Packet Coalescing Exploiting Data Redundancy in GPGPU Architectures

Kyung Hoon Kim, Rahul Boyapati, Jiayi Huang, Yuho Jin†, Ki Hwan Yum, Eun Jung Kim

Department of Computer Science and Engineering, Texas A&M University, †Advanced Micro Devices, Inc
khkim,rahulboyapati,jyhuang,yum,ejkim@cse.tamu.edu,Yuho.Jin@amd.com

ABSTRACT

General Purpose Graphics Processing Units (GPGPUs) are becoming a cost-effective hardware approach for parallel computing. Many executions on the GPGPUs place heavy stress on the memory system, creating network bottlenecks near memory controllers. We observe that data redundancy in communication traffic is commonplace across a wide range of GPGPU applications. To exploit the data redundancy, we propose a packet coalescing mechanism to alleviate the network bottlenecks by directly reducing the traffic volume. The key idea is to coalesce multiple packets into one without increasing the packet size when they carry redundant cache blocks. To ensure that the coalesced packets are delivered to their respective destinations, we adopt multicast routing for the interconnection network of GPGPUs. Our coalescing approach yields 15% IPC improvement (up to 112%) in a large-scale GPGPU with 2D mesh across various GPGPU applications, by reducing average memory access time (AMAT) by 15.5% (up to 65.2%) and obtaining network bandwidth savings by 13% (up to 37%). Also, our coalescing approach achieves 7% IPC improvement in the NVIDIA Fermi architecture with the crossbar.

CCS CONCEPTS

•Computer systems organization →Single instruction, multiple data;

KEYWORDS

GPGPU, Packet Coalescing, Multicast, Inter-core Locality

ACM Reference format:
Kyung Hoon Kim, Rahul Boyapati, Jiayi Huang, Yuho Jin†, Ki Hwan Yum, Eun Jung Kim. 2017. Packet Coalescing Exploiting Data Redundancy in GPGPU Architectures. In *Proceedings of ICS '17, Chicago, IL, USA, June 14-16, 2017,* 10 pages.
DOI: http://dx.doi.org/10.1145/3079079.3079088

1 INTRODUCTION

Modern GPGPUs, equipped with a large number of computation units, provide energy-efficient executions for a wide variety of high throughput data parallel applications. GPGPUs consist of multiple Streaming Multiprocessors (SMs), each comprising multiple compute units, and a set of on-chip Memory Controllers (MCs) connected via scalable Networks-on-Chip (NoCs) [3][11][32]. An enormous amount of parallel thread executions in GPGPUs place heavy stress on the memory systems causing the memory bandwidth to become the critical performance bottleneck [3][11][32]. The memory bottleneck leads to long memory access latencies in GPGPUs, which are hidden by fine-grained thread context switching [2][16][30].

As technology scales, the number of MCs in the GPGPUs does not scale with the SMs due to on-chip pin bandwidth limitation [17]. This exacerbates the memory bottleneck and renders the latency hiding less effective, due to increased AMAT, eventually leading to significant overall system performance degradation. A considerable portion of AMAT is caused by the *MC bottlenecks* where a large amount of reply data from MCs to SMs cannot be injected into the network due to restricted terminal bandwidth at the MC routers even when the data is ready to be sent [3]. The MC bottlenecks are even more aggravated by network hotspots in the NoC near the MCs that cannot transfer a large volume of traffic fast enough due to limited network bandwidth [11]. Therefore, it is critical to explore solutions in the NoC to alleviate the MC bottlenecks.

There have been previous studies on designing NoCs tailored to GPGPUs. Bakhoda et al.[3] proposed to provide additional terminal bandwidth using a multiport router design for the MC nodes. Such a design can alleviate the congestion problem at the MC routers by providing additional injection/ejection capabilities but does not reduce the underlying traffic directly. This design also becomes cost-ineffective as the GPGPUs scale up, thereby aggravating the MC router congestion. Recent work has attempted to investigate the issues of virtual channel (VC) allocation for request/reply traffic, MC placement, routing algorithm and network topology to find the optimal NoC design for GPGPUs [11][32]. However, none of these studies tried to solve the MC bottleneck issue in the standpoint of reducing the traffic volume, which we believe is critical to address the issue.

We propose a packet coalescing mechanism that reduces NoC traffic volume by exploiting data redundancy in the GPGPU communication traffic. The proposed mechanism coalesces multiple packets that exhibit data redundancy into a single packet without increasing the packet size, thereby reducing the number of packets injected into the network. Data redundancy in GPGPU communication stems from data sharing among multiple SMs, called *inter-core locality* [20]. We introduce a packet coalescing unit (*PCU*) in the MCs which captures a group of memory requests with inter-core

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICS '17, Chicago, IL, USA
© 2017 ACM. 978-1-4503-5020-4/17/06...$15.00
DOI: http://dx.doi.org/10.1145/3079079.3079088

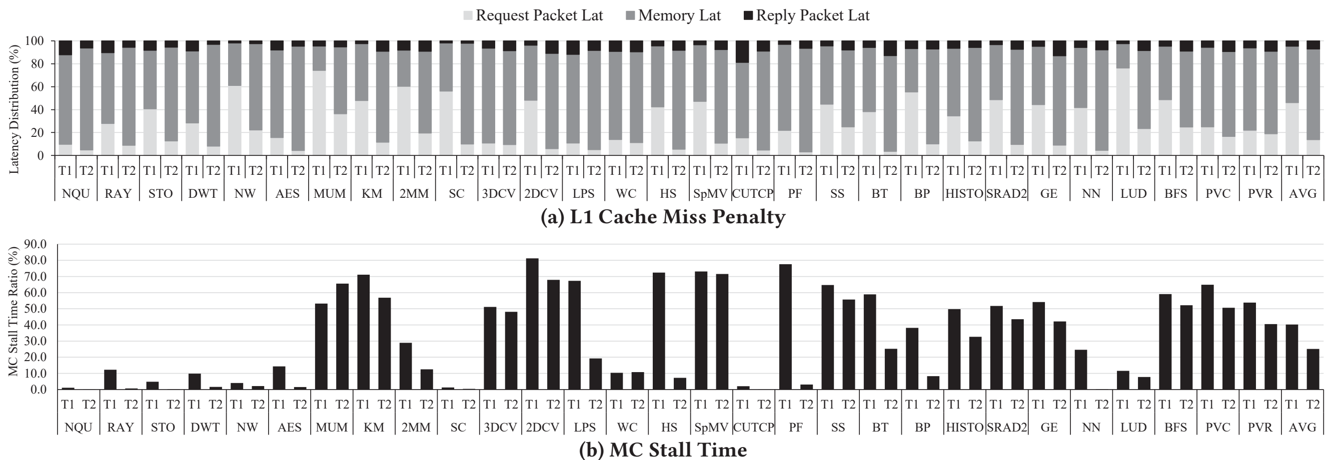


Figure 1: MC Bottlenecks in GPGPUs across 29 Benchmarks (T1: 2DMesh, T2: Crossbar)

locality from multiple SMs, and generates a single read reply packet destined for the requesting SMs. To make sure that the single packet is delivered to all the SMs, we adopt an existing multicast routing for GPGPUs. In this paper, we make the following contributions.

- We propose a new packet coalescing mechanism that alleviates the MC bottlenecks through traffic volume reduction.
- We adopt a multicast routing algorithm that delivers coalesced packets to SMs. To the best of our knowledge, this is the first work showing a good use of multicast in GPGPU applications.
- We analyze the MC bottleneck issue and inter-core locality common across various applications and characterize applications with inter-core locality.
- We comprehensively evaluate the proposed coalescing technique across various applications from GPGPU-SIM [4], Rodinia [6], Mars [9], Polybench [10], and Parboil [28] benchmark suites. Our coalescing approach yields 15% IPC improvement on average in a large-scale GPGPU with 2D mesh by reducing AMAT by 15.5% and obtaining network bandwidth savings by 13%. Also, our coalescing approach achieves 7% IPC improvement in the NVIDIA Fermi architecture with the crossbar.

2 MOTIVATION

In this section, we explain the MC bottleneck issue in a GPGPU with a 2D mesh/crossbar interconnect and motivate our proposed mechanisms based on the observation of widely common data redundancy in GPGPUs.

2.1 MC Bottleneck

A large amount of parallelism in GPGPUs places heavy stress on the limited number of MCs on the chip, especially because L2 cache banks are located only in the MC nodes, and hence every L1 cache miss access is destined for one of the MCs through an interconnection network. The communication patterns in GPGPUs are many-to-few in the request network from many SMs to a few MCs, and reversely few-to-many in the reply network from a few MCs to many SMs [3].

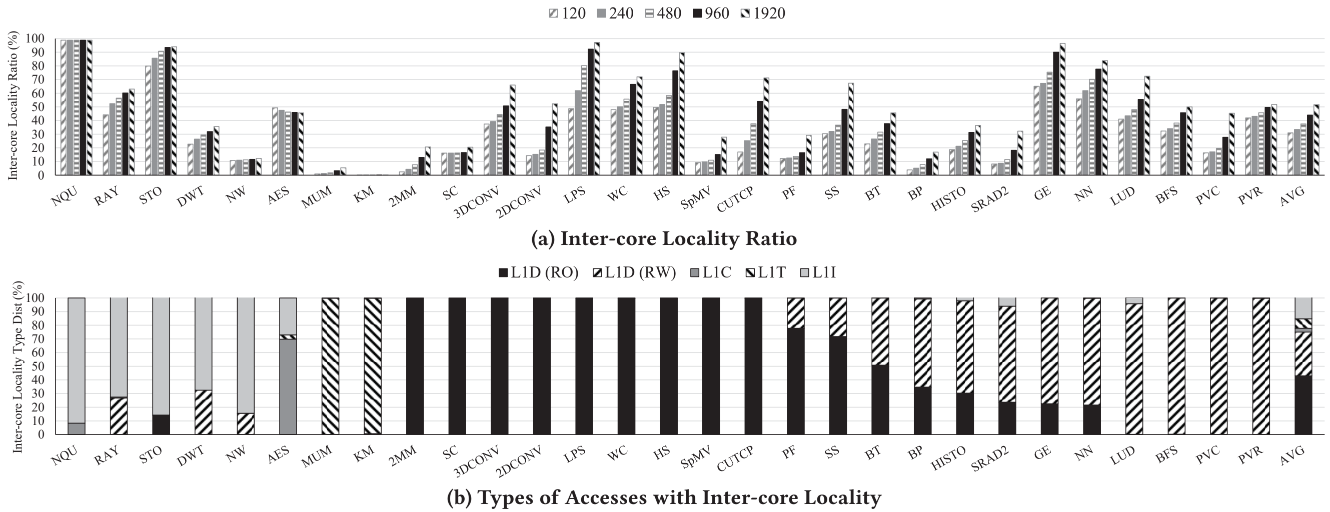
To understand the key reason of the MC bottlenecks, we analyze L1 cache miss penalty of AMAT in two different scales of GPGPUs. We model a large-scale GPGPUs of 56 SMs and 8 MCs with 2D mesh [11], while we do NVIDIA Fermi architecture of 15 SMs and 6 MCs with the crossbar [25]. Through these experiments, we see severe bottlenecks occur in both GPGPU models and the bottlenecks are mainly due to a large volume of traffic highly skewed toward the reply network.

Figure 1a[1] shows the breakdown of L1 cache miss penalty measured for all memory requests across 29 benchmarks. The penalty is divided into three latencies: request packet latency, memory latency, and reply packet latency. The request and reply packet latencies are calculated from the time packet's flits are created to the moment when its tail flit is accepted in the final destination. The memory latency is from the time a request packet is accepted by an MC to when a corresponding reply packet is created.

The MC bottlenecks are presented in the request packet latency that is asymmetrically longer than the reply latency. The average request latency is 10 and 2 times higher than the reply latency in 2D mesh and crossbar, respectively. It is due to the backpressure from the highly congested reply network to the request network. Once reply data is read from memory systems, it stays in MC reply queue placed between MC and its Network Interface (NI) input queue, waiting for being sent to the network. As the reply network gets more congested, the MC NI input queue is full, and thus reply data cannot be sent immediately and keeps waiting in MC reply queue. When the reply queue becomes full, an L2 cache cannot accept a request in MC request queue that stores new memory requests from SMs. It is because the reply queue has no more space to store reply data when the request hits the L2 cache. Then, request packets continue to wait in the request network until the MC request queue has available space, resulting in a long request latency.

To quantify the severity of MC bottlenecks, we measure the ratio of average MC stall time out of the total execution time for each benchmark. We count the stall time when MCs cannot inject packets due to the MC NI input queue being full. The average MC stall time ratio in 2D mesh is 40.4% as shown in Figure 1b. Such

[1]To analyze MC bottlenecks in the interconnect perspective, we present the L1 cache miss penalty rather than SM stall cycles. Due to severe response delay by the bottlenecks, the performance is highly affected by the miss penalty, although GPGPUs are designed for hiding latency.



(a) Inter-core Locality Ratio

(b) Types of Accesses with Inter-core Locality

Figure 2: Data Redundancy across 29 Benchmarks in 2DMesh

frequent MC stall has been also observed by earlier GPGPU NoC design work [3]. Interestingly, MCs in the crossbar incorporated by modern GPGPUs also stall 25.1% of execution time, particularly in memory-intensive applications. Consequently, the MC bottleneck increases AMAT and, in turn, degrades the overall system performance. Therefore, it is crucial to reduce the stalls by fundamentally reducing the number of packets sent to the MC NI input queue.

2.2 Data Redundancy

Data Sharing among SMs. Inter-core locality occurs when multiple SMs send requests to the same cache block within a relatively short period of time. In order to quantify its potential and temporal locality, and identify its sources, we analyze cache block access patterns in each MC across a wide-range of applications.

Once a read request arrives at an MC router, it is sent to L2 cache. Between them, we capture all read requests that are going to access the same cache block. To do this, we maintain a table where a cache block address is associated with the number of accesses in each entry. If a cache block address of a request does not exist in the table, a new entry is allocated, storing the block address and initializing the number of accesses to one. The entry is deallocated after a fixed-length time window. If a block address of a request does exist, the number of accesses increases by one. To capture all requests, the number of entries is assumed to be unlimited. Five time windows such as 120, 240, 480, 960 and 1920 cycles are adopted by taking multiples of the minimum L2 hit latency (i.e. 120 cycles) [25]. An entry has inter-core locality when it records multiple accesses. The inter-core locality ratio is measured by the percentage of the total number of accesses in all entries with inter-core locality out of the number of all read requests. As shown in Figure 2a, 31% of the requests have inter-core locality on average, when the time window is set to 120 cycles. As the time window is increased to 1920 cycles, the inter-core locality ratio increases up to 51.7%.

Figure 2b shows the distribution of access types with inter-core locality at the time window of 960 cycles. Inter-core locality mainly occurs from L1 data (L1D) cache misses by 75% where read-only data takes 43% and read-write data takes 32%. Modern GPGPUs do not support cache coherence protocols among SMs [25], but synchronization method among SMs is often used to avoid data race circumstances on the read-write data. Other sources of inter-core locality are cache misses from read-only caches such as L1 instruction (L1I), constant (L1C) and texture (L1T) cache , which take 16%, 2.8% and 7%, respectively.

Characterization of Applications. The inter-core locality associated with L1D cache misses occurs when an application is written to run many threads accessing shared data structures. We characterize the applications in terms of their computation characteristics on the shared data as follows.

- *Pair-wise Computation.* In MapReduce framework applications, Map stage passes a list of key and value pairs to Reduce stage. Group stage between them sorts the output of Map stage by keys. In the sorting process, threads fetch and compare data elements. When the elements that each SM needs exist in a cache block, inter-core locality occurs. Similarly, SC, CUTCP, HISTO and SpMV have inter-core locality due to pair-wise computation features.
- *Graph Data Computation.* In applications using graph data such as BFS, BT, NN and BP, a data node is explicitly connected with neighboring nodes. In their computation flow, they usually involve checking or obtaining previous data nodes. The inter-core locality occurs when multiple nodes processed by different SMs need data from the same previous node.
- *Stencil Computation.* Applications compute a data point by using neighboring data points. Although SMs are assigned a distinct data tile, the boundary regions, called halo regions [22], around the data tile are redundantly accessed by multiple SMs. HS, PF, SRADV2, 3DCONV, 2DCONV and LPS belong to this type.
- *Computation with row-wise and column-wise dependency.* Applications such as LUD, GE and 2MM access row and column data points associated with a data point, to compute the point. As a large dataset cannot be fit in the shared memory of an SM, such row/column data is necessary for multiple SMs.

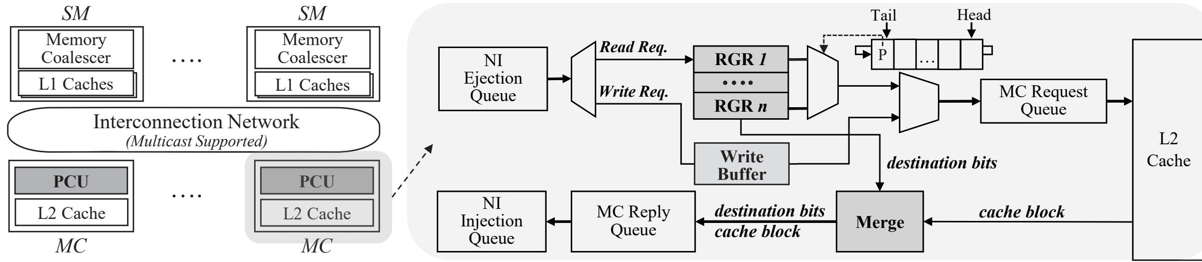


Figure 3: GPGPU Architecture Incorporating Proposed PCUs

3 PACKET COALESCING UNIT

In this section, we explain our packet coalescing units (PCUs) that reduce the number of packets by combining multiple packets into one without increasing the packet size.

3.1 Overview

Figure 3 shows the overall GPGPU architecture integrating PCUs to MCs. In each SM, memory accesses from threads of a warp are combined into fewer accesses by a memory coalescer [25]. When they have L1 cache misses, memory read or write requests packets are sent to MCs through an interconnection network. The MCs respond to them with read reply or write reply packets, respectively. The read reply packets, which are a key factor of MC bottleneck, are coalesced by a PCU, before being injected into the network. A PCU coalesces packets by up to the number of all SMs. To deliver the packet to final destinations, the interconnection network requires multicast capability that we will discuss in Section 4. In the following, we explain the details of a PCU.

3.2 Coalescing Mechanism

The proposed coalescing mechanism has two key features. First, multiple packets are coalesced into a single one without packet size increase. To coalesce packets one may think of appending packets back-to-back, but it does not attain our goal of reducing traffic volume. Instead, to exploit inter-core locality described in Section 2, we attempt to merge only read reply packets carrying the same cache block into a single one. The packet header and payload store multiple destinations and the cache block, respectively. As the unused space of a packet accommodates the destinations, the packet has the same size as a normal (i.e. uncoalesced) packet.

Second, coalescing is performed with low latency overhead. One way of coalescing is to keep track of cache blocks in the MC reply queue and merge ones with the same contents, while they are in the queue. Thus, the longer the packet stays in the queue, the more packets can be coalesced. However, adding extra waiting time to earlier packets is not desirable since it increases end-to-end latency. We determine a group of reply packets to be coalesced when their corresponding requests arrive at an MC. This process is called *Request Grouping*. Once a cache block for a group of requests returns from memory systems, we provide information such as the cache block and SM ids with NI to generate a packet destined for the SMs. This process is called *Reply Merging*. The request grouping leads to low latency overhead for identifying requests accessing the same cache block, while the reply merging does no overhead.

Suppose that a request to a cache block arrives at an MC from SM 1 and there were no requests to the block so far. The request grouping allows the request to access memory systems and records SM 1 as a requesting SM. Until the accessed block returns, if subsequent requests to the same block are sent from SM 2 and 3 to the MC, the request grouping records SM 2 and 3, and it does not allow them to access memory systems. When the block returns, the reply merging sends all recorded SM ids (i.e. 1, 2 and 3) and the cache block to NI that creates a packet destined for the SMs. Now we describe the details of request grouping and reply merging

Request Grouping. The request grouping is performed in a static time window that depends on memory access time. When a request forms a new request group, it is sent to memory systems. The request group continues to capture all subsequent requests that access the same cache block as the first one does. The number of accesses to the same cache block is bound by the number of SMs since the redundant accesses from each SM are blocked by MSHRs of L1 cache. This grouping is terminated when a cache block for the first request returns. This operation is similar to miss status holding register (MSHR) mechanism of an L2 cache. However, we introduce the request grouping mechanism before L2 cache separately to capture more requests with inter-core locality. GPGPU has a long L2 cache access time (120 cycles [25]) due to the delay of raster operation (ROP) unit coupled with an L2 cache. The request grouping can use it as the minimum time window when a request hits L2 cache. Upon miss in an L2 cache, the request grouping can make use of DRAM access time in addition to the L2 cache access time, which is the maximum time window. The accesses that do not access the main memory are frequently captured by request grouping with the minimum time window due to their temporal locality shown in Figure 2a.

To implement the request grouping, we introduce a Request Grouping Register (*RGR*) which groups requests with inter-core locality by storing a cache block address and their requesting SM ids. RGR has 1-bit *valid* field, 41-bit *block address* field, and 64-bit *destination bits* field where each bit position indicates the location of a requesting SM. The RGR that stores requests with inter-core locality has multiple bits of *destinations bits* field set to ones.

We design the request grouping in two stages to perform the grouping while the MC request queue is full. To send read requests to L2 cache in their arriving order, we maintain the PCU pointer ring-buffer where the locations of RGRs are stored according to their allocation order. The PCU head/tail pointers are used to read RGRs in that order. The request grouping mechanism operates in the following manner.

- **Stage 1.** When there is an available RGR, a read request is accepted by PCU. For the read request, all valid RGRs are sequentially accessed to find a match on the block address. If there is a hit in a valid RGR, the requesting SM id is stored in the destination bits field and the request is dropped (not sent to MC request queue). If an RGR miss occurs, an empty RGR (the *valid* field is zero) is located. The requesting SM id is stored in the *destination bits* field of the RGR. The accessing address is also stored in the *block address* field. The PCU head pointer is set to next available space in the PCU pointer buffer. The RGR location is stored at the space.
- **Stage 2.** Next read request is selected based on an RGR pointed by the PCU tail pointer. When the PCU head and tail pointers are the same, no read request is available. If MC reply queue has available space, a request selector chooses either a read request from the selected RGR or a write request in a write buffer in a round-robin way. When the read request is selected, the block address of the RGR is sent to MC request queue and the PCU tail pointer is set to next valid RGR.

Reply Merging. We introduce Merge unit that combines multiple replies in a single reply. Merge unit stands between L2 cache and MC reply queue. When a cache block returns from L2 cache, the Merge unit obtains the destination bits by accessing the corresponding RGR. Both the cache block and the destination bits are sent to the MC reply queue. At this point, the RGR is reset by clearing its *valid* field for new RGR allocation. The cache block is packetized by NI as a reply packet for SMs encoded in the *destination bits* field. A flit that stores a packet header accommodates the destination bits in its unused space (e.g. 8B in 2D mesh). If the *destination bits* field encodes a single destination, a reply packet is sent to the destination as a unicast packet. Otherwise, the reply packet is a multicast packet sent to all requesting SMs, which we will discuss details in the next section.

4 MULTICAST SUPPORT IN NOC

In this section, we detail multicast support for both large-scale and NVIDIA Fermi-style GPGPU architectures.

4.1 Overview

First, we present an overview of the NoC architectural details for both large-scale and NVIDIA Fermi-style GPGPUs. NVIDIA Fermi architecture uses a global crossbar interconnection network with destination tag routing [25]. For large-scale GPGPU architectures, we propose to use a 2D mesh interconnection topology among various NoC topologies as in [5][11] because the global crossbar is not a practical solution due to the complexity of layout and huge power consumption [32]. The efficiency of the proposed packet coalescing mechanisms, which exploit the application behavior of inter-core locality, is independent of the underlying interconnection network topology. For the global crossbar, the request and reply networks are separated by two different crossbar switches. For the 2D mesh, a single network is used for both request and reply communication. To avoid protocol deadlocks, the network is divided into two virtual subnetworks for the respective communication, where VCs are evenly dedicated to each subnetwork [4].

4.2 Multicast in Crossbar

To support multicasting in the crossbar, flit replication capability is primarily needed. To enable replication with high throughput, we manifest the matrix-crossbar in the Fermi architecture into a mux-based crossbar. In earlier multicast studies like VCTM [12], replication is performed by reading the same flit out of a VC and sending it to each output port one-by-one upon successful allocation. This has an advantage of a simple crossbar design but incurs serialization delay to multicast flits. Hence, we adopt a mux-based crossbar used by RPM [31] that supports high throughput at the cost of higher energy consumption.

4.3 Multicast in 2D Mesh

For multicast support in the 2D mesh topology, we adopt a multicast router supporting tree-based routing, similar to VCTM [12], RPM [31], BAM [21] and Whirl [19]. The routing algorithms in these routers have been optimized for the traffic patterns in Chip MultiProcessors where core-to-core communications are frequent. Jang et al [11] has shown that a Dimension Order Routing (DOR) is simple but effective in GPGPU due to the traffic patterns occurring between SMs and MCs only. Therefore, the multicast router in this paper implements DOR.

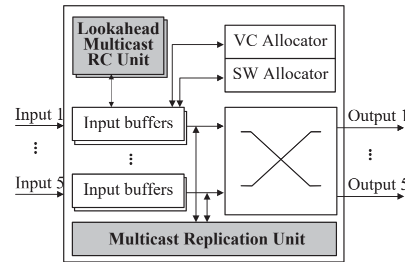


Figure 4: Multicast Router Architecture

We use a 3-stage lookahead router as the baseline router. A traditional NoC router has four stages: Routing Computation (RC), VC Allocation (VA), Switch Allocation (SA) and Switch Traversal (ST). To reduce the pipeline depth, the 3-stage lookahead router performs the routing computation for next hop router in the VA stage [8].

To support multicast routing in the baseline router, we incorporate a replication unit and lookahead multicast RC units as depicted in the shaded units in Figure 4. The replication unit copies a multicast packet at a replication point to different directions to make sure that the packet arrives at all final destinations. The multicast RC units decide the output port list to which replicated packets are directed, based on DOR. For each replica, it also splits a destination list of an original packet into a subset being routed via the same output port. Then, VC allocator uses the output port list to get an available VC from the downstream routers for replica packets. As a replica gets a free VC, it goes to the SA stage and a packet is replicated to an output port at the ST stage, storing the destination list for the replica in replica's header [31].

We use multiple RC units to support lookahead routing decision for replicas.When a packet is replicated to multiple directions, we need to make sure that the lookahead routing decision is performed for each replicated packet, which causes additional complexity. For all replicas, each input port is required to have lookahead RC

capability for all immediate neighboring routers. Since the multiple RC units work in parallel, they can be overlapped with the VA stage without increasing the critical path.

As a replication scheme, we choose an asynchronous replication scheme [31] where flits targeted for different destinations are forwarded independently. Then VA, SA and ST are done for each flit individually. An input VC keeps storing a flit until it copies the flit to all target output ports [31][21]. Replicating a packet to multiple output ports may have conflicts with other normal packets already in the router. Replicated flits are handled like normal flits for the VA/SA stages without giving priority on the replicated flits. To enable replication with high throughput, we choose the mux-based crossbar inside the multicast router discussed in Section 4.2.

5 EVALUATION

5.1 Methodology

To evaluate the proposed packet coalescing we integrate PCUs into a cycle-accurate GPGPU simulator, GPGPU-Sim 3.2.2 [4]. We modify Booksim [14], the NoC simulation component of GPGPU-Sim, to simulate multicast for crossbar and 2D mesh. To see the impact of routing to coalescing performance, we use two routing algorithms for 2D mesh, XY-XY and XY-YX, where both uses XY routing in the request network, and use XY and YX routing, respectively for the reply network. The number of RGRs that affect coalescing performance is set to 128 for each PCU. We use CACTI model 6.5 [23] to measure latency and energy overhead of RGR. Table 1 shows the detailed system parameters we use to model the baseline GPGPU architecture.

System Parameters	Details
Shader Core	56 / 15 Cores, 1.4Ghz
Memory Model	8 / 6 MCs, 924 MHz
Warp Scheduler	Greedy-Then-Oldest (GTO)
L1I, L1T, L1C Cache	2KB, 12KB, 8KB
L1D Cache, Shared Memory	16KB, 48KB
L2 Cache	64KB
Min L2, DRAM latency	120, 220 cycles
Topology	8 x 8 Mesh / Crossbar
Virtual Channel	4 VCs per Port (8-Flit Buffer)
Routing	DOR / Destination Tag
Flow Control	Wormhole, Credit-based
Channel Width	128 Bits / 256 Bits

Table 1: System Configuration Parameters

We select a variety of applications from multiple benchmark suites: AES, LPS, MUM, NN, NQU, RAY and STO from GPGPU-Sim [4], BFS, BP, B+tree (BT), Discrete Wavelet Transform (DWT), Gaussian Elimination (GE), HS, KM, LUD, NW, Path Finder (PF), SC and SRAD2 from Rodinia [6], CUTCP, HISTO, and SpMV from Parboil [28], PVC, PVR, SS and WC from Mars [9], and 2DCONV, 2MM and 3DCONV from Polybench [10].[2] We choose a mix of compute bound and memory bound benchmarks so as to show the prevalence of data redundancy across diverse applications.

Memory coalescing has a significant effect on reducing the number of memory requests because memory accesses from many threads are merged into smaller ones. Thus, we evaluate our packet coalescing mechanism in the presence of an intra-warp memory coalescer [25]. Also, we compare ours against a novel inter-warp memory coalescer (Warppool) [18] which merges more memory accesses from different warps on top of the intra-warp memory coalescing. As a result, Warppool can be used as an effective means to mitigate the MC bottlenecks. For fair comparison, we implemented the FIFO request selection policy both in our mechanism and in our implementation of Warppool. Note that Warppool also uses prioritization policy proposed by MRPB [13].

5.2 IPC Improvement Analysis

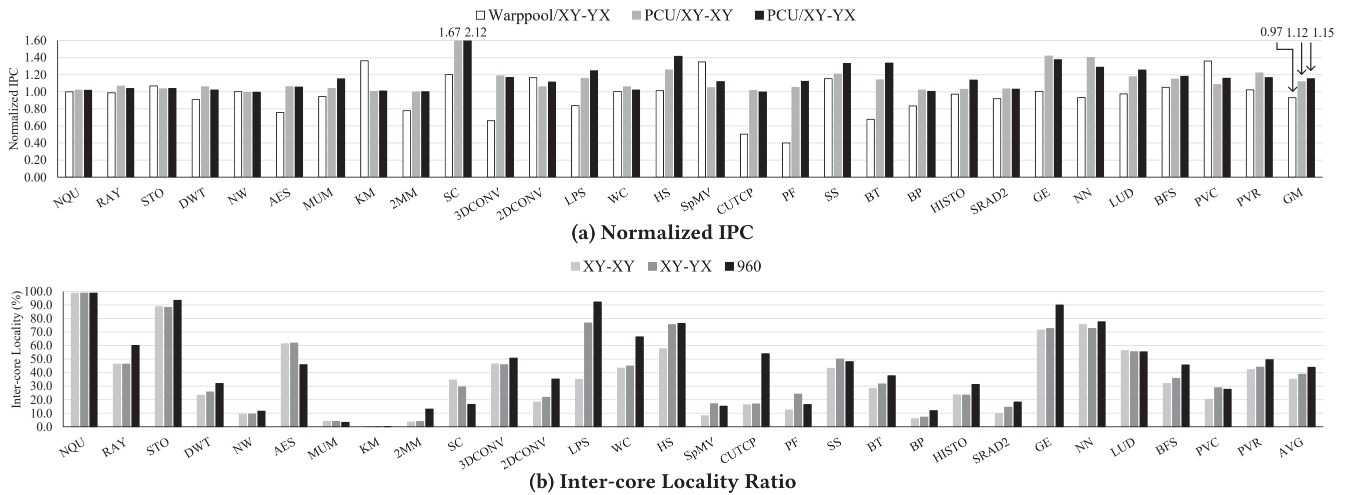
Figure 5a compares the normalized IPC of all benchmarks when Warppool is used with routing algorithm XY-YX, and PCUs are used with XY-XY and XY-YX. Each IPC is normalized over the baseline using the corresponding routing combination. Since the request grouping and reply merging in each PCU work together as a mechanism for packet coalescing, we do not show benefit for each separately.

We make two major observations in the IPC performance analysis. First, the proposed coalescing approach is more effective than Warppool. In XY-YX routing, our approach provides 15% IPC improvement on average, while Warppool does IPC performance degradation by 3%. With Warppool, only 8 out of 29 benchmarks (28%) have more than 5% IPC improvement, while others have performance degradation or minor improvement. Warppool is a novel idea, but the overhead of merging requests from different warps causes performance degradation in the benchmarks with limited inter-warp locality. Especially, since the merging process is on the critical path of cache accesses, the performance degradation appears more severe for benchmarks with low L1 data cache miss rates (e.g. PF and BT).

Second, the proposed coalescing approach becomes more effective when a better routing algorithm that mitigates reply network hotspots is used. As XY routing in the reply network causes network hotspots near MCs with bottom MC placement, YX routing has been shown more effective [11]. Our approach achieves the highest IPC improvement 15% with XY-YX routing, while it does 12% with XY-XY routing. Such performance gap arises due to worse coalescing performance in XY-XY routing, which is shown in benchmarks such as LPS, HS, SpMV, PF and PVC. Reply packets under XY routing are delivered with delay due to the network hotspots. After the reply packets are accepted by SMs, next requests with inter-core locality are sent to MCs with worse temporal locality, so PCUs are limited in involving more requests in request grouping.

Synergetic Effect of PCU and Warppool. Both PCU and Warppool synergetically improve the overall IPC performance when they work together, as shown in Figure 6. We simulate both mechanisms for benchmarks benefitting from Warppool. Both mechanisms achieve IPC improvement by 41% on average, while PCU and Warppool do by 22% and 21%, respectively. Such synergetic effect is due to the difference in the target that each mechanism works for. Warppool attempts to reduce unnecessary memory requests caused by inefficient use of an L1 cache (e.g. cache thrashing), but necessary requests to fill the L1 cache are sent and these still cause the MC bottlenecks. By reducing traffic volume of the corresponding replies with inter-core locality, the benefit from PCU keeps valid with Warppool. However, SC is more effective with PCU only.

[2]We use abbreviations of benchmarks as presented in their literatures



(a) Normalized IPC

(b) Inter-core Locality Ratio

Figure 5: System and Coalescing Performance

When Warppool is used only, the requests are waiting in SMs due to the backpropagation of MC bottlenecks, so Warppool effectively works since the latency overhead of merging requests is hidden. However, as our packet coalescing is introduced, this latency hiding is less effective since requests do not wait, thereby degrading the performance.

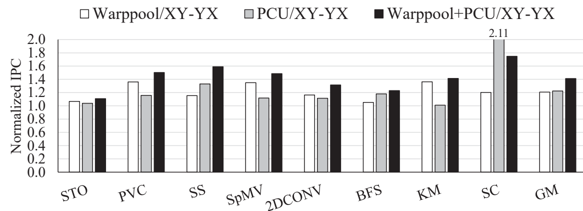


Figure 6: Comparison of Normalized IPCs

5.3 Packet Coalescing Analysis

The IPC performance improvement is mainly attributed to AMAT reduction by packet coalescing. We first analyze coalescing performance according to routing algorithms, then discuss the MC bottlenecks alleviated by the coalescing, and finally discuss memory regions with inter-core locality on two applications.

Coalescing Performance. We measure the coalescing performance based on the actual inter-core locality ratio measured by the percentage of the number of coalesced packets out of the total number of reply packets. In this analysis, we make two conclusions. First, the coalescing performance is affected by routing algorithms. As shown in Figure 5b, the actual inter-core locality ratio is 38.9% under XY-YX on average, while it is 35% under XY-XY. As discussed in Section 5.2 and [11], XY-XY routing suffers from more severe bottleneck than XY-YX in the reply network. It causes reply packets to be delivered to each SM with more delay between them. Accordingly, next requests with a potential inter-core locality are sent from each SM more sparsely. As a consequence, some requests lose a chance of grouping under XY-XY routing. However, there are some outliers that have slightly higher inter-core locality under XY-XY. For instance, SC has 34.5% and 29.4% inter-core locality ratio in XY-XY and XY-YX, respectively. Coalescing performance in SC is less sensitive to the temporal locality of accesses due to its long memory latency caused by high L2 cache miss rate (97%). The high MC bottleneck favorably gives larger time window for grouping, so extra requests are additionally grouped to existing RGRs backed up by a higher average number of coalesced packets in XY-XY than XY-YX.

Second, PCUs capture most of the requests with inter-core locality (88.4%) by using memory access time as its time window. To understand this, we conservatively compare the actual inter-core locality ratio to the potential ratio of 960-cycle time window in Figure 2a because the average memory latency is 649 cycles under XY-YX. The potential inter-core locality ratio is 44.0% on average, while the actual inter-core locality is 38.9%. By giving extra time to PCU's time window, PCUs are able to capture 5% more requests. However, the extra time becomes as a delay overhead to AMAT of 38.9% requests. This offsets the benefit of AMAT reduction by packet coalescing, thereby gaining no performance improvement.

Impact of Coalescing. We summarize two conclusions. First, our packet coalescing reduces AMAT by 15.5% and saves network bandwidth by 13%. As the packet coalescing merges multiple packets into one, the number of packets injected into the reply network is reduced by 19.7% on average, which alleviates MC stall time by 24.5% and finally leads to L1 cache miss penalty reduction by 16.3%, as shown in Figure 7. As a consequence, AMATs for L1I, L1C, L1T and L1D caches are reduced by 16.1%, 15.9%, 3.8% and 26.2%, respectively on average. The average AMAT reduction of all L1 caches is 15.5%.

Second, SC shows an interesting result where the MC stall time increases by 88% but L1 cache miss penalty is reduced. The impact of the increased stall time is minimal. The MC stall time ratio is 1.3% in the baseline as shown in Figure 1b, and increases to just 2.5% when coalescing is used. However, PCUs helps to alleviate bottlenecks caused by long memory latency. While requests accepted by MC keep waiting for their turn for memory accesses in the queue from L2 to DRAM, these backpressures back the MC request queue to be frequently full. MC node in the baseline cannot accept new requests, leaving them to wait in the request network, which

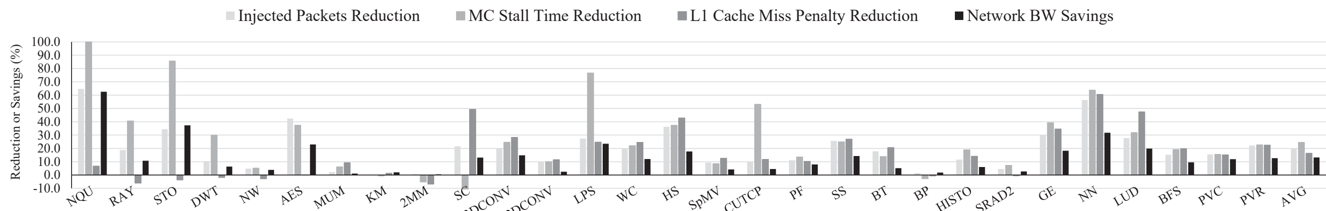


Figure 7: Injected Packets, MC Stall Time and L1 Cache Miss Penalty Reduction and Network Bandwidth Savings

makes a bottleneck. However, PCUs continue to accept requests for grouping, while MC is busy with reading data from DRAM. As a result, 29.4% requests are grouped and L1 cache miss penalty is reduced by 49.2%.

Memory Region with Inter-core Locality. Figure 8 depicts the entire memory region used by two applications, SS and LUD where the normalized degree of inter-core locality for all cache blocks is illustrated as a heatmap. To measure the degree, the number of requests with inter-core locality for each cache block is counted. Its normalized degree of inter-core locality is calculated as the count value of each block divided by the maximum count value among all cache blocks. To locate each cache block on the plot, the x and y axes indicate the row-wise and column-wise offsets from the base address of a global memory (i.e. 0x80000000).

Figure 8a shows almost all cache blocks that store an input matrix have high inter-core locality. It is because LUD kernels have many dependencies on row-wise and column-wise data [6]. LUD has three kernels such as lud_diagonal, lud_perimeter and lud_internal. Among accesses with inter-core locality, 88% and 12% occurs in lud_internal and lud_perimeter, respectively. Interestingly, cache blocks on the top-left region have higher inter-core locality than others. As LUD diagonally processes a matrix from top-left to bottom-right direction over multiple iterations, a range of data that a kernel needs to compute shrinks and thus the number of running SMs gets decreasing. Thus, the data on left side is accessed by more SMs, thus showing a higher degree of inter-core locality.

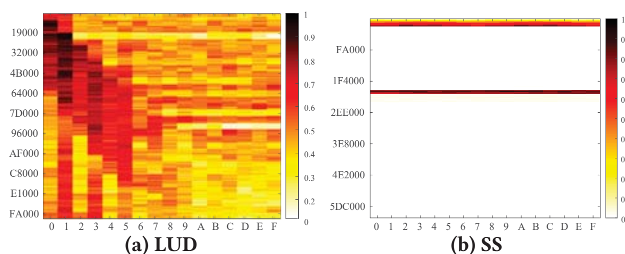


Figure 8: Memory Region with Inter-core Locality

SS usually has inter-core locality in two memory regions associated with Map and Group stage, respectively in the MapReduce framework as shown in Figure 8b. In the framework for SS, Map stage computes similarity scores for all pair-wise documents by using their feature vectors, while Group stage sorts the pair-wise similarity scores. As such pair-wise computation is performed through multiple thread blocks for large input data, Map stage running across multiple SMs needs to access feature vectors of redundant documents, which appears on the top in Figure 8b. 71% of inter-core locality occurs in this stage. Similarly, Group stage also needs to perform a pair-wise comparison between two scores to sort a series of similarity scores. 29% inter-core locality is related to Group stage, which is shown in the middle of Figure 8b. As Group stage is necessary for MapReduce-based applications, PVC, PVR and WC also have inter-core locality in this stage. While the sorting process in the stage requires frequent data movement [9], our coalescing unit effectively eliminates the bottlenecks caused by the data movement.

5.4 Sensitivity Analysis

Impact of RGR Size. We evaluate IPC performance and coalescing performance across all benchmarks as varying the number of RGRs: 64, 128, 256, 512 and 1204. Our coalescing technique achieves 39.4% inter-core locality ratio on average and 19% IPC improvement with 1024 RGRs across all benchmarks. Most benchmarks achieve saturated IPC performance at 128 RGRs except four benchmarks shown in Figure 9. While three benchmarks such as SC, SpMV and SS gain saturated performance at 256 RGRs, MUM obtains a monotonically increasing IPC improvements until 1024 RGRs are used. As the number of RGRs grows from 128 to 1024 in MUM, the inter-core locality ratio increases from 4% to 16% as shown in Figure 9, and the IPC improvement does from 15% to 88%. This happens because MUM has higher memory intensity than others [6].

Impact of L2 Hit Latency. To exploit a long hit latency of L2 cache for request grouping, we place PCUs before L2 cache as discussed in Section 3.2. We study the impact of a shorter L2 hit latency on coalescing performance. We model the L2 cache hit latency as 2 cycles based on CACTI model [23]. Figure 10 shows the normalized IPCs when the minimum L2 hit latency is set to 120 and 2 cycles, respectively. The IPC values of two configurations are normalized against the baseline with corresponding L2 latency. It also plots inter-core locality ratio as a line for each latency case. The IPC improvement increases up to 24% on average at 2-cycle hit latency, while it is 15% at 120-cycle hit latency. However, the average inter-core locality ratios do not show noticeable differences, which are 39% and 37% in the 120-cycle and 2-cycle latencies, respectively. As the cache access time is reduced in the 2-cycle case, the injection rate of reply packets becomes higher, which causes more severe MC stalls. Our coalescing becomes relatively more effective as a bottleneck alleviator in the 2-cycle case, resulting in higher IPC improvement.

Impact of MC placement. We compare four configurations such as bottom, top-bottom, edge and diamond MC placements studied in the previous literature [11]. The IPC values of all different configurations are normalized against the baseline with corresponding MC placements and XY-YX routing. Our coalescing technique achieves similar average IPC improvements, 15%, 15%, 14% in bottom, top-bottom and edge MC placements, while it does

lower improvement, 11% in diamond (unplotted). The diamond MC placement is commonly known as the optimal placement [1], but it is not when multicast is used. When MC nodes serve as a replication point of multicast packets, it causes contention between replicated packets and injected packets. As a result, it offsets the benefit of the MC bottlenecks lessened by our coalescing technique.

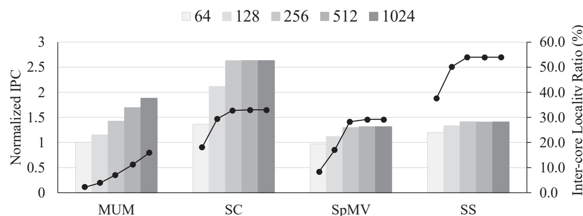


Figure 9: Normalized IPC (Bar) and Inter-core Locality Ratio (Line) as varying the number of RGRs

5.5 Coalescing in the Global Crossbar

We choose 17 benchmarks with MC bottlenecks under the crossbar such as MUM, KM, 2MM, SC, 3DCONV, 2DCONV, LPS, WC, SpMV, SS, BT, HISTO, SRAD2, GE, BFS, PVC and PVR. Our coalescing technique achieves 30.3% of inter-core locality ratio and yields 7% IPC improvement on average (unplotted). On the other hand, Warppool obtains 28% performance degradation on average and only shows IPC improvement for a few benchmarks such as KM, SC, SS and PVC by 21%, 6%, 10% and 23%, respectively.

5.6 Hardware Cost

Coalescing Overhead. We analyze the area overhead incurred by the proposed PCUs. Since a PCU uses 128 RGRs and PCU pointers which take 14B and 7 bits, respectively, a PCU per MC incurs the total overhead 1904B. As shown in Table 1, an SM has 86KB L1 caches and an MC has 64KB L2 cache. Compared to the total cache infrastructure of 56 SMs and 8 MCs, the total overhead incurred is just 0.28%. The overheads of RGR are summarized in Table 2.

Size	Access Time (ns)	Energy (J)	Leakage Pwr (W)
64	0.19	3.66E-12	2.92E-04
128	0.20	6.50E-12	5.37E-04
256	0.21	1.41E-11	1.07E-03
512	0.22	2.56E-11	2.14E-03
1024	0.26	4.16E-11	4.18E-03

Table 2: RGR Overhead

Multicast Overhead. The hardware overhead of a DOR-based multicast router has two parts. First, multiple RC units incurs an overhead to support lookahead routing. $3 \sim 4$ RC units per each input port are added to the baseline router. A multicast RC unit needs at most 59 OR gates for 64 destination nodes, so a router needs 944 OR gates which accumulate to the total area overhead of 0.40% per router based on DSENT [29]. Second, we adopted the mux-based crossbar that has been used by several multicast routers such as RPM [31] and BAM [21], to avoid serialization delay of replication in the matrix-crossbar. The mux-based crossbar has been analyzed to consume more energy than the matrix-crossbar [19]. However, our mechanism can be built on energy-efficient crossbar, mXbar that supports single-cycle replication with small energy overhead or even better energy efficiency compared to the matrix-crossbar [19].

6 RELATED WORK

Network Design in GPU. Bakhoda et al. [3] proposed a cost-efficient checkerboard router design with multi-ported routers for MCs to increase MC network injection bandwidth, for many read replies. Jang et al. [11] introduced a bandwidth efficient network design for GPU traffic through VC monopolization and partitioning. Ziabari et al. [32] explored asymmetric NoC designs where the reply subnetwork is provided with larger channel width. However, ours differs because we directly reduce the heavy reply traffic exploiting data redundancy. Hsu et al. [7] proposed a packet coalescing mechanism, but this study is applied to request network to rearrange memory requests for enhancing row buffer hits in DRAM. It improves DRAM bandwidth but does not reduce traffic volume in reply network as our coalescing approach.

Reducing Global Memory Access Demand in GPU. To alleviate high demand on global memory system, an intra-warp memory coalescer [25] and an inter-warp memory coalescer (Warppool) [18] were proposed, which has been compared to our technique. Jia et al. [13] proposed memory request prioritization method for effective caching, but it is limited to cache-sensitive applications. Dongdong et al proposed a DRAM scheduler exploiting inter-core locality to reduce memory access latency [20], which is orthogonal to our packet coalescing technique.

Compression in GPU Interconnect/Memory. Data compression has been studied in GPGPUs. Pekhimenko et al addressed a problem of increased dynamic energy caused by frequent communication switching of compressed data traffic [26]. To alleviate the off-chip memory bandwidth bottleneck, Sathish et al applied both lossless compression and lossy compression [27]. The data compression is complementary to our coalescing technique because ours reduce the number of packets, while compression mechanism reduces the size of each packet.

Warp Schedulers in GPU. GPGPU performance has been improved by novel warp scheduling policies. Narasiman et al proposed two-level scheduling that increases core utilization [24], and Jog et al proposed OWL scheduler that improves both L1 hit rate and DRAM bandwidth utilization [15]. As a homogeneous scheduling policy works across SMs, the inter-core locality patterns are maintained, so that the novel schedulers with our packet coalescing can synergetically improve performance.

7 CONCLUSIONS

In this paper, we identify that the performance of GPGPU applications is significantly impacted by MC bottlenecks near the MCs. To address this issue, we propose to reduce the traffic volume in the reply network from MCs to SMs by introducing PCUs in MCs. The key idea is to coalesce read reply packets in MCs when they deliver the same cache block to multiple SMs. To ensure that the coalesced packets arrive at the respective requesting SMs, we support multicast for the interconnection network. To the best of our knowledge, this is the first work showing a good use of multicast in GPGPUs. Our extensive evaluations across a wide range of benchmarks show that PCUs coupled with XY-YX routing obtain 15.5% AMAT reduction (up to 65.2%) and 13% network bandwidth savings (up to 67.8%) in a large-scale GPGPU with 2D mesh, and thus improve overall IPC by 15% (up to 112%) on average. Also, our coalescing approach achieves 7% IPC improvement in a GPGPU with the crossbar.

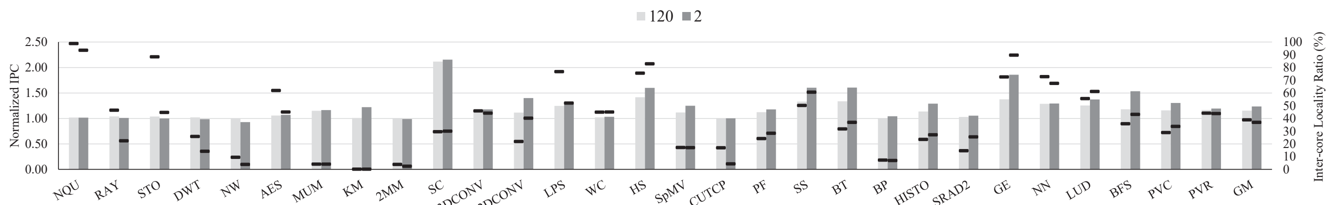


Figure 10: Normalized IPC (Bar) and Inter-core Locality Ratio (Line) when a minimum L2 hit latency is 120 and 2 cycles

ACKNOWLEDGMENTS

The authors would like to thank anonymous reviewers and Dr. Mutlu for their valuable comments and helpful suggestions. This work is supported by NSF award CCF-1423433.

REFERENCES

[1] Dennis Abts, Natalie Enright, Jerger John Kim, Dan Gibson, and Mikko Lipasti. 2009. Achieving predictable performance through better memory controller placement in many-core cmps. In *Proceedings of the 36th Annual International Symposium on Computer architecture (ISCA-36)*.

[2] Rachata Ausavarungnirun, Kevin Kai-Wei Chang, Lavanya Subramanian, Gabriel H. Loh, and Onur Mutlu. 2012. Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA-39)*. 416–427.

[3] Ali Bakhoda, John Kim, and Tor M. Aamodt. 2010. Throughput-Effective On-Chip Networks for Manycore Accelerators. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-43)*. 421–432.

[4] A. Bakhoda, G.L. Yuan, W.W.L. Fung, H. Wong, and T.M. Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *Proceedings of International Symposium on Performance Analysis of Systems and Software (ISPASS 2009)*. 163–174.

[5] James Balfour and William J. Dally. 2006. Design Tradeoffs for Tiled CMP On-chip Networks. In *Proceedings of the 20th Annual International Conference on Supercomputing (ICS 2006)*. 187–198.

[6] Shuai Che, J.W. Sheaffer, M. Boyer, L.G. Szafaryn, Liang Wang, and K. Skadron. 2010. A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads. In *Proceedings of International Symposium on Workload Characterization (IISWC 2010)*. 1–11.

[7] Chien-Ting Chen, YoshiShih-Chieh Huang, Yuan-Ying Chang, Chiao-Yun Tu, Chung-Ta King, Tai-Yuan Wang, Janche Sang, and Ming-Hua Li. 2014. Designing Coalescing Network-on-Chip for Efficient Memory Accesses of GPGPUs. In *Network and Parallel Computing*. 169–180.

[8] William Dally and Brian Towles. 2003. *Principles and Practices of Interconnection Networks.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[9] W. Fang, B. He, Q. Luo, and N. K. Govindaraju. 2011. Mars: Accelerating MapReduce with Graphics Processors. *IEEE Transactions on Parallel and Distributed Systems* 22, 4 (2011), 608–620.

[10] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. 2012. Auto-tuning a high-level language targeted to GPU codes. In *Innovative Parallel Computing (InPar), 2012*. 1–10.

[11] Hyunjun Jang, Jinchun Kim, P. Gratz, Ki Hwan Yum, and Eun Jung Kim. 2015. Bandwidth-efficient on-chip interconnect designs for GPGPUs. In *Proceedings of the 52nd Anual ACM/EDAC/IEEE Design Automation Conference (DAC 2015)*. 1–6.

[12] Natalie Enright Jerger, Li-Shiuan Peh, and Mikko Lipasti. 2008. Virtual circuit tree multicasting: A case for on-chip hardware multicast support. In *Proceedings of THE 35th Anual International Symposium on Computer Architecture (ISCA-35)*. 229–240.

[13] Wenhao Jia, K.A. Shaw, and M. Martonosi. 2014. MRPB: Memory request prioritization for massively parallel processors. In *Proceedings of the 20th Anual International Symposium on High Performance Computer Architecture (HPCA-20)*. 272–283.

[14] Nan Jiang, D.U. Becker, G. Michelogiannakis, J. Balfour, B. Towles, D.E. Shaw, J. Kim, and W.J. Dally. 2013. A detailed and flexible cycle-accurate Network-on-Chip simulator. In *Proceedings of 2013 IEEE Inernational Symposium on Performance Analysis of Systems and Software (ISPASS 2013)*. 86–96.

[15] Adwait Jog, Onur Kayiran, Nachiappan Chidambaram Nachiappan, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. 2013. OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. 395–406.

[16] Adwait Jog, Onur Kayiran, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. 2013. Orchestrated Scheduling and Prefetching for GPGPUs. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA-40)*. 332–343.

[17] S.W. Keckler, W.J. Dally, B. Khailany, M. Garland, and D. Glasco. 2011. GPUs and the Future of Parallel Computing. *Micro, IEEE* 31, 5 (2011), 7–17.

[18] John Kloosterman, Jonathan Beaumont, Mick Wollman, Ankit Sethia, Ron Dreslinski, Trevor Mudge, and Scott Mahlke. 2015. WarpPool: Sharing Requests with Inter-warp Coalescing for Throughput Processors. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. 433–444.

[19] Tushar Krishna, Li-Shiuan Peh, Bradford M. Beckmann, and Steven K. Reinhardt. 2011. Towards the Ideal On-chip Fabric for 1-to-many and Many-to-1 Communication. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*. 71–82.

[20] D. Li and T. M. Aamodt. 2016. Inter-Core Locality Aware Memory Scheduling. *IEEE Computer Architecture Letters* 15, 1 (2016), 25–28.

[21] Sheng Ma, N.E. Jerger, and Zhiying Wang. 2012. Supporting efficient collective communication in NoCs. In *Proceedings of the 18th IEEE International Symposium on High Performance Computer Architecture (HPCA-18)*. 1–12.

[22] Jiayuan Meng and Kevin Skadron. 2009. Performance Modeling and Automatic Ghost Zone Optimization for Iterative Stencil Loops on GPUs. In *Proceedings of the 23rd International Conference on Supercomputing (ICS '09)*. 256–265.

[23] Naveen Muralimanohar, Rajeev Balasubramonian, and Norm Jouppi. 2007. Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-40)*. 3–14.

[24] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt. 2011. Improving GPU performance via large warps and two-level warp scheduling. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 308–317.

[25] NVIDIA. 2009. Fermi: NVIDIA's Next Generation CUDA Compute Architecture. (2009).

[26] G. Pekhimenko, E. Bolotin, N. Vijaykumar, O. Mutlu, T. C. Mowry, and S. W. Keckler. 2016. A case for toggle-aware compression for GPU systems. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 188–200.

[27] Vijay Sathish, Michael J. Schulte, and Nam Sung Kim. 2012. Lossless and Lossy Memory I/O Link Compression for Improving Performance of GPGPU Workloads. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT '12)*. 325–334.

[28] John A. Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, vLi Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen mei W. Hwu. 2012. *Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing.* Technical Report IMPACT-12-01. University of Illinois at Urbana-Champaign, East Lansing, Michigan.

[29] Chen Sun, C.-H.O. Chen, G. Kurian, Lan Wei, J. Miller, A. Agarwal, Li-Shiuan Peh, and V. Stojanovic. 2012. DSENT - A Tool Connecting Emerging Photonics with Electronics for Opto-Electronic Networks-on-Chip Modeling. In *Proceedings of the 6th IEEE/ACM International Symposium on Networks-on-Chip (NOCS 2012)*. 201–210.

[30] N. Vijaykumar, G. Pekhimenko, A. Jog, A. Bhowmick, R. Ausavarungnirun, C. Das, M. Kandemir, T.C. Mowry, and O. Mutlu. 2015. A case for Core-Assisted Bottleneck Acceleration in GPUs: Enabling flexible data compression with assist warps. In *Proceedings of the 42nd ACM/IEEE Anual International Symposium on Computer Architecture (ISCA-42)*. 41–53.

[31] Lei Wang, Yuho Jin, Hyungjun Kim, and Eun Jung Kim. 2009. Recursive partitioning multicast: A bandwidth-efficient routing for Networks-on-Chip. In *Proceedings of the 3rd ACM/IEEE International Symposium on Networks-on-Chip (NoCS 2009)*. 64–73.

[32] Amir Kavyan Ziabari, José L. Abellán, Yenai Ma, Ajay Joshi, and David Kaeli. 2015. Asymmetric NoC Architectures for GPU Systems. In *Proceedings of the 9th Anual International Symposium on Networks-on-Chip (NOCS 2015)*. Article 25, 25:1–25:8 pages.