Opportunistic Computing in GPU Architectures

Ashutosh Pattnaik
The Pennsylvania State University

Xulong Tang
The Pennsylvania State University

Onur Kayiran*
Advanced Micro Devices, Inc.

Adwait Jog
College of William & Mary

Asit Mishra
NVIDIA Corp.

Mahmut T. Kandemir
The Pennsylvania State University

Anand Sivasubramaniam
The Pennsylvania State University

Chita R. Das
The Pennsylvania State University

ABSTRACT

Data transfer overhead between computing cores and memory hierarchy has been a persistent issue for von Neumann architectures and the problem has only become more challenging with the emergence of manycore systems. A conceptually powerful approach to mitigate this overhead is to bring the computation closer to data, known as Near Data Computing (NDC). Recently, NDC has been investigated in different flavors for CPU-based multicores, while the GPU domain has received little attention. In this paper, we present a novel NDC solution for GPU architectures with the objective of minimizing on-chip data transfer between the computing cores and Last-Level Cache (LLC). To achieve this, we first identify frequently occurring Load-Compute-Store instruction chains in GPU applications. These chains, when *offloaded* to a compute unit closer to where the data resides, can significantly reduce data movement. We develop two offloading techniques, called *LLC-Compute* and *Omni-Compute*. The first technique, LLC-Compute, augments the LLCs with computational hardware for handling the computation offloaded to them. The second technique (*Omni-Compute*) employs simple bookkeeping hardware to enable GPU cores to compute instructions offloaded by other GPU cores. Our experimental evaluations on nine GPGPU workloads indicate that the LLC-Compute technique provides, on an average, 19% performance improvement (IPC), 11% performance/watt improvement, and 29% reduction in on-chip data movement compared to the baseline GPU design. The Omni-Compute design boosts these benefits to 31%, 16% and 44%, respectively.

ISCA '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6669-4/19/06…$15.00
https://doi.org/10.1145/3307650.3322212

CCS CONCEPTS

• **Computer systems organization** → **Single instruction, multiple data**; **Interconnection architectures**; • **Networks** → *Network on chip*.

KEYWORDS

GPU, near data computing, computation offloading

ACM Reference Format:
Ashutosh Pattnaik, Xulong Tang, Onur Kayiran*, Adwait Jog, Asit Mishra, Mahmut T. Kandemir, Anand Sivasubramaniam, and Chita R. Das. 2019. Opportunistic Computing in GPU Architectures. In *The 46th Annual International Symposium on Computer Architecture (ISCA '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3307650.3322212

1 INTRODUCTION

The memory wall has been a major impediment to designing high performance von Neumann style computing systems. With the recent manycore trend, the cost of moving data from different levels of the memory hierarchy to the cores has further accentuated the memory wall problem [1, 33, 57]. This is likely to become more challenging with technology scaling as more transistors are squeezed into larger dies exacerbating the relative cost of moving data to that of a compute operation. Thus, the performance and energy overheads of data movement would be a continuing non-trivial challenge in designing high-performance, energy-efficient systems.

Processing-In Memory (PIM) and Near Data Computing (NDC) [15, 21, 27, 50, 58] concepts have been proposed to minimize these overheads by moving computation closer to data. Recent advances in technology have made computational logic cheaper, plentiful and easier to integrate. NDC is an abstract framework that requires answering several key design questions — *what to offload*, *where to offload* and *when to offload*. For example, in a traditional multi-level memory hierarchy, computation offloading can be done to the on-chip caches or off-chip DRAM and the granularity of computation would vary depending on where the computation is done.

Traditionally, NDC mechanisms have allowed CPUs to further improve their performance and energy-efficiency.

*This work was started when Onur Kayiran was at Penn State.

However, to our knowledge, the NDC concept has been little explored in the context of GPU architectures. As GPUs are likely to play a major role in achieving energy-efficient Exascale systems, further scaling of their performance and energy-efficiency is a critical task [56]. With technology scaling, the relatively high data movement costs in GPUs are starting to bottleneck further energy-efficiency improvements. The overheads of data movement are greatly accentuated in GPUs due to three major reasons. First, GPU applications are highly data parallel, and therefore, work on large amounts of data. Second, as GPUs scale, compute throughput improvements are outpacing the memory bandwidth improvements, hence, worsening the memory wall problem. Finally, state-of-the-art GPUs have many more cores [14, 48] that need a larger network-on-chip [7, 35] to connect them with memory, leading to an increased data movement and traversal costs. Furthermore, many prior works [16, 17] have identified on-chip interconnect bandwidth to be a bottleneck as well. These reasons have made it imperative to find novel mechanisms to further enhance energy efficiency of GPUs. Recent works [27, 50] have studied the PIM concept in GPUs, where they minimize off-chip data movement by facilitating computation at 3D-stacked DRAMs. To the best of our knowledge, prior NDC efforts do not optimize on-chip data movement in GPUs. Thus, the goal of this paper is to minimize on-chip data movement between the GPU cores and the LLC for improving performance and energy efficiency.

It is non-trivial to make GPUs amenable to NDC mechanism due to three main reasons. First, GPUs are load-store architectures, and the ALU operations are performed only on registers. Therefore, finding candidates for offloading involves searching for suitable instruction sequences. Second, GPUs are SIMT architectures and their fetch, decode and wavefront scheduling units are tuned for hiding memory latency by executing many parallel instructions. Specifically, GPUs try to execute as many instructions as possible from other threads to hide a cache miss, which in turn would lead to longer latencies to offload a set of instructions from a given thread. Hence, efficiently offloading a set of instructions involving multiple loads that incurred L1 misses from a given thread, while minimizing the execution of the other threads is a challenging issue. Third, due to the need for massive memory-level parallelism, data is interleaved across multiple memory partitions. If offloaded instructions require data from multiple partitions, it is not straightforward to find a "single" location to perform the offloaded computation.

The decision of *what to offload* is crucial for the effectiveness of NDC mechanisms. Ideally, one would like to find sequences of instructions that can be offloaded as a whole to minimize data movement. Our NDC mechanism first finds suitable instruction sequences for computational offloading, called *offload chains*, that corresponds to commonly used basic, high level instructions that appear across many applications. We use a compiler pass to tag the offloadable chains. Next, to address *where to offload* the computation, we introduce the term, Earliest Meet Node (EMN), which is the first intersecting node of the loads in the offload chain on their traversal paths. We then form a "*ComputePacket*" of these instructions that can be pushed to the EMN for computation. To perform the offload, we propose two computational offloading mechanisms. The first one, *LLC-Compute*, is employed when the EMN is the LLC node itself. By providing the required computation hardware in the LLC, the computation can be performed there. The second scheme, *Omni-Compute*, is employed when the EMN is an intermediate GPU node in the network; it offloads the computation to another GPU node, which is en route between the source GPU node and LLCs. *Omni-Compute* provides the necessary additional bookkeeping logic to the GPU cores to be able to compute instructions offloaded by other cores beyond the logic needed to execute its own instructions. We show that simply sending all offloadable chains to the EMN for computation is not optimal in terms of performance due to its implications on data locality. Therefore, *when to offload* the chains is critical for the efficiency of the NDC mechanism.

To our knowledge, this is the first work that considers reducing on-chip data movement in GPUs by *opportunistically* offloading computations either to (1) the last-level-caches, or (2) the ALU of another GPU core that would result in the lowest on-chip data movement for that computation. This paper makes the following major **contributions**:

- It proposes two NDC schemes to facilitate computational offloading in GPUs. It shows that basic forms of *load-compute-store instructions* are ideal candidates for offloading.
- It provides (i) compiler support for tagging offloadable instruction chains, (ii) the architectural modifications required for forming *ComputePackets*, (iii) the computational units and (iv) the controller units for LLC-Compute and Omni-Compute with negligible hardware overheads.
- It comprehensively evaluates our proposed NDC mechanisms using nine general purpose GPU workloads. The LLC-Compute technique provides, on an average, 19% and 11% improvement in performance (IPC) and performance/watt, respectively, and 29% reduction in on-chip data movement compared to the baseline GPU design. The Omni-Compute design boosts these benefits to 31%, 16% and 44%, respectively, by providing additional offloading opportunities.

2 BACKGROUND

Programming Environment: The parallel parts of CUDA [47]/OpenCL™ [43] applications, are called "kernels". A kernel contains multiple workgroups. The GPU hardware dispatcher schedules workgroups onto cores. The threads within a workgroup are organized into groups, called "wavefronts". Wavefronts are the granularity at which the GPU schedules threads into SIMD pipeline for execution.

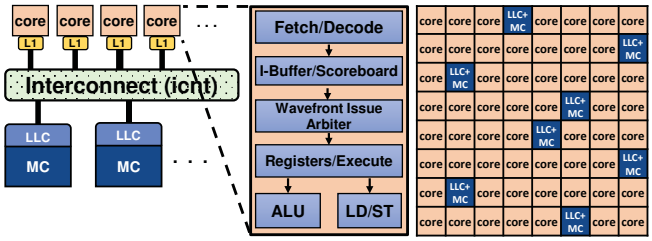

Figure 1: Baseline architecture.

Architecture: The evolution of GPU architecture indicates that the GPU compute capability is scaling along with the number of cores. Recent AMD Radeon™ RX Vega 64 GPUs [14] and NVIDIA® Tesla® V100 GPUs [48] are already equipped with 64 compute units (CUs) and 84 streaming multiprocessors (SMs), respectively. Fig. 1 shows our baseline GPU architecture. Each core has a private L1 data cache, a texture cache and a constant cache, along with a software-managed scratchpad memory (shared memory). The cores are connected to shared last-level cache (LLC) partitions via a Network-on-Chip (NoC). Each LLC partition is directly connected to a memory controller. In our baseline architecture, we model a GPU with 56 cores and 8 last-level cache (LLC)/memory controllers (MCs) connected through an (8×8) mesh-based NoC [31, 65] (YX routing) as shown in Fig. 1. The MC placement in our baseline GPU is a variation of the *checkerboard* configuration, which is shown to be throughput-effective for GPUs when compared with other MC placements [7]. This placement allows for efficient link usage, while minimizing link hotspots that arise due to dimensional routing for traditional MC placements.

Fig. 1 shows the micro-architecture of a core. To execute a wavefront, the fetch unit first fetches the next instruction from instruction cache based on the program counter (PC). The fetched instruction is decoded and placed into an instruction buffer for execution. This buffer is hard partitioned for each wavefront, and the decoded instructions are stored on a per-wavefront basis. The fetch-decode happens in a round-robin manner for all the wavefronts in a core. This keeps the pipeline full so that on a context switch, a new wavefront is ready to be issued immediately. In our baseline configuration, the buffer can hold two decoded instructions per wavefront, limiting only two instructions from a particular wavefront that can be issued continuously for execution. Also, when a wavefront encounters an L1 miss, the wavefront scheduler will switch out the current wavefront with another ready wavefront to hide the memory latency.

3 MOTIVATION AND ANALYSIS

Today's GPUs use separate instructions for memory accesses and ALU operations. Fig. 3 shows an example code fragment of an addition operation performed on operands a and b and stored in operand c. This instruction is broken down into multiple load/alu/store operations in a load-store architecture. The loads fetch the required data from the

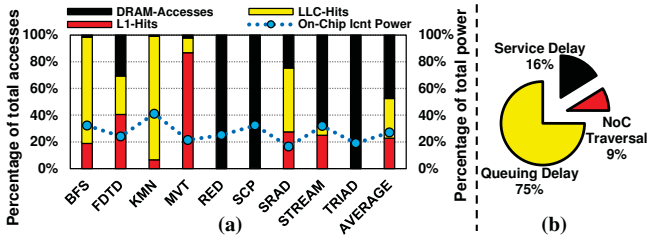

Figure 2: (a) Breakdown of memory requests across the memory hierarchy and the on-chip interconnect power as a percentage of the total GPU power. (b) Percentage of time spent by memory requests (L1 misses) for NoC traversal, queuing delay at the injection/ejection ports and LLC/DRAM service. The average of all applications is shown.

caches/memory into the registers of the core. The data could potentially come from the L1 cache, LLC or the off-chip main memory. Ideally, if all the memory requests are hits at the L1 cache, there will be minimal undesirable data movement apart from the data movement caused by cold misses.

Fig. 2(a) shows the amount of data being fetched from different levels of the memory hierarchy for nine applications. On average, only 23% of the available data is fetched from the L1 cache, while 30% is fetched from the LLC and the remaining 47% is fetched from the DRAM. We provide the detailed performance and energy efficiency evaluation methodology in Sec. 5. Fig. 2(b) shows the breakdown of the average memory latency of requests (L1 misses) across nine applications during their traversal from the cores to the LLC nodes and back. We observe that almost 75% of the memory latency is due to the queuing of the packets in the injection queue at the LLC nodes, 16% of the time is spent servicing these requests, while the rest 9% is taken up by the NoC traversal. NoC traversal constitutes the cost of VC allocation + route computation + switch allocation and link traversal [12]. These observations are mainly due to three reasons: (1) read to write ratio; (2) burstiness of the requests and; (3) traffic patterns in GPUs. Note that, the request traffic sends requests from multiple cores to a few LLCs/MCs, while in the response network, a few LLCs/MCs send data (with a higher payload) to multiple cores. These reasons cause a disparity between the effective request/service rate of the cores (more request packets can be sent) and LLCs nodes (fewer response packets can be sent). To summarize, *77% of the requested data is transferred over the on-chip interconnect, which contributes to an average power consumption of 27% of the total GPU power.* Therefore, the implications are twofold. First, we need to reduce the amount of data transfers over the network, and second, exploit computations rather than waiting for data that is queued up in the network.

3.1 Analysis of Data Movement

To reduce the on-chip data movement, we need to understand and analyze the flow of data from different levels of the

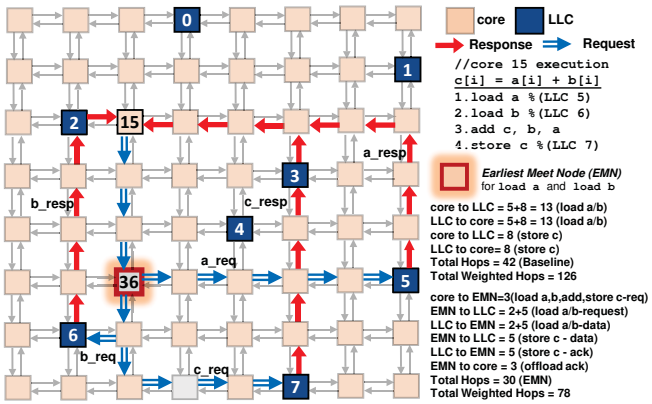

Figure 3: Earliest Meet Node for an instruction sequence (`c[i]= a[i]+b[i]`). For each memory operation, the request and response packets' traversal with YX routing is shown. All memory requests generate from core 15. The two loads and store head to LLC 5, LLC 6 and LLC 7, respectively. For this instruction sequence, the EMN is core 36.

memory hierarchy to the cores. Let us consider two scenarios for the flow of data during the execution of code snippet on core 15 in Fig. 3.

Scenario 1: Let both the load requests and the store request go to the same LLC partition (assume in Fig. 3, the memory requests are *all* en route to LLC 5). When `load a` is executed, a memory request is sent to LLC 5, which takes 8 hops to traverse from core 15 to LLC 5. Upon receiving the request, the LLC partition services it (returns the data if the request hits in LLC; forwards it to memory channel if the request is a miss) and sends a response packet with the data back to the core taking another 8 hops. This is also the case with `load b`. In total, it takes 32 hops to request and get the data back for the two load instructions. Finally, a store request is sent which takes another 16 hops to store the data and receive an acknowledgment. Therefore, to compute `c[i]=a[i]+b[i]`, a total of 48 hops is required for the instruction sequence for each wavefront. Note that, the payload of the packets in the response traffic is considerably larger than request traffic. If we assign weights to the payloads with 1:5 ratio (req/ack:data), the total weighted hop count is 144.

Scenario 2: Let us assume a scenario, where all three memory requests go to different LLC partitions (Fig. 3). In this case, `a` is mapped to LLC 5, `b` is mapped to LLC 6 and `c` is mapped to LLC 7. Getting the data for `load a` takes 16 hops and for `load b`, 10 hops. The `store c` is sent to LLC 7 and takes another 16 hops. Therefore, a total of 42 hops are needed for this computation. By considering the weights for each payload, the total weighted hop count is 126.

3.2 How to Reduce Data Movement?

To reduce data movement, we propose a new concept, called *Earliest Meet Node* (EMN). We observe that on the traversed path for both the load instructions, if we re-route the response messages with load data back to the request node in the same route as the request messages, there is a common node (LLC 5 in Scenario 1 and core 36 in Scenario 2 as shown in Fig. 3) through which the data for both the loads pass, albeit not necessarily at the same time. This node is the EMN for the instruction sequence of a given wavefront. Potentially, the computation (in this case – addition) can be performed at EMN and then, only an ack is sent back to the requesting core from the EMN on a successful computation.

For example, in Scenario 1, both the loads (along with the information of the store) can be sent as a single request (detailed in Sec. 4) from core 15 to the EMN (8 hops to LLC 5), and then the loads can be serviced at the LLC. Assuming the EMN can compute, the operation is performed once both the loads have been serviced. The store is then serviced by the same LLC, and an ack is sent back to core 15 (8 hops) indicating that the offload was successful. Therefore, the entire compute sequence requires 16 hops (reduced by 67%). Furthermore, the total weighted hop count reduces to 16 hops (reduced by 89%) as well. Similarly, for Scenario 2, shown in Fig. 3, if both the loads and the store were sent as a single request (details in Sec. 4) to the EMN (3 hops to core 36), and then if the two loads are split and sent to their respective LLCs (5 hops (LLC 5) + 2 hops (LLC 6) = 7 hops), it would take a total of 10 hops. On their way back, rather than going back to core 15, both `a` and `b` can be sent to the EMN (5 + 2 = 7 hops). After computation, the result (`c`) can be sent directly to the LLC 7 (5 hops). The ack message for the store operation takes another 5 hops to reach the EMN. Finally, from the EMN, an ack notifying of a successful computation is sent to core 15 (3 hops). This approach would require a total of 30 hops (reduced by 29%) and a total weighted hop counts of 78 hops (reduced by 38%). *Note that, we do not change the routing policy (YX) or introduce any network deadlock when we re-route the response packets via the EMN.* We only decouple the response packets from its precomputed route to the core and instead send it to the EMN. This changes the route of the packet, while still following the routing policy. Based on this motivation, we propose a novel GPU design to dynamically build offload chains at runtime and make intelligent decisions for *opportunistically* offloading computations onto a location closest to where the required data is present, while minimizing data movement.

4 OPPORTUNISTIC COMPUTING

The main idea of the NDC mechanism is to first find candidate *offload chains* in a GPU application and to compute this chain *opportunistically* as close as possible to LLCs. To this end, we propose two schemes: **LLC-Compute** and **Omni-Compute**. The first scheme, **LLC-Compute** (Sec. 4.2), reduces data movement for offload chains for which the operands are found in the same LLC partition. An offload

packet, which we call as *ComputePacket*, traverses to the destination LLC for computation and returns the result/ack back to the core. The second scheme, **Omni-Compute** (Sec. 4.3), is built on top of LLC-Compute and increases the coverage of operations, whose data movement can be further reduced, by enabling offloading for chains that request data from two different LLC partitions. In this case, the *ComputePacket* is sent to the EMN (another core), where the computation is performed. We discuss the process of finding the EMN in Sec. 4.3. If the data is placed on different LLCs with no common node in their YX/XY routes, then there is no EMN, and thus, computational offloading is ineffective. For example, in Fig. 3, if core 15 sent two loads to LLC 0 and 6, there is no common node, and therefore, no EMN. Fig. 4 shows the steps in facilitating our proposed mechanisms: (1) identify offloadable chains by using compiler analysis to check for code patterns and register reuse and then tag the offloadable instructions; (2) during execution, efficiently identify and execute offload chain; (3) dynamically decide whether the computation can be offloaded or not; (4) form *ComputePackets* efficiently and keep track of offloaded chains; and (5) enable computation at EMN (LLC/cores).

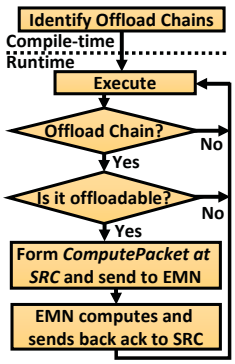

Figure 4: Key steps to realize computation offloading.

4.1 What to Offload?

For applications with good L1 cache locality, offloading loads for computation without caching them locally would increase unnecessary data movement for these loads. Therefore, the ideal candidates for such computation offloading are the applications that are streaming in nature or have a long reuse distances that render the L1D cache ineffective. Note that, LLC-sensitive applications are also going to be beneficial, as our proposed mechanisms reduce the on-chip data movement between the cores and the LLCs.

It is well known that applications have different characteristics during their execution, and even if the entire application cannot be offloaded, certain phases/sequences of instruction from applications can still benefit from computation offloading. Therefore, rather than enabling computation offloading for the entire application, we propose to offload computation at the granularity of an instruction or a set of instructions, which could correspond to a high-level language statement such as the one shown in Fig. 3. Note that, loads with further reuse at the core should not be offloaded, as it will negatively impact the L1D locality. Conservatively, we prioritize L1D locality over computation offloading. For this work, we target such instruction sequences that are amenable to computation offloading and are prevalent in many different types of applications such as machine learning kernels,

Table 1: Prevalent high-level code patterns along with their PTX instructions [46]. The response packet (type and size) details the amount of data that is sent back by the computation node to the source core.

Pattern	1	2	3	4	5	6	7	8	9
HLL code	c=f(a,b)	c=f(a,b,c)	c=a	c=f(a,c)	h(a,b)	h(a, i)	c=f(a,i)	d=f(a,d,g(b,i))	c=f(a,g(b,i))
PTX Code	ld a ld b alu c,a,b st c	ld a ld b alu c,a,b	ld a st c	ld a alu c,a st c	ld a ld b cmp a,b	ld a ld i cmp a,i	ld a ld i alu c,a,i st c	ld b ld i ld a alu c,b,i alu d,a,c	ld b ld i ld a alu c,b,i alu d,a,c st d
Response Packet	ack, 1 flit	data, 5 flits	ack, 1 flit	ack, 1 flit	bitmap, 1 flit	bitmap, 1 flit	ack, 1 flit	data, 5 flits	ack, 1 flit

i = immediate f(x,y),g(x,y) = arithmetic operator h(x,y) = logical operator

linear algebra algorithms, cryptography, high performance computing applications and big-data analytics. Table 1 shows nine types of instruction sequences that we find amenable for offloading. *Note that more offload patterns can be formed using longer instruction sequences, but we concentrate only on patterns that can be packed into a single flit.* Fig. 5 shows the *ComputePacket* format for the largest sequence.

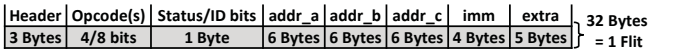

Figure 5: *ComputePacket* format for Pattern 9.

All nine patterns that we offload for computation start with a load instruction and end either with an ALU operation or a store operation. We call this sequence of instructions that can be offloaded as an *offload chain*. For this work, we tag the instructions in the applications that are potentially going to be offloaded at compile-time. These instructions can be easily found with a single compiler pass, and with multiple passes the compiler can even generate statically the data locality information [9] of the loads used by these instructions to make decisions for computation offloading. During execution, these tagged instructions get executed by the hardware, which selectively (based on the L1 locality) generates a single packet for computation offloading.

4.2 LLC-Compute

LLC-Compute offloads chains where *all* the operands are headed towards the same LLC partition. We now describe the steps of LLC-Compute design.

Identification of Offload Chains: In order to ensure that offloading is not performed in the presence of high L1 locality, we use compiler analysis to identify the offload chains that are expected to improve performance if offloaded. The compiler identifies offloadable instructions based on their "per-thread" reuse patterns. Note that our approach does not take into account the reuse distance of the memory request. If it finds no register reuse of the memory request by the same wavefront, it will tag it for offloading.

We analyze the Parallel Thread Execution (PTX) ISA [46] generated by the CUDA compiler [47]. By default, each load/store instruction in the PTX code is preceded by its effective offset calculation that is needed for address generation. We demonstrate this with an example in Fig. 6, which

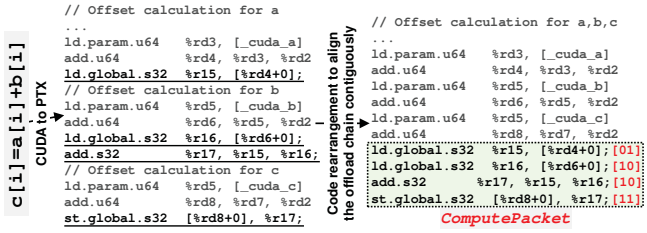

Figure 6: Representative code snippet. The offload chain is tagged and rearranged in the PTX code to align contiguously in memory.

shows a code snippet in high-level language and its corresponding PTX instructions. First, offset for a is calculated, and then a is loaded. The case is similar for b. In our approach, as shown in Fig. 6, we modify the compiler to transform the PTX code so that the offset calculations for the loads/store for the offload chain are executed earlier, and the offload chains' instructions are contiguously stored. This reduces the processing time to form a *ComputePacket*. Also, similar to prior work such as TOM [27], our compiler tags the opcodes of the offloadable instruction sequences with two bits to indicate the first, intermediate and the last instructions in the offload sequence. Specifically, we tag the first load with the bits `01` and then the intermediate PTX instructions with `10` until the final instruction, which is tagged as `11` indicating the end of the chain. Furthermore, these tags also allow for efficient wavefront scheduling that is computation offloading aware, as discussed later in this section.

Hardware Support for Offloading: Fig. 7 shows the required hardware changes (in black) to the baseline architecture for our two proposed mechanisms. It also shows the connections needed for LLC-Compute and Omni-Compute to be integrated with the GPU design. Fig. 8 provides the detailed implementation of the components (Offload Queue and Service Queue). We first describe the hardware additions needed for LLC-Compute. To enable offloading, we add an additional component called the Offload Queue (OQ) (❸), which is responsible for generating, offloading, and maintaining the offloaded chains status. As shown in Fig. 8(a), OQ is made up of three components: Offload Queue Management Unit (OQMU) (❹), Offload Queue Status Register (OQSR) (❺), and *ComputePacket* Generation Unit (CPGU) (❻). The OQMU is the controller unit. It (1) decides whether to offload computation or not based on EMN computation and L1 locality, (2) initiates computation offloading, (3) manages the OQSR, and (4) receives the result/ack for offloaded computation. The OQSR is a 48-entry (Sec. 6.2) status register to maintain the status of the offloaded chains. CPGU is responsible for generating a *ComputePacket* with the computed EMN based on both the load requests' LLC partitions and injecting it into the network for transmission. We give a detailed explanation of how these components are utilized in Sec. 4.4.

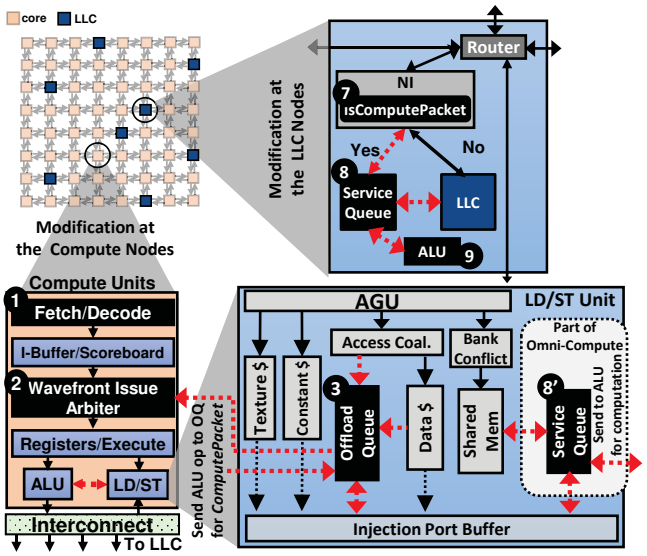

Figure 7: Proposed hardware modification to enable offloading. Additional/modified units are shown in black; The additional unit in Omni-Compute (over LLC-Compute) is the SQ in LD/ST unit 8'.

Efficient ComputePacket Formation: While the OQ is responsible for generating *ComputePackets*, it has no control on how instructions are fetched and decoded, and how wavefronts are issued. Therefore, relying only on the OQ for generating *ComputePackets* is not optimal in terms of performance due to two reasons. First, due to limited instruction buffers (Sec. 2), not all the instructions in an offload chain can be issued close in time. Furthermore, instruction fetch and decode takes place in a round-robin fashion for all the wavefronts, thus, leading to large queuing delays for issuing any remaining instructions in an offload chain of a given wavefront. Second, due to the baseline wavefront scheduling policy (Sec. 2), each load in an offload chain (that results in a cache miss) will cause the wavefront to be switched out for another wavefront. Therefore, in order to issue two loads from the same wavefront, many other wavefronts get executed. This leads to partially filled OQSR entries for many wavefronts. Only when all the offload chain instructions of a given wavefront are executed, a *ComputePacket* is formed. This leads to longer latencies in creating *ComputePackets*. Moreover, the CPGU would need to maintain buffers for each partial *ComputePacket* leading to higher overheads.

To mitigate the effects of wavefront scheduling and avoid the overheads of implementing a CPGU with large buffers, we modify the wavefront scheduling policy (Fig. 7 ❷) along with the instruction fetch and decode logic (Fig. 7 ❶) to prioritize *ComputePacket* formation. We achieve this by making the instruction tags in offload chains known to them. On the first instruction fetch of a wavefront with the tag `[01]`, we prioritize the fetch and decode of this wavefront over other wavefronts. Therefore, whenever a single entry in the

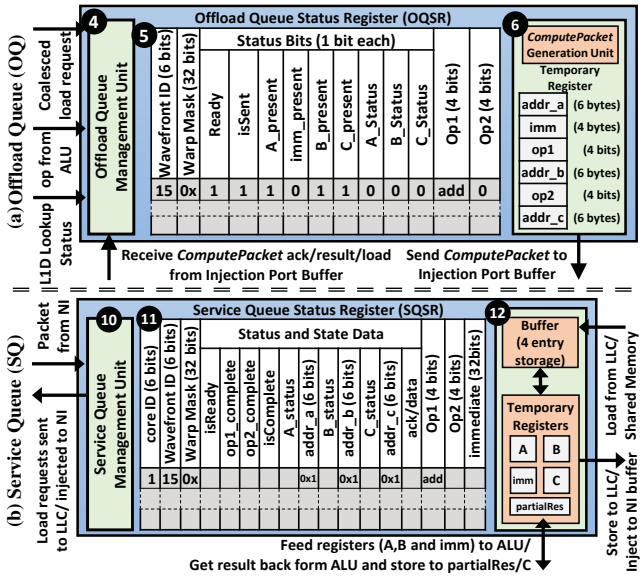

Figure 8: Design of (a) Offload Queue, and (b) Service Queue.

instruction buffer for this wavefront becomes empty, the fetch unit prioritizes the fetch for this wavefront over other wavefronts and decodes it and stores it in the available buffer space. Similarly, on the wavefront scheduler logic, we prioritize the wavefront that is issuing the offload chain. When the final instruction (tagged [11]) in the offload chain has been fetched, decoded and issued, the fetch, decode and wavefront scheduling logics fall back to their default scheduler logic. This reduces the latency for the creation of a *ComputePacket*, enabling better injection into the interconnect by interleaving the generation and injection of the *ComputePackets*. This also minimizes the storage overheads for the CPGU, as it only needs to maintain the full offload chain information for only one wavefront at any given moment. Once an instruction in the offload chain is added to the OQSR, it is treated as committed, after which other instructions in the offload chain are issued and executed, allowing the formation of *ComputePackets*. Only the final instruction in the offload chain will cause the wavefront to stall and wait for an ack.

Hardware Support for Enabling Computation at the LLC Partitions: To enable offloading to LLC partitions, we need to add computation unit, logic to decode *ComputePackets*, and status registers for bookkeeping of the offloaded computations to the LLC partitions. The first addition is a 32-wide single-precision floating point and integer ALU (Fig. 7 ❾) in order to compute the offload chains. We keep it to be 32-wide to maintain the same ALU latency as to that of a core. The second addition is a multiplexer on the network interface (Fig. 7 ❼). The multiplexer directs the packet to the Service Queue or the LLC based on whether the header bit identifies it as a *ComputePacket* or not. The third addition is a component called the Service Queue (SQ) (Fig. 7 ❽), which further comprises of three units as shown in Fig. 8(b): Service Queue Management Unit (SQMU) (Fig. 8(b) ❿), Service Queue Status Register (SQSR) (Fig. 8(b) ⓫), and a temporary buffer (4 entries) (Fig. 8(b) ⓬) to queue up multiple offload chains for computation. The SQMU decodes the received packet, updates the SQSR entries, and generates the required load requests to send to the LLC. SQSR is a 96-entry (Sec. 6.2) status table, and it maintains the addresses of the load and store requests of the offload chains. The SQSR then sends the load requests to the LLC and once data is available in the LLC, the LLC sends signals to the SQ to update the availability bit for the offload chain in the SQSR[1]. Note that, if the load request resulted in an LLC miss, we do not differentiate between the regular memory requests and the memory requests from an offload chain at the DRAM request scheduler. Prioritization techniques can be employed to improve the performance further, but we leave this as potential future work. A temporary buffer holds the offload chains' data. The buffer entries are filled on a first-come first-serve basis for the offload chains. The buffer fills the gap between the time it takes to read required data from LLC and bring them to SQ to feed to the ALU for computation. By having ready entries in the buffer, we can hide the latencies for other requests to be serviced and fetched from LLC into the buffer. If the buffer is full, no new loads are fetched from the LLC. Once an entry is ready to be computed, the data is moved from the buffer to the temporary registers, which are then fed to the ALU. Every cycle, if required, the buffer fetches the required data from LLC based on the status bits of the SQSR entry (to make sure it is present in the LLC), to maintain a set of compute-ready data. Once the computation is finished, SQMU removes the entry from SQSR. It then generates and injects an appropriate response packet to send to the core based on the type of offload chain.

4.3 Omni-Compute

As discussed in Sec. 3, EMN for an offload chain need not be an LLC partition. Thus, we propose Omni-Compute, which adds support for offloading computations to other cores, which are the EMN for a given offload chain.

Unlike LLC-Compute, finding the EMN in Omni-Compute is not straight forward because the required data is present in different LLC partitions. Algo. 1 details the mechanism that is used to find the EMN (example in Fig. 3). For a given GPU configuration, a simple lookup table can be populated using Algo. 1 and be used to find the EMN during GPU execution. The basic idea for finding an EMN is to find a node that is common in the traversal paths (XY or YX routes) of the two loads. If there are multiple common nodes in their paths, EMN is the node that is *closest to both* the LLC nodes.

[1]We do not send data from the LLC until it is needed for computation, which is initiated when all the required operands are present in the LLC. By exploiting the LLC to store data, we avoid any data storage overheads.

Algorithm 1 Finding EMN for 2 loads

```
INPUT: core_node            // Denotes the offloading GPU core node
       llc_nodes[0/1]       // Denotes the destination LLC nodes for the loads
if llc_node[0] == llc_node[1] then return llc_node[0].   //LLC-Compute scenario
for i ← 0 to 1 do
    traversal_path[2i] = xy_path(core_node, llc_node[i])
    traversal_path[2i + 1] = yx_path(core_node, llc_node[i])
for i ← 0 to 1 do
    for j ← 2 to 3 do
        common_nodes.append(traversal_path[i] ∩ traversal_path[j])
for nodes n in common_nodes do
    dist = manhattan_dist(n, llc_node[1]) + manhattan_dist(n, llc_node[2])
    if dist < min_distance then min_distance = dist; EMN = n
return EMN.
```

In order to provide the ability to offload *ComputePackets* to any node, we add a Service Queue (SQ) to all the cores (Fig. 7 ❽'). The SQ used in the cores is different from the one used in LLC partitions only in its input/output connections. Rather than sending load requests directly to the LLC queue, the SQ injects the load requests into the interconnect for transmission to their respective LLC partitions, receives the data from the LLCs, and updates the SQSR accordingly. In LLC-Compute, the SQ uses the LLC as a storage for the loads, whereas, in Omni-Compute, for the SQs in the cores, we reuse the shared memory as the storage structure to buffer the loads as they are received. Once the entry is ready to be computed, the data is transferred from the shared memory to the temporary registers, which are fed to the ALU of the core. Note that SQ only reuses the ALU when it is idle and stalls if it is in use by the core/wavefront scheduler. As Omni-Compute enables computation at any node, a higher number of offload chains can find EMNs and therefore be offloaded. This may leave core resources under-utilized. On the other hand, as Omni-Compute offloads computation to other cores, the SQs at these cores will make use of the ALUs and improve core utilization. Note that the computations and bookkeeping are done by the GPU core. Therefore, we do not make any modification to the router/interconnect.

4.4 How Does Our Mechanism Work?

In this section, we describe how LLC-Compute and Omni-Compute work under multiple scenarios.

4.4.1 Mechanism to Offload Computation. Let us refer to Fig. 6, when the first load instruction (tagged `[01]`) is fetched and decoded. The wavefront that the instruction belongs to is prioritized for fetch and decode until the final instruction in the offload chain that belongs to the same wavefront is fetched and decoded. This wavefront is then de-prioritized. Similarly, when the first instruction gets issued by the Wavefront Issue Arbiter, that wavefront gets prioritized for issuing. Starting from this point, we show how computation is offloaded using Fig. 9. The Wavefront Issue Arbiter issues the first tagged instruction in an offload chain to the LD/ST unit, which forwards it to OQ (①). There could be three possible scenarios after the first load is issued.

Scenario 1: L1 Lookup is a *hit*: The instruction is sent to the OQ and the OQSR is updated (②). Simultaneously, an

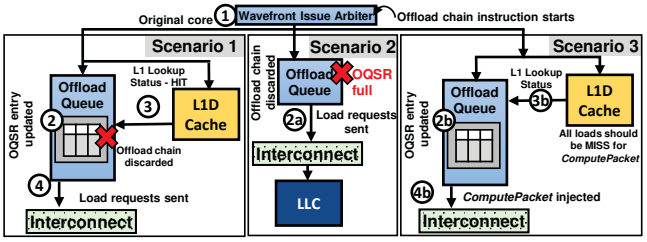

Figure 9: Scenarios for computation offloading.

L1D lookup is performed and the status is returned to OQ (③). In this case, the L1D lookup was a *hit*. This would cause the offload chain to be discarded and regular execution of the instructions to resume. If the load being executed was a second load in the sequence (assuming the first load was a miss), and this was a *hit* in the L1D cache, the OQSR entry is flushed and the first load is sent to the LLC partition (④). Note that, the first load does not execute again but does incur a 1-cycle latency to generate the packet and push into the interconnect. This additional cycle amortizes as the memory requests being fetched from LLCs take 100s of cycles.

Scenario 2: OQSR is full: When the first load in the offload chain is sent to the OQ, and OQSR is full, the offload chain is discarded and the wavefront resumes normal execution (②a). The second load cannot cause this as the first load would have either reserved an entry in OQSR or had been discarded.

Scenario 3: ComputePacket is formed: When all the loads are misses in the L1D cache (③b), if the computed EMN is compute capable, and when the final instruction in the offload chain is issued and the OQSR is filled (②b), a *ComputePacket* is formed and injected into the network (④b) and the wavefront is stalled until the EMN sends back the result/ack to continue execution. If the EMN is not compute capable, the OQSR entry is flushed and the loads are sent individually. For example, in LLC-Compute, only the LLC nodes are capable of computing the offloaded chains.

4.4.2 Mechanism to Execute Offloaded Computation. When a packet is ejected from a router, a header bit is checked to determine if it is a *ComputePacket*. If it is, then it is sent to the Service Queue (SQ) (Fig. 10 ⑤). Fig. 10 shows three possible scenarios when a *ComputePacket* is received.

Scenario 1: EMN is an LLC partition: When a *ComputePacket* is received, the SQMU decodes the packet, fills a new entry in the SQSR and updates the respective bits (⑥). Two loads for a and b are sent to the LLC partition (⑦). When any of the loads is serviced by the LLC (once it is available in the cache), an update message from the LLC is received (⑧), and the status bit for the particular load is set. When both the loads are available, the Ready bit (see Fig. 8 (❺)) is set. Then, the SQ sends load requests to the LLC (⑨), and the data is brought into the buffer ⑩. If the buffer is full, the SQ stalls until the buffer has an empty entry. Once the entry is ready in the buffer, the entry is popped and sent to the temporary registers which transmit the data to the ALU for

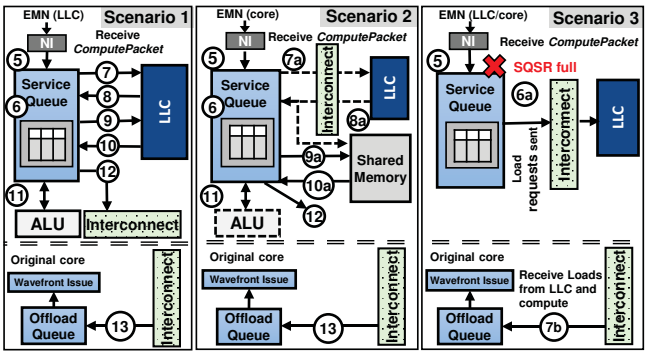

Figure 10: Scenarios when *ComputePacket* is received.

computation and receive the results back (11). Once the result is received, based on the status bits, the store is forwarded to its corresponding LLC partition and an ack is sent to the core that offloaded the computation ((12)). When the core receives the ack, the wavefront resumes regular execution and proceeds to remove the entry from its OQ ((13)).

Scenario 2: EMN is a compute capable core: The computation offloading process at a core is similar to offloading to LLC. The only difference being, the SQ receives the *ComputePacket*, requests for the loads from the respective LLCs ((7a)), fetches ((8a)) and stores them in shared memory ((9a)). The shared memory location is stored in the SQSR rather than the memory address. Similarly, when both the loads are received, the data is fetched from the shared memory to the buffer ((10)). The rest of the process is similar to Scenario 1.

Scenario 3: SQSR at EMN is full: When the SQSR is full (either the ones at the LLC or core), upon a *ComputePacket* being received by the SQ ((5)), the SQMU generates two loads and send them to LLC for servicing ((6a)). It tags the memory requests as non-offloadable, causing the memory requests to go back to the core that offloaded them. These loads will reach the core ((7b)), and the OQMU will check if this was an ack or not. Upon finding a load rather than an ack, the OQSR entry appropriately updates itself to highlight the status of the loads (a_present, b_present). Once the computation is done, the entry is removed from OQSR, a store request is sent, and the wavefront resumes regular execution.

4.5 Limitations of Computation Offloading

In this work, we only consider offload chains whose loads are cache miss to preserve as much locality as possible. Furthermore, the issues of address mapping and data placement play a big role in whether an offload chain can be computed at the EMN or not. For example, due to the LLC placement, there can be offload chains with two load requests without any overlapping nodes during their NoC traversal, and therefore, no computation offloading is performed. Additionally, due to the lock step execution of wavefronts (Sec. 2), applications with high degree of control-flow and irregular access patterns may lead to control-flow divergence and memory-level divergence, respectively. This can cause significant amount of different computations to take place for different threads in a wavefront and multiple memory requests to be generated per wavefront, respectively. This would result in multiple *ComputePackets* from a single wavefront to be generated, leading to higher bookkeeping overheads. In case of only memory divergence, the *ComputePacket* is generated such that each instruction can only generate a single memory request (threads that require other data will be inactive in the warp mask). Currently, for wavefront divergence, we have warp mask bits in OQ and SQ for each entry, while we handle memory divergence by passing the mask bits in another flit attached to the *ComputePacket*. Divergence can be mitigated by smarter data and compute placement [67] or by efficiently forming wavefront dynamically [19]. However, we do not explore such optimizations, but rather provide the hardware design for enabling computation offloading. Also, as shared memory is used by the core and the SQ, we have to make sure it is not overprovisioned. Conservatively, we offload only in situations where the shared memory is large enough to accommodate the needs of both core and SQ. During compilation, we check the shared memory usage in the kernel and disable computation offloading if needed.

Need for Hardware Optimizations: The end result of reducing data movement can also be achieved using compiler analysis to dynamically compute the desired indices (thereby changing the required data) for the threads to work as shown in [61]. However, compiler-based optimizations rely heavily on static analysis and cannot adapt themselves to dynamic behavior of applications/runtime parameters. For example, finding EMN at compile time for all the computations requires prior knowledge of address mapping, data placement, compute placement (thread-block and wavefront scheduling), and other architecture specific parameters. Most of these parameters are not exposed to the compiler, and also change dynamically during runtime (e.g., data migration, compute placement, etc.). Not having all the required *a priori* knowledge will make the analysis of the compiler incomplete. Therefore, it will not completely optimize the computation offloading for reducing data movement. Also, other hardware optimizations such as forming macroinstruction for the offload chains can be performed. But, using macroinstruction will lead to larger hardware overheads as multiple memory requests will need to be decoded concurrently. Therefore, handling a macroinstruction will require multiple fetch/decode/load-store units. If request generation is serialized, it effectively becomes similar to our scheme.

5 EXPERIMENTAL METHODOLOGY

Simulated System: We simulate the baseline architecture as mentioned in Table 2 using GPGPU-Sim v3.2.2 [6]. We extensively modified GPGPU-Sim to implement our proposed schemes. OQ and SQ were added to the GPU datapath and integrated into the core model. We add an ALU and a Service Queue to the LLC datapath. We model the GPU cores,

Table 2: Configuration parameters of the GPU.

GPU Features	1.4GHz, 56 cores, 32 SIMT width, GTO wavefront scheduler
Resources/Core	48KB shared memory, 64KB register file, Max. 1536 threads (48 wavefronts, 32 threads/wavefront)
Private Caches/Core	16KB L1 D-cache, 12KB T-cache, 8KB C-cache, 4KB I-cache, 128B block size
L2 Cache	0.5MB/Memory Partition, 16-way 128 KB Line size
Memory Model	8 MCs, FR-FCFS, 8 banks/MC, 1000 MHz Partition chunk size 128 bytes, 500 GB/s peak BW
GDDR5 Timing	t_{CL} = 11, t_{RP} = 11, t_{RC} = 39, t_{RAS} = 28, t_{CCD} = 2 t_{RCD} = 11, t_{RRD} = 5, t_{CDLR} = 5, t_{WR} = 12
Interconnect	8×8 2D Mesh, 1400MHz, YX Routing, 1 core/node, 8VCs, flit size=32B, Buffers/VC=8, islip VC & switch allocators

Table 3: List of evaluated benchmarks. The patterns listed here correspond to the patterns in Table 1.

Micro-benchmark	Pattern
COMPARE	5
COPY-ALIGNED	3
COPY-STRIDED	3
DENSITY	6
VECADD-ALIGNED	1
VECADD-STRIDED	1
NORM	7

Workload	Pattern	Suite	Dyn.Inst.	Mem.Req.
BFS	2,6,7	CUDA	18%	47%
FDTD	1,2	PolyBench	7%	67%
KMN	2,4	Rodinia	23%	67%
MVT	1,2	PolyBench	25%	69%
RED	1,2	SHOC	16%	75%
SCP	2	CUDA	12%	67%
SRAD	1,2,3,7	Rodinia	3%	75%
STREAM	2,8	Rodinia	33%	67%
TRIAD	9	SHOC	18%	60%

cache and DRAM power using GPUWattch [37]. Based on the injection rate obtained from the simulations, we configure DSENT [59] to find the average power of the interconnect. We run the applications until completion or 1 billion instructions, whichever comes first. The applications and their inputs are large enough to fill the workgroup slots.

Benchmarks: We simulate multiple microbenchmarks that are commonly present in many applications such as scientific computing, machine learning, linear algebra, big data, etc. We also analyze 9 GPGPU workloads from SHOC [13], PolyBench [23], CUDA SDK [47] and Rodinia [11] benchmark suites. Table 3 lists the evaluated microbenchmarks and workloads. The microbenchmarks include: `COMPARE`, which performs point-wise comparison of two strings to count the number of different elements; `COPY-X`[2], which copies one array into another array; `DENSITY`, which counts the number of 0's in an array; `VECADD-X`, which sums two 1-D arrays and stores the result in a third array; and `NORM`, which normalizes a 1-D array with a given value. To highlight the significance of the instructions that we optimize for, we show the fraction of dynamic instructions and total memory requests (Table 3) that can be offloaded. On an average, 17% of dynamic instructions and 66% of memory requests are tagged as offloadable.

The proposed techniques in this work require global memory accesses and are ineffective towards applications that are optimized using GPU features such as shared memory, constant caches, etc. This does not necessarily mean that the scope of the proposed techniques is reduced. Rather, by rewriting applications to make use of computation offloading, it is possible to achieve better energy efficiency/performance. Hence, as a proof of concept, we modified the `RED` application that relies on shared memory to compute partial sums to make use of global memory. We further develop a hybrid version of the workload that combines the global memory and shared memory approaches to find a sweet spot in energy efficiency/performance (Sec. 6.2). Similarly, applications using fused-multiply-add (fma), which require 3 different loads, were broken down into multiply and add instructions, and only the multiply instruction with 2 loads is offloaded and the result is sent back to the core for accumulation with the reused third load. However, our baseline execution results use all the applications' unmodified version.

Hardware Overheads: We implement a 48-entry OQSR in OQ and a 96-entry SQSR in SQ, which was empirically decided as discussed in Sec. 6.2. The OQ also consists of OQMU logic and 30 bytes of storage that is needed for *ComputePacket* generation, but the largest component is the OQSR. OQSR needs roughly 330 bytes of storage. Considering all the register/buffer overheads in SQ, it requires approximately 2.4kB of storage. Current generation GPUs have L1 caches as large as 128kB in their cores [48], indicating that the area overheads are negligible. We use CACTI [44] to model the additional structures and integrate their leakage and dynamic power requirements in our experimental results. For the additional ALUs in the LLCs, we assume it to be similar to the power and area of an SP unit in a core, which is modeled using GPUWattch [37]. To find the EMN at runtime, a lookup table of 64 entries (8×8 combinations) is stored at each core that uses MC ids to find the EMN. All the EMNs for each combination of core and MCs are computed statically (using Algo. 1) as the routing paths are static. The additional ALUs, EMN lookup tables, and all the OQs and SQs require less than 1% area of a high-performance GPU.

6 EXPERIMENTAL RESULTS

To evaluate the benefits of the proposed schemes, we measure the GPU performance (instructions-per-cycle (IPC)), energy efficiency (performance/watt), normalized average memory latency and reduction in on-chip data movement (weighted hop count). The weighted hop count is the sum of all link traversals by flits, which indicates the total amount of on-chip data movement. The average memory latency is the round-trip latency of all the memory requests (L1 misses) in an application. It is made up of the service delay and NoC delay. Service delay is the time the LLC/DRAM takes to access the memory and service the request. The NoC delay consists of the traversal delay, the queuing time at the injection/ejection buffers of the cores/LLCs. All results are normalized to the execution of unmodified workloads on the baseline GPU without computation offloading.

6.1 Effects of Proposed Mechanisms

Effects of LLC-Compute: Fig. 11 shows the performance, performance/watt and the weighted hop count benefits of our LLC-Compute mechanism for seven microbenchmarks and

[2]X is either ALIGNED or STRIDED, indicating whether the memory addresses belong to same or different LLC nodes, respectively.

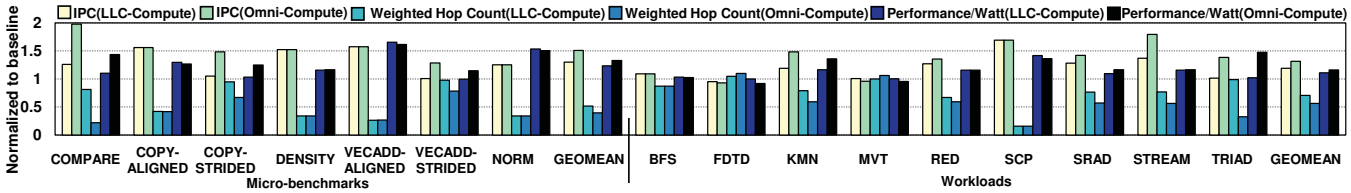

Figure 11: Impact of proposed mechanisms on performance, power efficiency and weighted hop count.

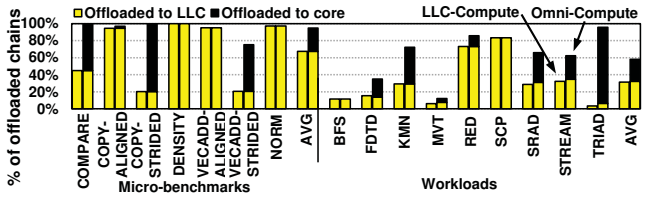

Figure 12: Percentage of offloaded chains.

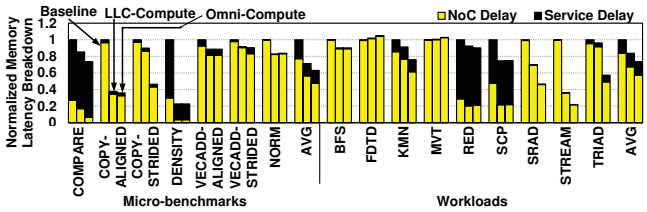

Figure 13: Percentage reduction and breakdown of average memory latency.

nine workloads. LLC-Compute increases the performance and performance/watt for the workloads by 19% and 11%, respectively. It also reduces the weighted hop counts for the workloads by 29%. For the microbenchmarks, it achieves performance and performance/watt improvement of 30% and 23%, respectively. Fig. 12 shows the percentage of offloadable chains that were offloaded by the proposed mechanisms. As mentioned in Sec. 4.5, not all offload chains can be offloaded due to their data placement, cache behavior and/or due to the lack of free slots in OQ/SQ. For applications such as `COPY-ALIGNED`, `DENSITY`, `VECADD-ALIGNED`, `NORM`, `RED` and `SCP`, LLC-Compute is able to improve performance immensely. This is due to the high amounts of offloading as seen in Fig. 12. Applications such as `COMPARE`, `STREAM`, `KMN` and `SRAD` achieve modest gains due to the relatively less amount of offloading. The performance gains achieved by offloading can be correlated with the reduction in average memory latency of the packets as shown in Fig. 13. Note that, Fig. 13 is the detailed version of Fig. 2(b). On an average, for the workloads and microbenchmarks, the average memory latency reduces by 16% and 29%, respectively.

For applications like `COPY-STRIDED`, `BFS`, `FDTD`, `MVT`, `TRIAD` and `VECADD-STRIDED`, we see that LLC-Compute is not able to improve performance. Rather, in the case of `FDTD`, the performance is slightly reduced and the weighted hop counts slightly increased. `FDTD` and `MVT` do not show improvements because of good L1 locality, which leads to less offloaded chains as seen in Fig. 12. The small amount of offloaded chains also causes slight increase in L1 misses, thereby counteracting any benefits in case of `MVT`, but slightly reducing performance and increasing the data movement for `FDTD`. For `COPY-STRIDED`, `VECADD-STRIDED` and `TRIAD`, the reason behind the lack of benefits is due to the lack of offload chains that can be computed at the LLCs. In `BFS`, there are many offload chains, but due to wavefront and memory divergence, the number of in-flight offload chains (each wavefront generates multiple offload chains) is much larger than what OQ and SQ can service.

Effects of Omni-Compute: Fig. 11 also shows the simulation results for our Omni-Compute mechanism. With Omni-Compute, performance and performance/watt for the workloads increase on an average by 31% and 16%, respectively. For the microbenchmarks, it improves the performance and performance/watt by 51% and 33%, respectively. It also reduces the weighted hop counts by 44% for the workloads and by 61% for the microbenchmarks. Fig. 12 shows the percentage of offloadable chains that were offloaded by the proposed mechanism. As shown in Fig. 13, on average, the average memory latency for the workloads and microbenchmarks reduces by 27% and 37%, respectively.

As Omni-Compute builds on top of LLC-Compute, it allows for computation offloading at other GPU cores as well. We see that applications such as `COMPARE`, `COPY-STRIDED`, `VECADD-STRIDED`, `KMN`, `SRAD`, `STREAM` and `TRIAD` improve greatly when compared to LLC-Compute. This is because of the additional opportunities for computation offloading to other cores that is available to Omni-Compute. Applications such as `FDTD` and `MVT` suffer even more than LLC-Compute due to a reduction in L1 locality due to the increase in computation offloading. Note that the cache behavior is dynamic and by offloading computations, we do not get the load requests back to the core, thereby, changing the access pattern. Applications such as `COPY-ALIGNED`, `DENSITY`, `VECADD-ALIGNED`, `NORM`, `BFS`, `SCP` and `RED` do not improve much when compared to LLC-Compute because of their respective data placement. Most of the offload chains are already offloaded to LLCs, making LLC-Compute good enough. Fig. 14 shows the percentage of execution time when either the core or the SQ is in contention for the ALU. In applications such as `VECADD-STRIDED`, `FDTD`, `KMN` and `SRAD`, the contention for the ALU is relatively high compared to other applications. This is due to the fact that not all offload chains are offloaded and are left for the core to execute. Also, in `SRAD` and `KMN`, there are many compute instructions apart

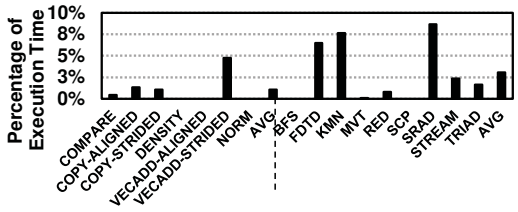

Figure 14: Percentage of execution time when either the core or the SQ contend for ALU.

from offload chains. This causes the SQ to contend for ALU while the core is using it.

6.2 Sensitivity Studies

Interconnect Topology: We evaluated Omni-Compute with multiple interconnect topologies: butterfly (4-ary, 3-fly), crossbar (56×8), and mesh (8×8). We also evaluated them with double their bandwidth.

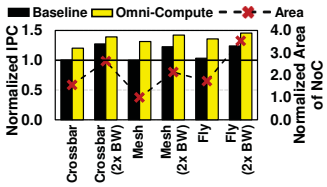

Figure 15: Impact of interconnect topology on performance and area.

Fig. 15 shows the impact of using Omni-Compute towards average performance across all the workloads for each of the topologies. On an average, the performance of crossbar and mesh remain similar while the performance improves by only 3% on butterfly topology. This is due to the large queuing delay at the injection ports at the MCs as observed in Fig. 2(b). Even with decreased hop count of butterfly and crossbar, they do not affect the overall latency of the requests by much. With Omni-Compute, the performance of crossbar and butterfly improves by 20% and 36%, respectively. The benefits in crossbar are not as much as butterfly and mesh. This is because, in crossbar, computations can only be offloaded to LLC nodes, while in butterfly, all the routers and LLC nodes are capable of computing. Furthermore, butterfly is able to achieve a higher performance compared to mesh as it is able to offload chains that were not offloadable in baseline (mesh) due to its dimensional routing policy. Note that these improvements are due to the reduction in NoC delay similar to mesh in Fig. 13. Furthermore, we also doubled the bandwidth of mesh, crossbar and butterfly and found that the performance improves by 27%, 23% and 24%, respectively. With Omni-Compute, it further improves to 39%, 42% and 46%, respectively. This highlights the fact that the on-chip bandwidth is a bottleneck in GPUs and doubling the bandwidth is still not sufficient to eliminate all the congestion delays as we still achieve (albeit relatively lower) performance improvements with Omni-Compute. Fig. 15 also shows the normalized area of using these topologies.

LLC Partition Placement: To analyze the impact of LLC partition placement on Omni-Compute, we study a different LLC placement [31] as shown in Fig. 16(a). Fig. 16(b)

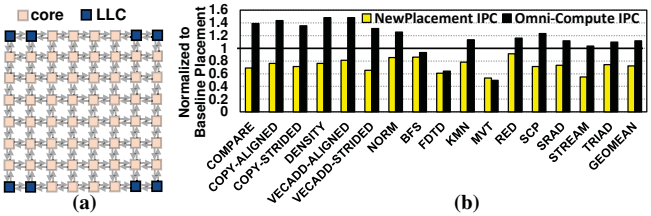

Figure 16: Impact of LLC placement. (a) LLC placement, (b) Performance of new LLC placement.

shows the performance of Omni-Compute with the new LLC placement. This LLC placement is easier for physical layout implementation, but due to dimensional routing, it suffers from higher link utilization on the edge links as seen from the performance degradation when compared to our baseline GPU (Sec. 2). On an average, the new placement scheme leads to a performance degradation of 28% compared to the baseline GPU. With Omni-Compute, the overall performance improves by 56% compared to the no-offload execution. Note that, the performance gains achieved by Omni-Compute for this placement are relatively higher than the one achieved for the baseline GPU. This is due to the fact that more computation offloading can be done in this placement due to the proximity of the LLCs (two LLCs are close to each other) allowing more offload chains to find a suitable EMN.

Shared Memory Optimizations: Applications such as `RED` heavily make use of shared memory in GPUs. This limits the scope of our computation offloading. To this end, we modified the source code of `RED` to make use of global memory rather than shared memory. We also made

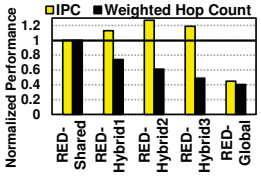

Figure 17: Impact of shared memory optimization.

multiple different variations that use a hybrid approach consisting of global and shared memory. Initial stages of reduction are done using global memory, while the later stages are done using shared memory. Fig. 17 shows the performance and weighted hop count for five different versions of `RED` using LLC-Compute. The first is the unmodified version that uses only shared memory, while the `RED-Global` performs using global memory. Similarly, three different hybrid approaches (`RED-HybridN`) were prepared where the first N level of reduction happen in global memory and then the following levels use shared memory. `RED-Hybrid2` achieves the best performance, and we use this variant for our experimental analysis in Sec. 6.1.

OQ and SQ Size: To determine the size of OQ and SQ, we performed a sweep of multiple (OQ entries, SQ entries) sizes from (12, 24) to (128, 256) to find a feasible design point. We keep SQ size larger than OQ as each SQ will handle requests from multiple cores whereas each OQ is only used by its core. The performance gains of Omni-Compute plateau at 34% for (64,128) and onwards. This is because most of the offloaded

chains can be accommodated and are not discarded due to OQ/SQ being full. We choose the design point of (48,96) due to the resident hardware wavefront limit of 48 for a core. Specifically, without wavefront/memory divergence, there can be a maximum of 48 offload chains from a given core. The SQ size of 96 was chosen empirically.

Dedicated ALU Units: As mentioned in Sec. 4.3, in Omni-Compute, the ALU is shared between the core and the SQ. We also studied the effects of adding dedicated ALUs of varying SIMD-width towards the average performance improvements (only for the 9 workloads) achieved by Omni-Compute. We observe that, for a 4-wide, 8-wide, and 16-wide ALU, average performance degrades by 12%, 5% and 2%, respectively, over the shared ALU scenario (the LLCs have dedicated 32-wide ALUs). With a 32-wide ALU, the performance improves only by 3% compared to shared ALU.

7 RELATED WORK

GPU Optimizations and Computation Offloading: There have been many studies in the past on optimizing the GPU architecture [5, 19, 20, 29, 30, 32, 40, 55, 60, 62, 67] and on computation offloading and scheduling [3, 4, 24, 25, 27, 45, 53, 54, 64, 68, 69]. Meng *et al.* [40] developed a dynamic warp/wavefront subdivision scheme where the wavefronts are split into smaller units and scheduled at a finer granularity to reduce stalls. Zhang *et al.* [67] proposed multiple heuristics for removing the warp and memory divergence using data reordering and job swapping. These proposals reduce the divergence in an application and can potentially increase opportunities for offloading, therefore, complementing our proposed NDC schemes. Moreover, many prior works [7, 16, 17, 28] have identified on-chip bandwidth in GPUs to be a bottleneck as well.

Near Data Computing (NDC) Architectures: The idea of moving computation closer to data is not new, since it has been studied in different contexts including the memory system, known as PIM [10, 21, 22, 36, 49]. While the PIM concept can be traced back to early 1970s [58], due to technological limitations, it could not be fully realized. Recent advances in 3D stacking technology have rejuvenated the interest in PIM [2, 8, 18, 25, 34, 38, 39, 45, 50, 66]. Hsieh *et al.* [27] proposed programmer transparent schemes for offloading code segments to PIM cores and co-locating code and data together in a multi-PIM scenario. Tang *et al.* [61] proposed a software approach, which partitions loop statements into sub-statements to reduce data movement in a CMP. Our work is different in two aspects. First, we target a GPU architecture, whose memory access pattern is more complicated due to massive number of parallel threads. Second, their approach requires synchronizations to ensure correctness. Such synchronization is unsafe and very costly in GPUs. Hashemi *et al.* [26] developed a dynamic scheme that migrates computation to memory controllers to reduce the cache miss latency in CMP. While their approach focuses on dependent cache misses to the same memory controller, our approach is more generic and we offload computations to potentially any location including LLCs and other cores. Any off-chip NDC techniques are complementary to our proposal. Compared to prior efforts, we are the first to explore the notion of earliest-meet node in GPUs.

NoC Optimization: Prior works such as [42, 63, 70] have proposed active networking, wherein routers have sufficient intelligence to perform simple operations on packets as they flow through them. Network packet inference at the routers has been exploited by prior works such as [41, 51, 52] for better congestion management. Kim *et al.* [35] developed a packet coalescing mechanism for GPUs, that reduces the congestion at the MCs, improves performance and reduces data movement. Kim *et al.* [28] provide VC monopolizing and partitioning support for better bandwidth efficiency in GPUs. Bakhoda *et al.* [7] proposed a "checkerboard" mesh for throughput architectures. We use a variation of their proposed LLC placement for our baseline GPU. While such optimizations can help reduce network latency, the network eventually becomes a bottleneck with large problems and datasets. Our proposal can work hand in hand with these techniques for added benefit.

8 CONCLUSION

In this paper, we present two complementary computation offloading techniques for minimizing on-chip data movement in GPU architectures, and hence, improve performance and energy efficiency. The first technique enables computational offloading to the LLCs, while the second technique complements the first technique by adding offloading capability to any node in the 2D mesh interconnect. We identify several small and basic instruction chains in GPU applications that can be offloaded to any one of the locations. The required compiler support, hardware modification to the cores and LLC are presented to facilitate the NDC mechanisms. Simulation results show that our proposed mechanisms are quite effective for reducing on-chip data movement to improve performance and energy efficiency in modern GPUs.

ACKNOWLEDGMENTS

This research is supported in part by NSF grants #1763681, #1629915, #1629129, #1439021, #1317560, #1657336 and #1750667, and a DARPA/SRC JUMP grant. AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. OpenCL is a trademark of Apple Inc. used by permission by Khronos Group, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

REFERENCES

[1] Vignesh Adhinarayanan et al. 2016. Measuring and Modeling On-Chip Interconnect Power on Real Hardware. In *IISWC*.

[2] Junwhan Ahn et al. 2015. PIM-enabled Instructions: A Low-overhead, Locality-aware Processing-in-memory Architecture. In *ISCA*.
[3] Ashwin Mandayam Aji et al. 2015. Automatic Command Queue Scheduling for Task-Parallel Workloads in OpenCL. In *CLUSTER*.
[4] Ashwin M. Aji et al. 2016. MultiCL: Enabling automatic scheduling for task-parallel workloads in OpenCL. *Parallel Comput.* 58 (2016).
[5] Jayvant Anantpur and R. Govindarajan. 2017. Taming Warp Divergence. In *CGO*.
[6] Ali Bakhoda et al. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *ISPASS*.
[7] Ali Bakhoda et al. 2010. Throughput-Effective On-Chip Networks for Manycore Accelerators. In *MICRO*.
[8] Amirali Boroumand et al. 2018. Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks. In *ASPLOS*.
[9] Steve Carr et al. 1994. Compiler Optimizations for Improving Data Locality. In *ASPLOS*.
[10] J. Carter et al. 1999. Impulse: building a smarter memory controller. In *HPCA*.
[11] Shuai Che et al. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *IISWC*.
[12] William J. Dally and Brian Towles. 2001. Route Packets, Not Wires: On-chip Inteconnection Networks. In *DAC*.
[13] Anthony Danalis et al. 2010. The Scalable Heterogeneous Computing (SHOC) Benchmark Suite. In *GPGPU*.
[14] Advanced Micro Devices. 2017. Radeon Vega Architecture.
[15] Jeff Draper et al. 2002. The Architecture of the DIVA Processing-in-memory Chip. In *ICS*.
[16] S. Dublish et al. 2016. Characterizing Memory Bottlenecks in GPGPU Workloads. In *IISWC*.
[17] S. Dublish et al. 2017. Evaluating and mitigating bandwidth bottlenecks across the memory hierarchy in GPUs. In *ISPASS*.
[18] Amin Farmahini-Farahani et al. 2015. DRAMA: An Architecture for Accelerated Processing Near Memory. *IEEE CAL* 14, 1 (2015).
[19] Wilson W.L. Fung et al. 2007. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *MICRO*.
[20] Wilson W. L. Fung and Tor M. Aamodt. 2011. Thread Block Compaction for Efficient SIMT Control Flow. In *HPCA*.
[21] Maya Gokhale et al. 1995. Processing in Memory: the Terasys Massively Parallel PIM Array. *Computer* 28, 4 (1995).
[22] A. Gottlieb et al. 1983. The NYU Ultracomputer Designing an MIMD Shared Memory Parallel Computer. *IEEE Trans. Comput.* (1983).
[23] Scott Grauer-Gray et al. 2012. Auto-tuning a High-level Language Targeted to GPU Codes. In *2012 Innovative Parallel Computing (InPar)*.
[24] Jashwant Gunasekaran et al. 2019. Spock: Exploiting Serverless Functions for SLO and Cost aware Resource Procurement in Public Cloud. In *CLOUD*.
[25] Ramyad Hadidi et al. 2017. CAIRO: A Compiler-Assisted Technique for Enabling Instruction-Level Offloading of Processing-In-Memory. *TACO* 14, 4.
[26] Milad Hashemi et al. 2016. Accelerating Dependent Cache Misses with an Enhanced Memory Controller. In *ISCA*.
[27] Kevin Hsieh et al. 2016. Transparent Offloading and Mapping (TOM): Enabling Programmer-Transparent Near-Data Processing in GPU Systems. In *ISCA*.
[28] Hyunjun Jang et al. 2015. Bandwidth-efficient On-chip Interconnect Designs for GPGPUs. In *DAC*.
[29] Adwait Jog et al. 2015. Anatomy of GPU Memory System for Multi-Application Execution. In *MEMSYS*.
[30] Adwait Jog et al. 2016. Exploiting Core Criticality for Enhanced GPU Performance. In *SIGMETRICS*.
[31] Onur Kayiran et al. 2014. Managing GPU Concurrency in Heterogeneous Architectures. In *MICRO*.
[32] Onur Kayiran et al. 2016. μC-States: Fine-grained GPU Datapath Power Management. In *PACT*.
[33] Stephen W. Keckler et al. 2011. GPUs and the Future of Parallel Computing. *IEEE Micro*.
[34] Hyojong Kim et al. 2015. Understanding Energy Aspects of Processing-near-Memory for HPC Workloads. In *MEMSYS*.
[35] Kyung Hoon Kim et al. 2017. Packet Coalescing Exploiting Data Redundancy in GPGPU Architectures. In *ICS*.
[36] Peter M. Kogge. 1994. EXECUBE-A New Architecture for Scaleable MPPs. In *ICPP*.
[37] Jingwen Leng et al. 2013. GPUWattch: Enabling Energy Optimizations in GPGPUs. In *ISCA*.
[38] Gabriel H. Loh. 2008. 3D-Stacked Memory Architectures for Multi-core Processors. In *ISCA*.
[39] Gabriel H. Loh et al. 2013. A Processing-in-Memory Taxonomy and a Case for Studying Fixed-function PIM. In *WoNDP*.
[40] Jiayuan Meng et al. 2010. Dynamic Warp Subdivision for Integrated Branch and Memory Divergence Tolerance. In *ISCA*.
[41] George Michelogiannakis et al. 2011. Packet Chaining: Efficient Single-cycle Allocation for On-chip Networks. In *MICRO*.
[42] Asit K. Mishra et al. 2011. A Case for Heterogeneous On-chip Interconnects for CMPs. In *ISCA*.
[43] Aaftab Munshi. 2009. The OpenCL Specification. (2009), 1–314.
[44] Naveen Muralimanohar et al. 2009. CACTI 6.0: A tool to model large caches. *HP Laboratories* (2009), 22–31.
[45] Lifeng Nai et al. 2017. GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks. In *HPCA*.
[46] NVIDIA. 2008. Parallel Thread Execution (PTX). (2008).
[47] NVIDIA. 2011. CUDA C/C++ SDK Code Samples.
[48] NVIDIA. 2017. NVIDIA TESLA V100 GPU Architecture.
[49] David Patterson et al. 1997. A Case for Intelligent RAM. *IEEE Micro*.
[50] Ashutosh Pattnaik et al. 2016. Scheduling Techniques for GPU Architectures with Processing-In-Memory Capabilities. In *PACT*.
[51] Mukund Ramakrishna et al. 2016. GCA:Global Congestion Awareness for Load Balance in Networks-on-Chip. *IEEE TPDS* (2016).
[52] Rohit Sunkam Ramanujam and Bill Lin. 2010. Destination-based Adaptive Routing on 2D Mesh Networks. In *ANCS*.
[53] Prasanna Venkatesh Rengasamy et al. 2017. Characterizing diverse handheld apps for customized hardware acceleration. In *IISWC*.
[54] Prasanna Venkatesh Rengasamy et al. 2018. CritICs Critiquing Criticality in Mobile Apps. In *MICRO*.
[55] Timothy G. Rogers et al. 2013. Divergence-Aware Warp Scheduling. In *MICRO*.
[56] Michael J. Schulte et al. 2015. Achieving Exascale Capabilities through Heterogeneous Computing. *IEEE Micro* 35, 4 (2015).
[57] Akbar Sharifi et al. 2012. Addressing End-to-End Memory Access Latency in NoC-Based Multicores. In *MICRO*.
[58] Harold S. Stone. 1970. A Logic-in-Memory Computer. *IEEE Trans. Comput.* C-19, 1 (1970).
[59] Chen Sun et al. 2012. DSENT-a tool connecting emerging photonics with electronics for opto-electronic networks-on-chip modeling. In *NoCS*.
[60] Xulong Tang et al. 2017. Controlled Kernel Launch for Dynamic Parallelism in GPUs. In *HPCA*.
[61] Xulong Tang et al. 2017. Data Movement Aware Computation Partitioning. In *MICRO*.
[62] Xulong Tang et al. 2018. Quantifying Data Locality in Dynamic Parallelism in GPUs. *Proc. ACM Meas. Anal. Comput. Syst.* 2, 3 (Dec. 2018).
[63] David L Tennenhouse et al. 1997. A Survey of Active Network Research. *Communications Magazine, IEEE* 35, 1 (Jan. 1997).
[64] Prashanth Thinakaran et al. 2017. Phoenix: a constraint-aware scheduler for heterogeneous datacenters. In *ICDCS*. IEEE.
[65] Jia Zhan et al. 2016. OSCAR: Orchestrating STT-RAM Cache Traffic for Heterogeneous CPU-GPU Architectures. In *MICRO*.
[66] Dongping Zhang et al. 2014. TOP-PIM: Throughput-oriented Programmable Processing in Memory. In *HPDC*.
[67] Eddy Z. Zhang et al. 2011. On-the-fly Elimination of Dynamic Irregularities for GPU Computing. In *ASPLOS*.
[68] Haibo Zhang et al. 2017. Race-to-sleep + Content Caching + Display Caching: A Recipe for Energy-efficient Video Streaming on Handhelds. In *MICRO*.
[69] Shulin Zhao et al. 2019. Understanding Energy Efficiency in IoT App Executions. In *ICDCS*.
[70] Amir Kavyan Ziabari et al. 2015. Asymmetric NoC Architectures for GPU Systems. In *NOCS*.