The Non-Uniform Compute Device (NUCD) Architecture for Lightweight Accelerator Offload

Mochamad Asri[1], Curtis Dunham[2], Roxana Rusitoru[3], Andreas Gerstlauer[1], Jonathan Beard[2]
[1]The University of Texas at Austin, Austin, TX, USA
[2]Arm Research, Austin, TX, USA
[3]Arm Research, Cambridge, UK

***Abstract*—Heterogeneous architectures have arisen as a well-suited approach for the post-Moore era. Among them, architectures that integrate programmable accelerators in or near memory are gaining popularity due to the potential advantages of reduced data movement. Such near-memory accelerators benefit from launching a large number of fine-grain tasks to hide memory latency while exploiting bandwidth gains. This requires low-overhead and portable mechanisms for interfacing of accelerators. If not managed carefully, the hard and soft costs of host and accelerator interactions, such as programming and device driver overheads for actuation, context transfer and synchronization can severely limit acceleration benefits.**

We present the non-uniform compute device (NUCD) system architecture as a novel lightweight and generic accelerator offload mechanism that is tightly-coupled with a general-purpose processor core. Different from conventional offload mechanisms that rely primarily on device drivers and software queues, the NUCD system architecture extends a host core micro-architecture to enable a low-latency out-of-order task offload to heterogeneous devices. In the NUCD programming model, a candidate region for offload in the code is marked with a special instruction. The NUCD microarchitecture then accelerates function offloading, actuation, synchronization for out-of-order parallel execution in hardware with little driver or runtime software involvement, while maintaining standard sequential program semantics.

Results demonstrate that the NUCD system architecture can achieve an average performance improvement of 21%-128% over a conventional driver-based offload mechanism. This in turn enables whole new forms of fine-grain task offloading that would otherwise not see any performance benefits.

I. Introduction

Specialized, heterogeneous and accelerator-rich architectures have emerged as viable solutions to address the impending end of traditional semiconductor scaling [1], [2], [3]. In particular, architectures that integrate programmable accelerators in or near memory are gaining attention due to the data movement reduction benefits of placing the processing elements closer to the data [4], [5], [6]. A key challenge in such accelerator-rich systems is programmability and code portability across architectures with diverging, heterogeneous compositions of accelerators. Furthermore, integration of accelerators into any system has to address the costs of offloading, initiation, context transfer and synchronization, which can quickly outweigh any potential acceleration benefits. This is particularly the case for accelerators placed nearer to memory, which rely on task-level parallelism and multi-threading to hide latency. Being able to launch a large number of potentially fine-grain tasks with little programming and offloading overhead is critical for such near-memory accelerators.

Traditionally, interfacing hardware accelerators within a system requires a driver specific for a targeted device [7], [8], [9], [10]. The device driver provides a clean abstraction for the software to handle low-level operations that interface with the accelerator device. These low-level operations include data movement, context management/setup, translation, actuation, and synchronization[1]. Regardless of the process, the device driver overheard must be amortized by acceleration gains with each invocation. Moreover, each new type of hardware accelerator within a system requires integrating a driver specific for each component with each new application, e.g. to manage context transfer and synchronization between device and host. These hard and soft costs of accelerator actuation skew application designers towards coarse-grained accelerator offload for specific applications, in direct opposition to the needs of near-data and accelerator-rich systems.

In this paper, we introduce the non-uniform compute device (NUCD) system architecture (SA) to enable generic, hardware-assisted and driver-less low-latency compute offload to task-level accelerators. The NUCD SA provides a canonical programming model that adds a single instruction to indicate a region of interest to offload. This instruction provides fast identification of critical information to setup an accelerator context, such as code region size, memory footprint/granularity, and output/input registers. The NUCD microarchitecture then handles all offload, initiation, synchronization and forwarding of an accelerator context without driver involvement. NUCD tightly integrates with a general-purpose core's out-of-order mechanisms to maintain standard sequential programming semantics while enabling transparent parallel execution between the host core and accelerators.

In summary, this paper makes the following contributions:

- We propose an instruction set extension to present accelerator tasks to a *host* core in a portable manner.
- We propose a micro-architecture that implements the `nucd` instruction for conditional accelerator offload and parallel/out-of-order execution of offloaded tasks while maintaining sequential program consistency.
- We propose a system architecture that supports lightweight accelerator invocation, context transfer and synchronization tied to the NUCD micro-architecture.
- Using a cycle-accurate simulator [11], we evaluate the NUCD SA for near-memory acceleration of several data-intensive applications. Results show an average system-wide performance benefit of 21%-128% when compared to a more traditional driver-based offload mechanism and up to 2.6× speedup for applications utilizing fine-

[1]There are many variations on this process

Fig. 1. System-wide speedup with and without offload overhead.

grain tasks that achieve no performance gains under a traditional driver model.

The rest of paper is organized as follows. We first present some further motivation in Section II. We then introduce the NUCD programming model and system architecture, along with its micro-architecture in Sections III and IV. We present our evaluation and results in Section V, related work in Section VI, and lastly our conclusions in Section VII.

II. Motivation

Figure 1 shows the system-wide speedup of several benchmarks (details in Section V) where lightweight, fine-grain kernels are offloaded to near-memory accelerators with and without device driver overhead. The speedup shown is normalized to no-offload execution. As can be seen, for some applications, the actual benefit of hardware acceleration is undone by the cost of driver overhead. Instead of experiencing system-wide speedup, offloading these kernels to accelerators using the traditional device driver approach, can be worse than not offloading them in the first place.

Moreover, in complex heterogeneous architectures where compute elements are sprinkled throughout the memory hierarchy, maintaining standard sequential program semantics efficiently is a challenge [12]. Data forwarding and exceptions from remote accelerators can occur naturally. This synchronization and control-flow back to the host core is normally handled by the software device driver (i.e. completion queue). However, leaving such low-level signalling mechanisms to software places a burden on the programmer, which could hamper widespread adoption of heterogeneous architectures. Therefore, a system architecture that provides portable and transparent mechanisms for accelerator offloading while maintaining standard sequential program semantics including exceptions during acceleration is essential.

In the next sections, we detail our approach to address these objectives and associated design challenges.

III. NUCD Programming Model

In this section, we first discuss the NUCD programming model. The NUCD SA employs a `nucd` instruction to initiate and assist in the offloading process. The programmer identifies a region of interest (ROI) in the code by using the `nucd` instruction to mark the offloaded region. This can be done directly in assembly code or by a compiler. In both cases, necessary and relevant offload information is prepared and passed to the micro-architecture through the `nucd` instruction operands. We describe task identification and the `nucd` instruction in the following.

```
  …
  …
  …
for(i=0; i<64; i=i+16){
foo:
  A[i] = 0;
offload:
 #pragma nucd_start
 for(j=i; j<i+16; j++){
     …
     …
     A[i] += …;
   }
 #pragma nucd_end
bar:
  B[0] += A[i];
}
  …
  …
```

```
  …
 mov x1, #0
foo:
 ldr x10, =addr_A
 add x10,x10,x1 LSL #2
 mov x11, #0
offload:
 //set up x6, x5, x4
 …
 nucd x6, x5, x4
 mov x2, x1
 add x3, x2, #16
 loop:
 …
 add x11, x11, …
 add x2, x2, #1
 cmp x2, x3
 blt loop
 str x11,[x10]
bar:
 ldr x11, [x10]
 …
 add x1,x1,#16
 cmp x1,#64
 blt foo
 …
```

Fig. 2. Task identification in the NUCD programming model.

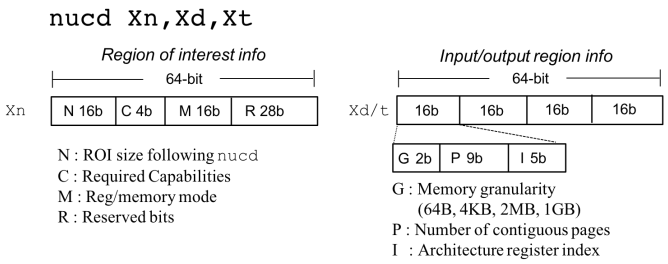


Fig. 3. Register format of the `nucd` instruction.

A. Task Identification

Figure 2 illustrates the offload task identification in the NUCD architecture. The programmer marks the `offload` code block as a ROI for offload in the code using `nucd_start` and `nucd_end` pragmas. The compiler in turn inserts the `nucd` instruction before the instruction block for the `offload` ROI in the assembly code.

B. NUCD Instruction

Figure 3 details the format and content of the three register operands used by the `nucd` instruction. Register `Xn` indicates the base offload information, while registers `Xd` and `Xt` contain more information about the inputs and outputs required by the offloaded task, respectively. `N` in register Xn indicates the size of the offloaded region, i.e. the number of instructions to offload following the `nucd` instruction. `C` denotes capabilities needed by the offloaded code, and `M` contains additional information such as whether the offloaded code uses registers and/or memory locations as inputs and outputs. The rest of the bits are reserved (`R`). For `Xd/Xt`, information about up to four input/output registers or memory regions can be provided. Each 16-bit segment contains 2 bits to describe the memory granule size, 9 bits to describe the size of the memory region given as the number of contiguous granules of memory, and 5 bits to point to a host architecture register index. If the offloaded task uses memory locations, this register index will point to the starting address of that memory

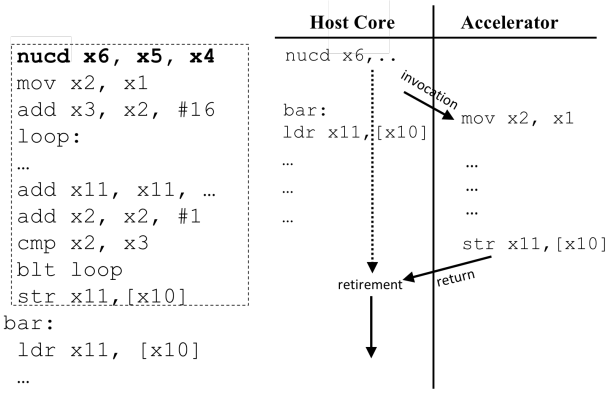


Fig. 4. Offload flow from the host core to the accelerator and back.

location. With these four segments, `nucd` instructions can conceptually support up to four live input and output registers or memory locations. To simplify the register dependence logic, the return value is limited to a single dependent register in our current implementation.

C. Offload Flow

Figure 4 shows the general offload flow for executing the `nucd` instruction. Once the `nucd` instruction and all of its information is decoded, the micro-architecture triggers the offload process. The host core first invokes delegation of the ROI to the accelerator by transferring the context to the accelerator. Once it is sent, the host core continues fetching and executing the instructions following the NUCD sequence (`bar`). At this point, the accelerator and host core can operate in parallel. During acceleration, standard sequential program consistency is maintained through dependency checks in the NUCD microarchitecture. This allows host and accelerator to execute concurrently as long as no dependencies dictated by sequential program semantics are violated. Finally, when the accelerator finishes the offloaded task and returns, the `nucd` instruction is finally retired and the offload completes.

D. Multi-Task Offload

Note that with this support for concurrent execution with the accelerator, the host can encounter multiple `nucd` instructions in the stream and hence can offload several concurrent in-flight NUCD tasks, as shown for the example of a parallelized loop in Figure 2. The outer loop in the figure spawns four NUCD tasks operating independently on the regions specified by the inner loop. As discussed in the previous section, after launching the first task in the first loop iteration, the host core will continue executing instructions and additional loop iterations. Since iterations are independent, this will result in all four tasks to be successively launched and offloaded such that they can execute concurrently. We will explain how we enhance the NUCD micro-architecture to maintain consistency among in-flight tasks in the following section.

IV. The NUCD Micro-Architecture

We describe the NUCD micro-architecture extensions in this section. At its core, NUCD extends the standard out-of-order (OoO) micro-architecture to support a driverless offload process where all procedures are handled by the hardware.

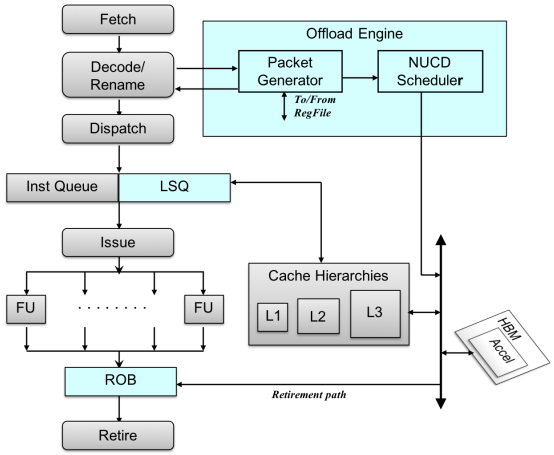


Fig. 5. NUCD system architecture.

Moreover, the extended micro-architecture also enables NUCD to maintain standard sequential program semantics during acceleration. OoO execution between the region of interest and the host core is made possible by integrating NUCD support into the host core's existing dependency tracking mechanisms.

Figure 5 demonstrates the NUCD system architecture. NUCD extensions are marked in blue color. A standard OoO architecture is extended with an offload engine for accelerator invocation, and dependency and retirement tracking integrated into the existing host core load-store queue (LSQ) and reorder buffer (ROB) structures. The offload engine is responsible for handling the accelerator invocation and context transfer process once the `nucd` instruction is decoded. The LSQ and ROB handle dependency tracking and retirement on the return path back from the accelerator while allowing for out-of-order execution between offloaded tasks and the host core. Further details of the offload process and relevant components will be described in the following sections.

A. Accelerator Invocation and Context Transfer

Once a `nucd` instruction is decoded, the host core prepares the accelerator context and performs the context transfer to the accelerator device. All instructions prior to the `nucd` instruction are retired before the `nucd` instruction is issued, effectively creating a sequential barrier. This guarantees that any dependent input registers for the NUCD offload region are ready and have the latest architectural value before the offload begins. Moreover, to preserve sequential memory consistency, the host core flushes any modified lines needed for NUCD execution to memory (i.e., from memory ranges specified through the instruction). This guarantees that the latest data is resident in memory when an accelerator core reads from locations that have been modified but not written to memory prior to offload.

Based on the register/memory information encoded in the `nucd` instruction, the host core then further arranges the context transfer by assembling a *packet* consisting of necessary information for the accelerator to execute the ROI. This includes the program counter of the first and last instruction in the region of interest as well as values of all needed live input registers. Once the *packet* is created, it will be sent to the target accelerators work queue. After the context transfer

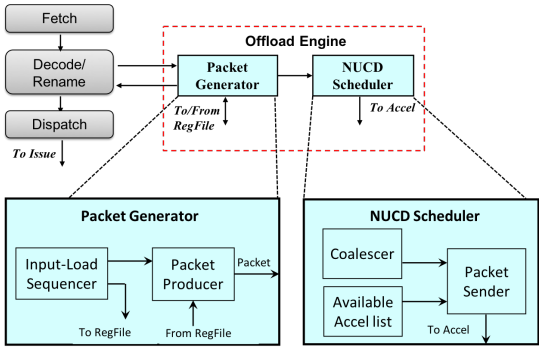


Fig. 6. Extended host core architecture for invocation and context transfer.

completes, the program counter of the host core is set to the instruction following the NUCD sequence.

Figure 6 shows the required extension on the host core architecture to support NUCD invocation and context transfer. A standard OoO core pipeline is enhanced with two necessary components for hardware-offloading purposes: a *Packet Generator* and a *NUCD Scheduler*. The *Packet Generator* obtains signals from the decode stage that identify if a `nucd` instruction is in-flight. The front-end of the core is stalled until the context creation and context transfer is completed. In the beginning of the offload process, the *Input-Load Sequencer* issues a sequence of operations to read relevant registers such as the program counter identifying the beginning of the region of interest and live input registers using encoded information from the `nucd` instruction. Once the relevant information is collected from the register file, the *Packet Producer* assembles the packets for accelerator offload. Afterwards, *NUCD Scheduler* takes the packets and coalesces them if necessary. When the coalesced packet is ready, the *Packet Sender* selects a target from the pool of available accelerators and offloads the final *packets* to the accelerator's memory-mapped work queue.

The offload engine also handles dependency checks to maintain sequential consistency among multiple in-flight `nucd` instructions, i.e. multiple concurrent task offloads. In general, different NUCD tasks can proceed concurrently if they are independent, i.e. if their input and output data regions do not overlap, or if there is only an overlap in their input regions. In all other cases, i.e. if there is an overlap and hence data dependency or anti-dependency between an output region of one task and an input or output region of another task, the younger task is blocked and not allowed to proceed. The NUCD scheduler in the offload engine can be equipped with a dependency table that tracks input and output region dependencies among in-flight NUCD tasks. Alternatively, the memory consistency tracking logic in the load-store queue (LSQ) can be extended to provide task dependency information to the NUCD scheduler. In either case, with such information, the scheduler can delay any task offloads that violate memory consistency until any older dependent task completes.

The packet generator can be seen as a small-entry instruction/data queue for a maximum of four dependent input registers and context data. Moreover, the NUCD scheduler can be designed as a write buffer consisting of few cacheline entries for the context packet, and additional control logic for a coalescer. Such capabilities in general already exist in a standard LSQ design. Thus, the area overhead of the offload engine is similar to that of a conventional LSQ design. Moreover, the main operations performed by the offload engine are majorly register file reads and issuing memory requests. They can run in parallel to the main pipeline and are not on the critical path.

B. Accelerator Execution and Address Translation

Once the context packet reaches the accelerator's work queue, the accelerator loads the context and starts execution of offloaded code. During execution, the accelerator accesses conventional user-space virtual addresses. As such, translation faults can potentially occur during the execution. Any accelerators that need to interact with a virtual memory system via an input/output memory management unit (IOMMU) often can only efficiently run relatively regular applications [13], [14], [15], which are the opposite of the types of applications that benefit from near-memory acceleration. In fact, translation overheads for accelerators can degrade performance by up to 50% [16]. Hence, low-overhead translation mechanism in accelerator devices, especially near-memory ones, are a critical component for performance.

The NUCD SA utilizes a simplified mechanism to mitigate overhead for address translation and memory management on the accelerator side. User-space virtual addresses accessed by offload code are allocated to a physically contiguous memory region by the OS. Translation from virtual to physical addresses on the accelerator core is accomplished using the physical base address of each set of contiguous pages (often termed *range-based* translation [17]). If a page is not loaded in memory, the fault is taken in the host core, freeing the accelerator from having to tolerate memory page faults. In the NUCD SA, the OS memory/slab allocator is modified to allocate virtual ranges that are also contiguous in physical space, which is a common existing approach for many networking and accelerator applications and supported by the Linux kernel allocator *kmalloc* by default [18]. A different design with the ability to handle non-contiguous physical address spaces can also be provided. However, this requires virtual-to-physical translation support and overhead in NUCD devices.

C. Accelerator Retirement and Sequential Consistency

While the accelerator executes the ROI, from the host core's perspective, the `nucd` instruction continues to execute and retires only when the ROI completes on the accelerator. This enables the host core to resume execution of the following instructions past the NUCD sequence. Once execution of the ROI on the accelerator is done, a completion signal is sent to the core from the accelerator devices. Moreover, if the acceleration produces an output, the value of the corresponding accelerator register is transferred back to the host core along with the completion signal. The host core then retires the `nucd` instruction and the entire offload process completes.

To maintain sequential program semantics during concurrent accelerator and host executions, the NUCD SA uses the support of existing out-of-order host core mechanisms for consistency management, namely scoreboarding and the load-store-queue (LSQ) for tracking of read-after-write (RAW) dependencies, and the Reorder Buffer (ROB) for handling of

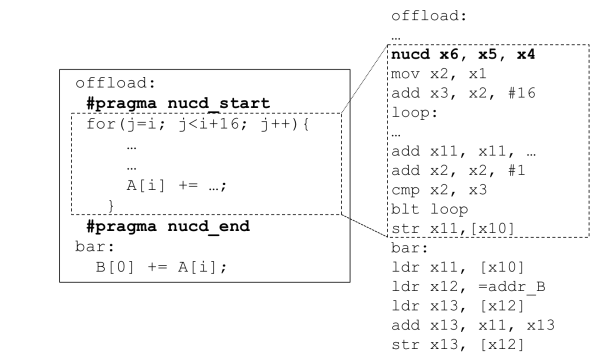
write-after-write (WAW), and write-after-read (WAR) output, and anti-dependencies. We explain how different dependency tracking and sequential consistency management is supported by scoreboarding, LSQ, and ROB in the following subsections.

1) Scoreboarding: For consistency tracking of RAW dependencies between output registers produced by the offloaded task accessed by any following instructions executed on the host core, NUCD makes use of the standard scoreboarding mechanism. Any instructions past the NUCD sequence that are dependent on the output register of the offloaded task will be stalled in the issue stage until acceleration finishes.

2) Dependency Tracking with LSQ: The LSQ further tracks RAW dependencies between any subsequent memory load instructions dependent on the acceleration results. Any reads originating from the host core to NUCD output memory locations that are potentially written by the offloaded task are blocked by the LSQ while independent reads can progress to the issue stage normally. This is achieved by tracking the address range of the output region to which a NUCD task is writing results. The LSQ is augmented to store the base address and number of consecutive pages that are required by the NUCD offload task. Then, every load from the host during acceleration will be checked against the NUCD task's output region to identify whether a RAW dependency exists. In case of a dependency, the LSQ blocks the instruction and schedules it only once the acceleration completes. Any independent load/store instructions can progress without waiting for the NUCD task to retire. This mechanism maintains memory dependency-tracking between the accelerators and the host.

3) Sequential Retirement with ROB: Finally, the ROB ensures that instruction retirement and hence committing of data to registers and memory follows the program order to guarantee sequential consistency of memory and register writes and thus avoid any WAW and WAR hazards between the host CPU and accelerator. While any younger and independent instructions after the `nucd` sequence can be scheduled to issue, their retirement in the ROB has to wait until the acceleration finishes execution. Once the NUCD kernel has finished, a signal is sent back to the ROB to indicate that the operation completed or threw an exception. If an exception is thrown, the operation is handled based on the offset from the NUCD program counter. If no exception has occurred, the return register is committed as the ROB head reaches the `nucd` instruction. Note that the near-memory accelerator will have committed all its memory data by the time it finishes execution and notifies the ROB. The `nucd` instruction is then retired, deallocated and the operation completes.

An alternative design choice can opt for a less conservative synchronization mechanism. It is possible to speculatively let the `nucd` instruction commit from the ROB such that it will not block any younger instructions from committing. However, this requires a checkpoint mechanism to detect violations and safely roll back the architectural state. Prior work has used eviction timing of any modified lines under speculative execution to detect violations and roll back accordingly [19]. In our work, we do opt for the ROB as synchronization point due to its simpler design.



Inst	R/W	Addr	Page Size	# of Pages	ID
nucd x6, x5, x4	W	[addr_A]	4KB	12	NUCD
ldr x11, [x10]	R	[addr_A]	4KB	1	Host
ldr x12, addr_B	R	addr_B	4KB	1	Host
ldr x13, [x12]	R	addr_B	4KB	1	Host
str x13, [x12]	W	[addr_B]	4KB	1	Host

Load Store Queue (LSQ)

nucd x6, x5, x4
ldr x11, [x10]
ldr x12, =addr_B
ldr x13, [x12]
add x13, x11, x13
str x13, [x12]

Reorder Buffer (ROB)

Fig. 7. LSQ and ROB snapshot of the host core during acceleration. An instruction marked as red in the LSQ is blocked until NUCD acceleration completes to satisfy dependencies. Other instructions can execute subject to sequential ROB retirement.

4) Putting It All Together: Figure 7 illustrates an example of LSQ and ROB operation during offload execution. Once decoded, an entry is allocated in the ROB for the `nucd` instruction, similar to standard decoding and ROB allocation in the OoO pipeline. In addition, an entry for the `nucd` instruction is allocated in the LSQ. It contains information about the memory locations to which the NUCD task writes.

The offload process then begins as described in Section IV-A. Once the context transfer completes, the accelerator starts execution of the ROI. At the same time, the host core program counter will branch to the code following the offloaded region (*bar*, with respect to Figure 4). The host core then decodes instructions and allocates ROB and LSQ entries from there. On encountering a load instruction `ldr x11, [x10]` that reads a value from the address region dependent on the NUCD task, the LSQ will block the execution of the load and wait until the NUCD-offloaded task completes (marked in red), therefore avoiding memory consistency violations. By contrast, the following instruction `ldr x12, addr_B` reads a value from a memory location `addr_B`, and since there is no read-after-write dependency between the instruction and the NUCD-offloaded task, the LSQ can proceed and issue the instruction. Moreover, while the following `ldr x13, [x12]` does not depend on the NUCD-offloaded task, it has a read-after-write dependency with the previous instruction (a.k.a `ldr x12, addr_B`). Thus, it will execute once the preceeding instruction completes. Afterwards, the host core performs a store to `addr_B`, which is independent from the NUCD task. As such, the LSQ will go ahead and issue it, subject to sequential retirement in the ROB. The process repeats until there are no more instructions or ROB or LSQ become full.

On task completion, the ROB ensures that retirement from the accelerator to the host core follows sequential order. As shown in Figure 7, while any younger instructions after the `nucd` call are able to issue and execute, their retirement in the

ROB has to wait until the acceleration finishes and the `nucd` instruction retires.

All in all, together with scoreboarding and LSQ tracking, retirement and committing of instructions in program order in the ROB enable the NUCD architecture to maintain sequential memory and register consistency while overlapping accelerator with host core executions. This guarantees standard sequential program semantics during acceleration while exploiting any additional concurrency available on the host core.

V. Experiments and Results

We evaluate the NUCD system architecture using the gem5 full-system simulator [11]. A 64-bit Arm architecture was used as a baseline in a configuration to emulate an out-of-order host core with similar characteristics to an Arm A57 [20]. In-order 64-bit Arm cores (with similar characteristics to an Arm A53 [21]) in configurations with 4, 8, 16, and 32 cores are employed as near-memory accelerators on a cross-bar network interposed between the main coherent network and the HBM1 memory controller (with 8 primary channels). This configuration enables the memory controller to use a standard interleaving pattern while allowing the accelerator and host cores to access the memory fabric at full bandwidth. We use HBM1 as the main memory in a standard configuration [22]. We compare accelerations under NUCD offload with a driver-based mechanism that uses traditional memory-mapped device actuation along with standard doorbell interaction mechanisms following the driver implementation described in [7]. We model NUCD offload engine and context transfer overhead assuming standard register file and memory transaction costs. All simulation parameters are summarized in Table I.

We use several fine-grain benchmarks that represent key data-intensive kernels widely used across applications. Table II summarizes the parameters, inputs, and offloaded ROI characteristics of the benchmarks. Meabo [23] is a multi-phased memory-intensive benchmark. Phase2 (meabo2) and Phase5 (meabo5) are selected for the evaluation. Both phases access memory using a random indirection vector. They employ one and two indirection vectors, respectively. Ebox [24] is an extended box filtering approximation of Gaussian convolution. Stride [25] is a benchmark stressing memory systems with several light compute kernels. SPATTER [26] is a benchmark for timing scatter/- gather kernels. Finally, SpMV [27] is the sparse matrix-vector multiplication in CSR format. We manually partition each ROI call into as many tasks as the number of accelerators, launching one task to one accelerator.

TABLE I
Summary of simulation parameters.

	Host CPUs	Near-Memory Accelerators
ISA	ARMv8 (64-bit)	ARMv8 (64-bit)
Core Configuration	1 OoO core [20]	4-32 in-order cores [21]
L1 I/D Cache	32 KB, 2-cycle	32 KB, 2-cycle
L2 Cache	1 MB, 12-cycle	N/A
HBM Config	HBM Gen1 [22]	
HBM Peak-BW	128 GB/s	

TABLE II
Workload parameters.

Benchmark	Option	Input	Total Dyn. Inst.	ROI	
				Dyn. Inst.	Calls
Meabo2 [23]	Phase2	8,000 elements	73,235	73,100	1
Meabo5 [23]	Phase5	8,000 elements	89,234	89,102	1
Ebox [24]	Stride 8	8,000 elements	392,909	392,701	1
Stride [25]	Distance 8	4,096 elements	1,506,743	40,129	8
Spatter [26]	Distance 8	8,000 elements	83,770	83,450	1
SpMV [27]	N/A	circuit 1 (D)	277,875	277,650	1

A. Speedup

Figure 8 shows the system-wide speedup across different applications with 4, 8 or 16 accelerator cores. We compare the original host-core execution to execution with different near-memory accelerators under driver-based vs. NUCD offload. The figure also indicates the ideal theoretical speedup that can be achieved assuming zero offload overhead for each accelerator configuration.

Results show that accelerations with NUCD offload can yield up to 3.6x speedup across different applications and accelerator configurations, averaging 21%-128% performance improvement over driver-based offload on 4-16 accelerator cores. In all cases, NUCD speedups are equal or close to the theoretically achievable maximum. Closer observation of *meabo2*, *meabo5*, *stride*, and *spatter* shows that NUCD-based offload can unlock performance benefits that do not exist with device driver offload. While driver-based offload results in performance slowdown compared to original host execution even when using 16 accelerator cores, the NUCD mechanism can achieve 1.05x-2.6x speedup for such configurations. Note that even with ideal offload, no performance benefits are observed from offloading to 4 devices across most applications due to the limited compute and reduced bandwidth available with fewer of the simpler accelerator cores. In case of *spmv*, significant slowdowns are seen that do no improve even when the number of accelerator cores increases. This is due to the data access behavior and unbalanced work distribution in *spmv* contributing to poor performance on the in-order accelerator cores and limited task-level parallelism among cores.

B. Offload Overhead

In general, driver offload of a single task to one accelerator core can take up to 40k cycles; predominantly stemming from the overhead of memory-mapping accelerator to the user program. This overhead grows proportionally with the number of tasks and cores. By contrast, in the NUCD SA, such overhead is minimized by delegating much of the process to the hardware, requiring only 150-200 cycles per offload with the configuration given in Table I. Figure 9 illustrates the fraction of execution time spent on offload for different applications under different offload mechanisms. Driver-based offload can occupy up to 70% of total execution time, as seen in *spatter* and *ebox*. Driver overhead averages 18%, 35%, and 47% for 4, 8, and 16 accelerator cores respectively. By contrast, the hardware-assisted NUCD offload mechanism is able to restrict offload overhead to less than 4% of the total execution time. This in turn results in significantly greater system-wide speedup as shown in Figure 8. Note that due to the slowdown experienced under offloading and the longer runtime of the benchmarks, offload overhead occupies only a smaller fraction of total execution time in *spmv* and *stride*. In general, NUCD offload benefits fine-grain tasks the most. In coarse-grained tasks, higher offloading costs are more easily hidden by the longer overall execution times.

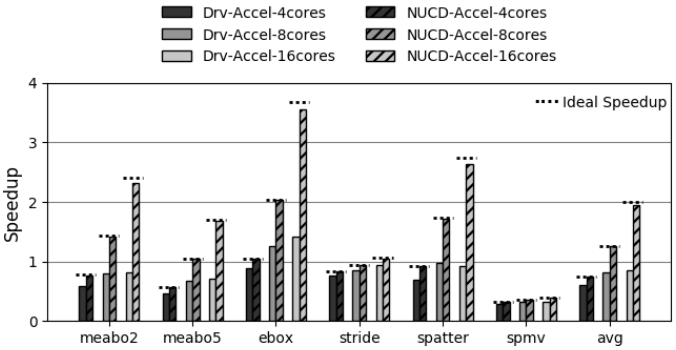


Fig. 8. Speedup under different offload mechanisms with varying accelerators.

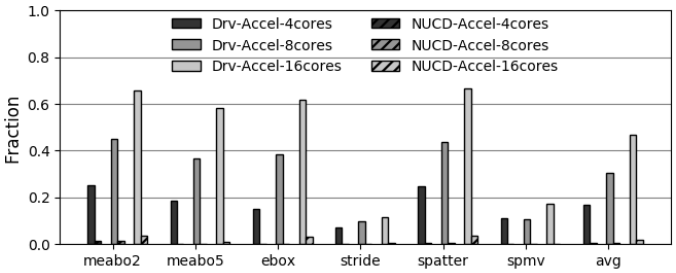


Fig. 9. Fraction of execution time spent on offload.

C. Scaling

Figure 10 further shows the speedup scaling under driver-based and NUCD acceleration as a function of up to 64 accelerator devices, normalized to the speedup with one accelerator core. We observe that *meabo2* under NUCD acceleration scales very well as the number of accelerator devices increases. Acceleration under 8 and 16 cores results in 6.8x and 11x speedup gain, respectively, while utilizing 32 cores improves performance by 16x. By contrast, a driver-based offload of *meabo2* is unable to provide similar scalable performance benefits. As shown in Figure 8, achievable speedups only increase minimally when adding up to 16 cores. This is due to increasing driver overhead outweighing the acceleration benefits with additional cores. Acceleration with 32 cores in turn results in 20% performance slowdown over execution with 16 cores as acceleration benefits saturate while driver overheads continue to increase proportionally.

On the other hand, *stride* experiences a relatively constant speedup as the number of accelerator devices increases for both NUCD and driver offload. While acceleration on four cores in NUCD can boost performance by 1.9x, performance benefits saturate when using 8 or more cores in both cases. This is due to a limited number of instructions within the ROI as shown in Table II, where the offload overhead occupies only a smaller fraction of total execution time in *stride*.

Note that in all cases, without support for out-of-order task offload in NUCD, only one task could be in flight at any time, and speedups would be limited to a single accelerator. Out-of-order execution allows independent NUCD tasks to be simultaneously issued to achieve scalable performance.

VI. Related Work

The increasing demand for energy-efficiency and high performance has spurred a growing number of hardware accelerators as essential parts of future computing systems. Several

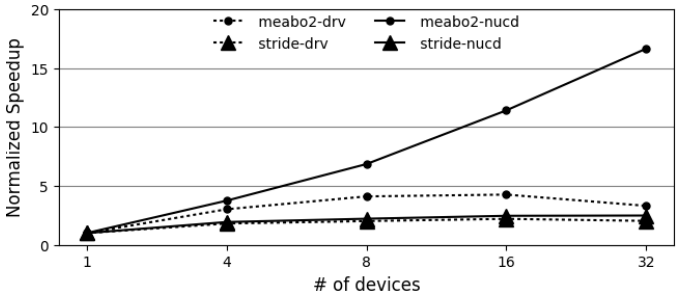


Fig. 10. Performance scaling with number of accelerator cores on *meabo2* and *stride* applications.

designs and optimizations have been proposed for accelerators in various domains [1], [2], [3]. A number of prior works have also shown the performance benefits and energy savings of near-memory accelerators [4], [5], [6]. While the performance gains demonstrated by these works are promising, they place less emphasis on the efficiency of integrating and interfacing such accelerators with the host core.

In practice, several software and/or hardware extensions are required to integrate an accelerator into a system architecture. Hardware optimizations to integrate accelerators typically begin with a mechanism to communicate "jobs" between a host/master and the endpoint device. This is often in the form of a ring buffer or other form of synchronized communication (e.g., VIRTIO [28]). Our mechanism replaces the ring-buffer, however, it does not replace virtualization standards such as SR-IOV (single-root IO Virtualization [29]), our mechanism works with these.

Several software approaches exist in literature that attempt to optimize various aspects of the CPU-accelerator runtime/system software interface. Gdev [9] allows GPUs to access main system memory directly and has been shown to be beneficial for low-latency applications. PTask [10] provides an OS abstraction for GPU computing resource and data transfer management. It presents a dataflow programming model that exposes information to enable the OS kernel to better assist with performance isolation and data movement coordination. Finally, Pagoda [30] presents a runtime system that dynamically manages GPU resources to improve utilization for narrow kernels with limited parallelism where the actual time spent executing (vs. offload overhead) is relatively low. While the above approaches have demonstrated improvements in data and resource management, they do not address optimization of the actual offload process.

Several hardware mechanisms have been proposed to mitigate accelerator offload costs. Lustig et.al. [31] present fine-grained host CPU-accelerator synchronization. Using full-empty bits to track data transfer completion, the accelerator can process data as soon as it is ready without waiting for full offload completion. While this can benefit by amortizing data transfers during offload and kernel computation, the offload process itself is still performed by the runtime software. Other mechanisms such as AMD's extended task queue mechanism [32] and Arm's Revere System Architecture [33] aim to reduce the overheads of interacting with device work queues. However, the actual offload overhead itself is not fundamentally reduced. Other approaches enhance the ISA to

support fine-grain accelerator offload through the host core [4], [34], [35]. However, they target instruction-level offloading and thus are not able to support light-weight tasks that consist of instruction streams larger and more complex than just a simple arithmetic operation.

In summary, existing works fall into either traditional driver-centric offload, hardware-based offloading targeted at instruction-level granularity, or hardware-software mechanisms to accelerate work queues. Driver-based approaches tend to suffer from the non-trivial latency caused by the runtime software and system overhead that can limit the benefit of offload for lightweight tasks. Existing microarchitecture-driven mechanisms are limited to instruction-level offload, and are typically unable to handle lightweight tasks. Lastly, mechanisms that focus solely on acceleration of the work queue tend to reduce the overhead of initiating an accelerator, but do very little for the core-side invocation and targeting of that accelerator.

VII. Summary and Conclusions

We proposed the proposed NUCD system architecture as a general and hardware-assisted accelerator offload mechanism that provides low-latency and low-overhead offload to accelerators. The NUCD system architecture incorporates a task dispatch mechanism that is tightly coupled with the core micro-architecture to perform context transfer, actuation, and synchronization integrated with out-of-order execution. This dispatch mechanism includes features that ensure memory consistency for accelerators outside of the coherence network using a common interface. The NUCD architecture allows maintaining standard sequential program semantics under concurrent acceleration coordinated in hardware from a main core. Results show that the NUCD architecture can improve performance by 21%-128% across different applications compared to traditional driver-based offload.

VIII. Acknowledgements

We thank reviewers for their valuable feedback. This work was funded in part by Arm Research Austin, U.S. DoE FastForward-2 Contract - subcontract No. B609229, and National Science Foundation (NSF) grant CCF-1725743. Special thanks to Kevin Pedretti, Si Hammond (Sandia National Labs), Maya Gokhale (Lawrence Livermore National Labs, LLNL), Bronis de Supinski (LLNL), Giri Chuckapalli (Marvell Inc.), Eric Van Hensbergen, Wendy Elsasser, John Linford, Jose Joao, Alex Rico, Lee Eisen (Arm Inc.) and many others for their helpful comments and review.

References

[1] T. Nowatzki *et al.*, "Stream-dataflow acceleration," in *International Symposium on Computer Architecture (ISCA)*, June 2017.

[2] T. Chen *et al.*, "DianNao: a small-footprint high-throughput accelerator for ubiquitous machine learning," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[3] A. Pedram, A. Gerstlauer, and R. van de Geijn, "Codesign Tradeoffs for High-Performance, Low-Power Linear Algebra Architectures," *IEEE Transaction on Computers (TC)*, vol. 61, December 2012.

[4] J. Ahn *et al.*, "PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture," in *International Symposium on Computer Architecture (ISCA)*, 2016.

[5] M. Gokhale, B. Holmes, and K. Iobst, "Processing in memory: The Terasys massively parallel PIM array," *Computer*, vol. 28, 1995.

[6] D. Zhang *et al.*, "TOP-PIM: throughput-oriented programmable processing in memory," in *International Symposium on High-Performance Parallel and Distributed computing (HPDC)*, 2014.

[7] "Open Source Mali GPUs UMP User-Space Drivers," https://developer.arm.com/tools-and-software/graphics-and-gaming/mali-drivers/ump-user-space.

[8] "Nouveau: Accelerated Open Source driver for nVidia cards," http://nouveau.freedesktop.org.

[9] S. Kato *et al.*, "Gdev: First-class gpu resource management in the operating system," in *USENIX Annual Technical Conference*, 2012.

[10] C. J. Rossbach *et al.*, "Ptask: Operating system abstractions to manage gpus as compute devices," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[11] N. Binkert *et al.*, "The gem5 Simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, May 2011.

[12] L. Lamport, "On programming parallel computers," in *ACM Sigplan Notices*, vol. 10, no. 3, 1975.

[13] N. Amit, M. Ben-Yehuda, and B.-A. Yassour, "Iommu: Strategies for mitigating the iotlb bottleneck," in *International Symposium on Computer Architecture (ISCA)*, 2010.

[14] J. Vesely *et al.*, "Observations and opportunities in architecting shared virtual memory for heterogeneous systems," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2016.

[15] Y. Hao *et al.*, "Supporting address translation for accelerator-centric architectures," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2017.

[16] J. C. Beard and J. Randall, "Eliminating dark bandwidth: a data-centric view of scalable, efficient performance, post-moore," in *International Conference on High Performance Computing (HPC)*. Springer, 2017.

[17] J. Gandhi *et al.*, "Range translations for fast virtual memory," *IEEE Micro*, 2016.

[18] A. Rubini and J. Corbet, *Linux device drivers*. " O'Reilly Media, Inc.", 2001.

[19] A. Boroumand *et al.*, "Lazypim: An efficient cache coherence mechanism for processing-in-memory," *IEEE Computer Architecture Letters (CAL)*, vol. 16, Jan 2017.

[20] "ARM Cortex-A57 MPCore Processor : Technical Reference Manual." Arm Inc., 2013.

[21] "ARM Cortex-A53 MPCore Processor : Technical Reference Manual." Arm Inc., 2013.

[22] D. U. Lee *et al.*, "High-Bandwidth Memory (HBM) stacked DRAM with effective microbump I/O," in *International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, 2014.

[23] ARM, "Meabo," https://github.com/ARM-software/meabo, 2018.

[24] P. Getruer, "A survey of gaussian convolution algorithms," 2013.

[25] "The coral-2 benchmark suite," https://asc.llnl.gov/coral-2-benchmarks/.

[26] P. Lavin *et al.*, "Spatter: A benchmark suite for evaluating sparse access patterns," 2018. [Online]. Available: http://arxiv.org/abs/1811.03743

[27] A. Elafrou, G. I. Goumas, and N. Koziris, "Performance analysis and optimization of sparse matrix-vector multiplication on modern multi- and many-core processors," http://arxiv.org/abs/1711.05487, 2017.

[28] Virtual I/O Device (VIRTIO) Version 1.1. https://docs.oasis-open.org/virtio/virtio/v1.1/virtio-v1.1.html. Accessed October 2019.

[29] Single Root I/O Virtualization. http://www.pcisig.com/specifications/iov/single_root/. Accessed October 2019.

[30] T. T. Yeh *et al.*, "Pagoda: Fine-grained gpu resource virtualization for narrow tasks," in *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2017.

[31] D. Lustig and M. Martonosi, "Reducing GPU offload latency via fine-grained CPU-GPU synchronization," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2013.

[32] M. LeBeane *et al.*, "Extended task queuing: Active messages for heterogeneous systems," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2016.

[33] Arm, "Revere-amu system architecture," https://pages.arm.com/rs/312-SAX-488/images/ARM-IHI-0078A_ALP-03_b_architecture_external.pdf, 2019.

[34] L. Nai *et al.*, "GraphPIM: Enabling Instruction-level PIM offloading in Graph Computing Frameworks," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[35] S. Thoxiyoor, J. Brockman, and D. Rinzler, "PIM lite: a multithreaded processor-in-memory prototype," in *ACM Great Lakes Symposium on VLSI (GSVLSI)*, 2005.