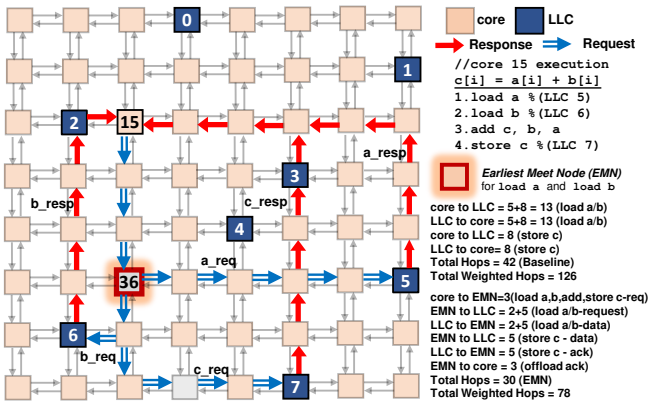

Figure 3: Earliest Meet Node for an instruction sequence (`c[i]= a[i]+b[i]`). For each memory operation, the request and response packets' traversal with YX routing is shown. All memory requests generate from core 15. The two loads and store head to LLC 5, LLC 6 and LLC 7, respectively. For this instruction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

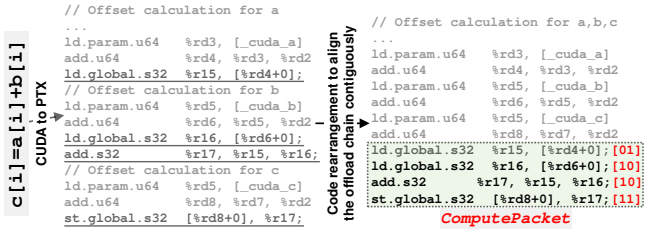

Figure 6: Representative code snippet. The offload chain is tagged and rearranged in the PTX code to align contiguously in memory.

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.

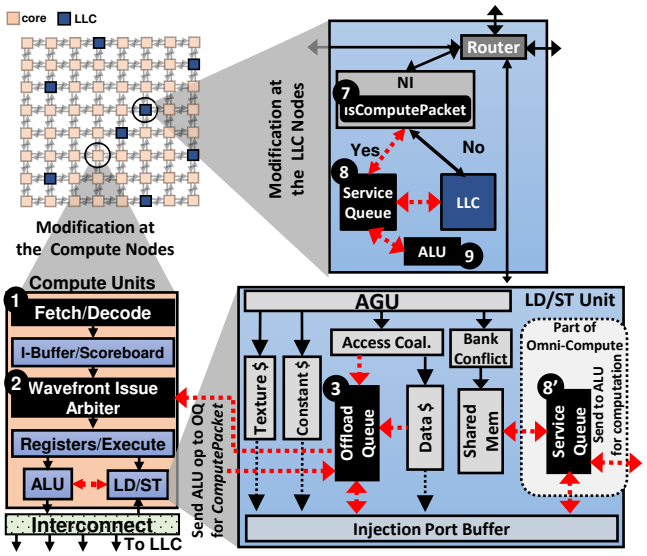

Figure 7: Proposed hardware modification to enable offloading. Additional/modified units are shown in black; The additional unit in Omni-Compute (over LLC-Compute) is the SQ in LD/ST unit 8'.

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

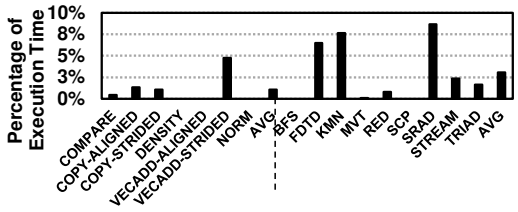

Figure 14: Percentage of execution time when either the core or the SQ contend for ALU.

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.

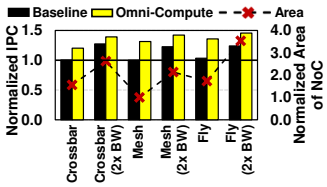

Figure 15: Impact of interconnect topology on performance and area.

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)

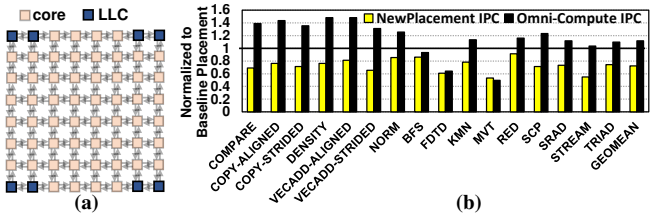

Figure 16: Impact of LLC placement. (a) LLC placement, (b) Performance of new LLC placement.

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

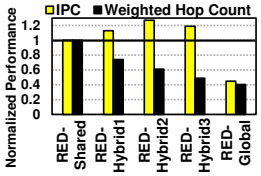

Figure 17: Impact of shared memory optimization.

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

[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*.