker, Z. Xiao, and B. Baas, "Toward more accurate scaling estimates of cmos circuits from 180 nm to 22 nm," *VLSI Computation Lab, ECE Department, University of California, Davis, Tech. Rep. ECE-VCL-2011-4*, vol. 4, p. m8, 2011.

[59] D. Wijerathne, Z. Li, A. Pathania, T. Mitra, and L. Thiele, "Himap: Fast and scalable high-quality mapping on cgra via hierarchical abstraction," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021.

[60] C. Tan, T. Geng, C. Xie, N. B. Agostini, J. Li, A. Li, K. Barker, and A. Tumeo, "Dynpac: Coarse-grained, dynamic, and partially reconfigurable array for streaming applications," in *2021 IEEE 39th International Conference on Computer Design (ICCD)*. IEEE, 2021, pp. 33–40.

[61] G. Venkatesh, J. Sampson, N. Goulding-Hotta, S. K. Venkata, M. B. Taylor, and S. Swanson, "Qscores: Trading dark silicon for scalable energy efficiency with quasi-specific cores," in *Proceedings of the International Symposium on Microarchitecture*. NY, USA: IEEE, 2011, pp. 163–174.

[62] N. Clark, J. Blome, M. Chu, S. Mahlke, S. Biles, and K. Flautner, "An architecture framework for transparent instruction set customization in embedded processors," in *Proceedings of the International Symposium on Computer Architecture*. NY, USA: IEEE, 2005, pp. 272–283.

[63] T. Nowatzki, V. Gangadhar, and K. Sankaralingam, "Exploring the potential of heterogeneous von neumann/dataflow execution models," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015, pp. 298–310.

[64] M. J. Lyons, M. Hempstead, G.-Y. Wei, and D. Brooks, "The accelerator store: A shared memory framework for accelerator-based systems," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 8, no. 4, pp. 1–22, 2012.