2018 IEEE International Parallel and Distributed Processing Symposium

Intra-Cluster Coalescing to Reduce GPU NoC Pressure

Lu Wang[1], Xia Zhao[1], David Kaeli[2], Zhiying Wang[3], and Lieven Eeckhout[1]

[1]Ghent University,{luluwang.wang, xia.zhao, lieven.eeckhout}@UGent.be
[2]Northeastern University, kaeli@ece.neu.edu
[3]National University of Defense Technology, zywang@nudt.edu.cn

***Abstract*—GPUs continue to increase the number of streaming multiprocessors (SMs) to provide increasingly higher compute capabilities. To construct a scalable crossbar network-on-chip (NoC) that connects the SMs to the memory controllers, a cluster structure is introduced in modern GPUs in which several SMs are grouped together to share a network port. Because of network port sharing, clustered GPUs face severe NoC congestion, which creates a critical performance bottleneck.**

In this paper, we target redundant network traffic to mitigate GPU NoC congestion. In particular, we observe that in many GPU-compute applications, different SMs in a cluster access shared data. Issuing redundant requests to access the same memory location wastes valuable NoC bandwidth — we find on average 19.4% (and up to 48%) of the requests to be redundant. To reduce redundant NoC traffic, we propose intra-cluster coalescing (ICC) to merge memory requests from different SMs in a cluster. Our evaluation results show that ICC achieves an average performance improvement of 9.7% (and up to 33%) over a conventional design.

I. Introduction

Graphics Processing Units (GPUs) are widely deployed in modern computing systems to provide high performance for a wide class of general-purpose applications. A GPU-compute application typically consists of several kernels that are composed of (up to hundreds of) thousands of threads. These threads are organized into cooperative thread arrays (CTAs) that are scheduled on streaming multiprocessors (SMs). To continuously increase the raw computational power of modern GPUs, the SM count keeps increasing. Whereas the Nvidia Fermi GPU implemented 16 SMs, the recent Nvidia Pascal [1] and the current Volta GPUs [2] feature 60 and 84 SMs, respectively.

The SMs feature private L1 caches and are connected to the L2 cache and memory controllers (MCs) through a Network-on-Chip (NoC). With the large number of SMs we are observing today, designing a scalable NoC poses a challenge. Typically, a crossbar is deployed as the GPU's NoC due to its low latency and high bandwidth [1]. However, a crossbar NoC faces scalability issues as hardware costs increase quadratically with increasing port count.

To address the GPU NoC scalability challenge, a cluster structure is implemented in modern-day GPUs to group several SMs into a cluster. For example, Pascal supports 6 clusters, with each cluster consisting of 10 SMs [1]; Volta features 14 SMs per cluster for the same number of clusters [2]. By sharing NoC ports among SMs in a cluster, the total number of ports to the network is reduced and so is the overall hardware cost of the crossbar NoC.

Previous research has shown that NoC congestion is a severe GPU performance bottleneck for many memory-intensive applications [3], [4], [5]. Unfortunately, clustered GPUs further exacerbate this performance issue. By sharing ports among SMs in a cluster, congestion significantly increases as SMs need to compete with each other in a cluster for network bandwidth. This creates a new and critical performance challenge for the NoC in clustered GPU organizations.

In this paper, we address the GPU NoC performance bottleneck by reducing NoC traffic, and more specifically by eliminating redundant NoC requests. We do this by coalescing L1 cache misses from different SMs within a cluster before sending them to the NoC. L1 cache miss coalescing not only reduces NoC pressure, it also reduces L1 cache miss latency leading to overall performance improvements.

Memory coalescing, or grouping memory accesses from different threads to the same cache line in a single memory request, is widely deployed in a GPU. More specifically, intra-warp coalescing merges L1 cache accesses across threads within a warp [6]; WarpPool merges L1 accesses across warps within the same SM [7]; L1 MSHRs merge L1 misses across warps within a single SM. However, to the best of our knowledge, **no prior work coalesces L1 misses across SMs within a cluster**.

In this paper, we make the observation that many GPU-compute applications exhibit **inter-CTA locality**, as different CTAs access the same cache line or access the same read-only data. For clustered GPUs, this implies that memory requests from CTAs that execute on the same cluster will access the same cache lines. According to our experimental results, we find that on average 19.4% (and up to 48%) of all L1 misses originating from a cluster indeed access the same cache line. These memory requests are redundant and can be eliminated.

In response, we propose **intra-cluster coalescing (ICC)** to reduce GPU NoC pressure. Intra-cluster coalescing groups memory requests, from different SMs in a cluster, to the same L2 cache line to reduce NoC traffic. In particular, ICC records the memory requests sent to the NoC, and when subsequent memory requests access the same cache lines as outstanding requests, ICC coalesces them. By doing so, ICC significantly reduces NoC traffic.

In this paper, we make the following contributions:

- We observe that GPU-compute applications exhibit high degrees of inter-CTA locality. We analyze and categorize the sources of data sharing among CTAs.

1530-2075/18/$31.00 ©2018 IEEE
DOI 10.1109/IPDPS.2018.00108

- We propose *Intra-Cluster Coalescing (ICC)* to track and coalesce L1 cache misses from different SMs in a cluster before issuing them across the NoC.
- We demonstrate the significant interaction between ICC and CTA scheduling, i.e., ICC benefits more when the CTA scheduling policy maps neighboring CTAs to the same cluster to better exploit inter-CTA locality.
- We comprehensively evaluate ICC and demonstrate an average performance improvement of 9.7%, and up to 33%, over a state-of-the-art distributed CTA scheduling policy [8]. The hardware cost is limited to 276 bytes per cluster.

II. Background

Before motivating the problem we are addressing in this paper more deeply, we first summarize some background information.

A. GPU Thread Hierarchy

Using Nvidia's terminology, a GPU-compute application consists of kernels, grids, CTAs, warps and threads, organized in a hierarchy. A kernel is a parallel code region that runs on a GPU and consists of multiple grids, which in turn consists of multiple CTAs. Each CTA is a batch of threads that can coordinate with each other through synchronization using a barrier instruction [9]. Threads in a CTA share a fast, on-chip scratchpad memory called shared memory. Since all the synchronization primitives are encapsulated within a CTA, different CTAs can be executed in any order. This is an important feature that we will explore to understand how the mapping of CTAs to clusters affects intra-cluster locality.

B. GPU Architecture

Our baseline GPU architecture is shown in Figure 1: 12 clusters are connected via a crossbar NoC to 8 memory controllers (MCs). Each MC has an associated L2 cache bank for the memory partition that the MC serves, and has one network port. Each cluster consists of 5 SMs, so there are 60 SMs in total. Each SM has a private L1 data cache, a read-only texture cache, a constant cache and shared memory. An L1 cache miss triggers a request to be sent over the NoC to reach one of the L2 cache banks; in case of an L2 cache miss, the request proceeds to main memory. In our baseline architecture, we assume one NoC injection port buffer that is shared by all SMs in a cluster. (In the evaluation section, we will study the sensitivity of our design to the number of clusters and the effective network ports per SM.) Each cluster has a response FIFO queue to hold incoming packets from the NoC; responses are directed to one of the SMs in the cluster according to the control information in the packet.

C. CTA Scheduling

Scheduling on a GPU is done in three steps. First, a kernel is launched on the GPU. In this work, we assume that only one kernel is active at a given time. Second, the CTA scheduler maps CTAs to the available SMs. The baseline CTA scheduler follows a 2-level round-robin (RR) policy [10], which first schedules CTAs across clusters and then across SMs within a cluster. In particular, CTA 1 is allocated to the first SM in cluster #1, CTA 2 is allocated to the first SM in cluster #2, and so on. Once all clusters are assigned one CTA, the next iteration allocates a CTA to the second SM in each cluster, etc., until all SMs are assigned one CTA. If an SM has enough resources to execute more than one CTA, additional CTAs are assigned — this is done in a round-robin manner similar to the procedure just described. By doing so, a two-level RR policy balances the load among clusters and SMs, so that all clusters and SMs have a similar number of CTAs to execute. The maximum number of CTAs that can be scheduled per SM is determined by the SM's resources. Finally, the warp scheduler in each SM schedules warps (from one or more CTAs) to execute, which we model to follow the Greedy-Then-Oldest (GTO) policy [11].

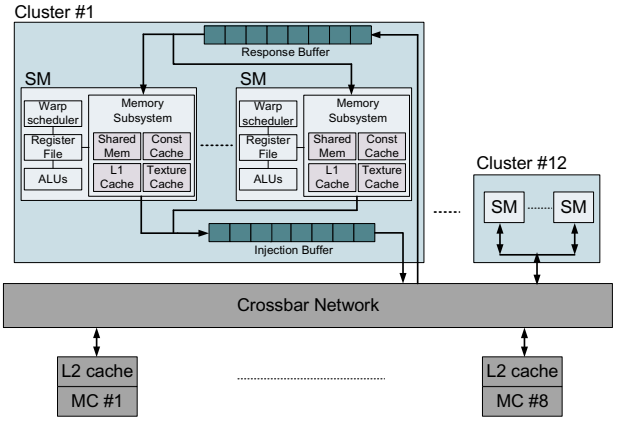

Fig. 1. Clustered GPU architecture: *SMs within a cluster go through the NoC to access the L2 cache and main memory to serve L1 cache misses.*

III. Motivation and Opportunity

We now further motivate the problem and describe the opportunity.

A. NoC Bandwidth Bottleneck

We first demonstrate that the NoC indeed constitutes a performance bottleneck in a clustered GPU architecture. In particular, we study how sensitive performance is to NoC bandwidth. Figure 2 quantifies performance when increasing the NoC bandwidth by $2\times$. To ensure an overall balanced design, we also increase the LLC bandwidth accordingly. This is done by increasing the clock frequency of the NoC and LLC subsystems by $2\times$. From an implementation and power perspective, this may not be a feasible design point, however, it provides us with a meaningful measure for how sensitive performance is to the available NoC (and LLC) bandwidth. (Further details about our experimental setup are given in Section VI.) We find that performance increases for all benchmarks, up to 78%, with an average improvement of 41.4%. This clearly demonstrates that NoC bandwidth indeed is a severe bottleneck. Limited NoC bandwidth leads to

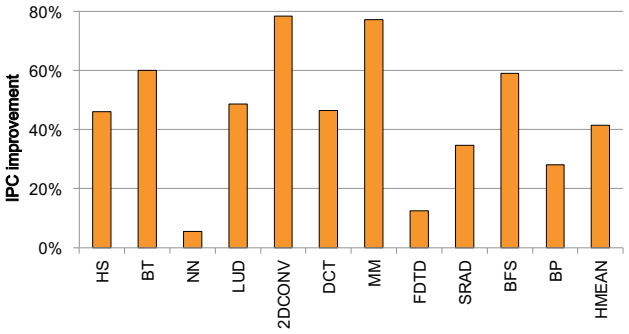

Fig. 2. Quantifying the NoC bottleneck: IPC improvement when increasing the NoC and LLC frequency by 2×. *NoC (and LLC) bandwidth is a severe performance bottleneck.*

TABLE I
GPU COALESCING TECHNIQUES AND THEIR SCOPE.

Technique	Scope
Intra-warp coalescing [6]	Across threads in a warp
WarpPool [7]	L1 accesses across warps in an SM
L1 MSHR [12]	L1 misses across warps in an SM
ICC (this work)	L1 misses across SMs in a cluster

congestion within a cluster for memory requests that need to proceed through the NoC to reach the L2 cache and beyond.

B. Request Merging

GPU-compute applications exhibit various forms of locality in the memory hierarchy. Merging memory requests is widely deployed across the memory hierarchy in a GPU to increase the effective memory system throughput. Table I provides a comparison between existing techniques and our work.

Intra-warp locality, or different threads within the same warp accessing the same or neighboring memory locations, is the most common and obvious form of data locality present in GPU-compute applications. To exploit this characteristic, a memory coalescing unit merges multiple memory accesses to the same cache line within the same warp before sending the request to the L1 cache [6]. In other words, **intra-warp coalescing merges requests across threads within a warp**. This is easily done as different threads within a warp execute in SIMD lockstep.

For memory-divergent applications, where different threads in a warp request more than one cache line in a load or store instruction, the memory coalescing unit becomes a memory system throughput bottleneck because the different memory requests now need to be serialized. Kloosterman et al. [7] propose **WarpPool** which **merges memory requests across warps in an SM** before accessing the L1 cache. By merging requests from different warps in an SM, they increase the effective L1 cache bandwidth. WarpPool does not address NoC congestion though: WarpPool reduces the number of requests to the L1 cache, but goes no further. SMs in the same cluster that are accessing the same address, an address that presently is not in the L1 cache, generate multiple NoC requests.

Miss Status Handling Registers (MSHRs) are used at the L1 cache level to track outstanding L1 cache misses and merge multiple requests to the same cache line in the L2 cache and beyond. This avoids having to send redundant requests over the NoC to the next level in the cache hierarchy. Note that L1 MSHRs eliminate redundant NoC requests originating from a single SM. In other words, L1 cache **MSHRs** are limited in scope and **coalesce L1 cache misses across warps within an SM**. There may still be redundant NoC requests originating from different SMs within a single cluster, as we will demonstrate in this paper.

To summarize, although intra-warp coalescing and WarpPool reduce the number of requests to the L1 cache and although L1 MSHRs merge outstanding L1 cache misses, there is no coalescing or merging happening for accesses to the L2 cache. In other words, different SMs within the same cluster may issue multiple requests to the same or neighboring data elements, which leads to redundant NoC traffic. In this paper, we **eliminate redundant NoC traffic by coalescing L1 cache misses across SMs within a cluster** before sending requests to the L2 cache. By doing so, we increase the effective NoC bandwidth.

C. Intra-Cluster Locality

In this paper, we observe and exploit the notion of intra-cluster data locality in GPU-compute applications. In this section, we first quantify intra-cluster locality, and we then investigate its root cause.

1) Quantifying Intra-Cluster Locality: To quantify intra-cluster locality, we first define the notion of a *redundant request*. A data request is said to be redundant if it accesses a cache block that has been accessed by a previous request from the same cluster; the previous request needs to have happened recently, within a given window size of requests prior to the current request. (We will vary this window size when we quantify intra-cluster locality.) We define *Intra-Cluster Locality (ICL)* as

$$ICL = \frac{no.\ redundant\ requests}{total\ no.\ data\ requests}. \quad (1)$$

To quantify intra-cluster locality, we track all data requests in a cluster before they are injected into the NoC, i.e., after having accessed the L1 cache, so this includes all L1 misses. We then calculate the ratio of redundant requests to the total number of data requests for different window sizes of past memory requests. We consider window sizes ranging from 500 to 2000 cycles. The reason for this wide range is that we observe L1 cache miss latencies ranging up to a couple thousands of cycles, which we observe for some of our benchmarks that suffer from severe NoC congestion.

Different applications exhibit different degrees of intra-cluster locality, see Figure 3. On average, for a window size of 2000 cycles, we observe that 19.4% of the memory requests are redundant. For HS and DCT, up to 48% and 45.4% of the requests are redundant at the cluster level, respectively. This result supports the hypothesis in this paper that it is

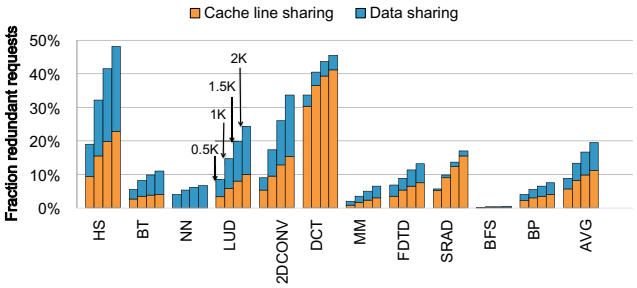

Fig. 3. Intra-cluster locality (fraction redundant requests versus total number of requests in a cluster) as a function of a past window of requests under the distributed CTA scheduling policy. A distinction is made between cache line sharing and data sharing. *A substantial fraction of NoC requests are redundant because of intra-cluster locality due to cache line sharing or data sharing.*

possible to significantly reduce NoC traffic in clustered GPUs by coalescing memory requests within a cluster.

2) Inter-CTA Locality: It is interesting to investigate where intra-cluster locality comes from. Intra-cluster locality in fact stems from **inter-CTA locality** because of data reuse among CTAs mapped to SM cores in the same cluster. We identify two categories of inter-CTA locality. Figure 3 quantifies their relative contribution.

(1) Inter-CTA locality due to cache line sharing. Inter-CTA locality may result from adjacent CTAs accessing neighboring data items in the same cache line. If one cache line is big enough to hold the data accessed by multiple CTAs, we may observe this form of inter-CTA locality. The number of threads within a CTA is typically a multiple of 32. It may be the case that all threads within a CTA access less than a cache line worth of data, e.g., 32 or 64 threads in a CTA access 128 or fewer bytes. Hence, for a cache line of 128 bytes, this implies that different CTAs will access the same cache line, exhibiting inter-CTA locality through the same cache line. A couple benchmarks feature cache line sharing predominantly, especially DCT and SRAD, see Figure 3.

(2) Inter-CTA locality due to data sharing. In many GPU-compute applications, we observe that different CTAs access the *same* (read-only) data. Data sharing may result from different reuse patterns depending on how the CTAs are organized.

We illustrate this using two benchmarks. Hotspot (HS), see Figure 4 for a code excerpt, is a benchmark that exhibits high intra-cluster locality. HS has its threads and CTAs organized in a 2D structure. Different threads in different CTAs access the same data through the `power[]` data structure. The computed `index` is a linear combination of the two-dimensional index of the thread and CTA. If this linear combination evaluates to the same value, different threads from different CTAs will access the same data, yielding inter-CTA locality.

LUD is another example 2D application, see Figure 5, in which each submatrix L_{ij} and U_{ij} is processed by one CTA. One iteration (one instance of the kernel) is used to calculate the decomposition of one row and column of submatrices. For example, in the first iteration, submatrices L_{j1} and U_{1i} are

```
int small_block_rows = BLOCK_SIZE - border_rows × 2;
int small_block_cols = BLOCK_SIZE - border_cols × 2;
int ty = small_block_rows ×blockIdx.y + threadIdx.y - border_rows;
int tx = small_block_rows × blockIdx.x + threadIdx.x - border_cols;
index=grid_cols×ty+tx
if (0<ty<grid_rows-1) && (0<tx<grid_cols - 1))
power_on_cuda[ty][tx] = power[index];
```

Fig. 4. Code excerpt for hotspot (HS). *Different threads in different CTAs access the same data through the* `power[]` *data structure if the* `index` *evaluates to the same value.*

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 & 0 \\ L_{21} & L_{22} & 0 \\ L_{31} & L_{32} & L_{33} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{bmatrix}$$

First row/column sub-matrix calculation:

$$A_{11} = L_{11} \times U_{11}$$

$$U_{12} = \frac{A_{12}}{L_{11}}; U_{13} = \frac{A_{13}}{L_{11}}\ ;\ L_{21} = \frac{A_{21}}{U_{11}}; L_{31} = \frac{A_{31}}{U_{11}}$$

Fig. 5. Data sharing in LUD. L_{11} *is reused for calculating submatrices* U_{12} *and* U_{13} *(reuse along rows), while* U_{11} *is reused for calculating submatrices* L_{21} *and* L_{31} *(reuse along columns).*

computed: L_{11} is reused for calculating submatrices U_{12} and U_{13} (reuse along rows), while U_{11} is reused for calculating submatrices L_{21} and L_{31} (reuse along columns).

IV. Intra-Cluster Coalescing (ICC)

Based on the notion of inter-CTA locality, we propose *intra-cluster coalescing (ICC)*. The key idea is to merge requests from different SMs in a cluster to the same L2 cache line before issuing the request to the NoC.

A. ICC Unit

Figure 6 illustrates the overall architecture of the intra-cluster coalescing unit. The central structure of the ICC unit is the *merge table*. Its goal is to track all memory requests coming from the SMs in the cluster before injecting them into the network. To achieve this, the merge table contains multiple entries. Each entry consists of three fields, namely an address field, the SM list and a valid bit. An entry is responsible for coalescing all memory requests to the same L2 cache line. The merge table is implemented as a fully-associative cache.

When an SM core wants to inject a memory request into the network, the ICC unit first searches the merge table using the request's address. If there already exists an entry for the requested cache line (a merge table hit), the ICC unit will append the ID of the requesting SM to the SM list. The memory request will not be sent to the network — there already is a request outstanding for that same L2 cache line. If on the other hand, there is no entry allocated in the merge table for that cache line (a merge table miss), the ICC unit

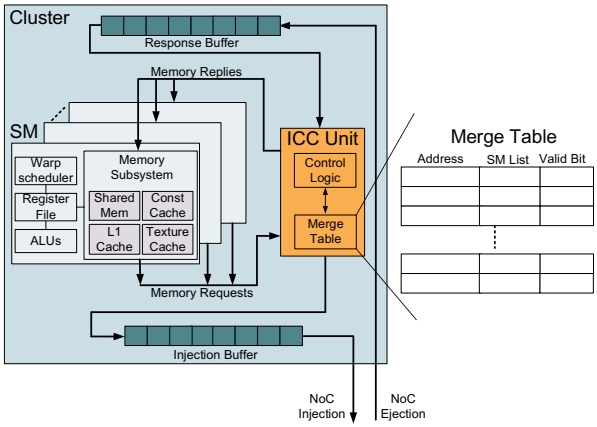

Fig. 6. The intra-cluster coalescing (ICC) unit merges L1 cache misses across SMs within a cluster.

will allocate a new entry (if the merge table has empty entries available) and then send the memory request to the network; the SM sending this request is added to the SM list and the valid bit is set. If, under a merge table miss, all entries in the merge table are occupied, the memory request will be injected into the NoC directly. To increase the effective utilization of the merge table, ICC only records the memory read requests to global memory, as they account for a large fraction of all memory requests and, in addition, have a big impact on performance. We do not consider write requests, i.e., all write requests bypass the merge table.

When a cluster receives a reply packet from the network, the ICC unit first uses the reply address to index the merge table. If there exists an entry for that address (a merge table hit), the ICC unit will read the corresponding SM list and broadcast the memory reply to all SMs in the list. Next, the corresponding entry in the merge table is set to invalid, which means that the entry can be re-used for other memory requests. If the address cannot be found in the merge table (a merge table miss), the reply will be delivered to the SM based on the destination stored in the reply packet.

ICC enjoys two performance benefits. First, by design, the total number of transactions sent to the network is reduced and this relieves the network bottleneck. Second, average memory access latency reduces for requests that hit in the merge table. A request to an already outstanding request only sees the remaining access latency, which is (much) smaller compared to the latency of a newly initiated request.

B. Merge Table

The size of the merge table is likely to affect performance. The larger the size, the higher the opportunity to exploit intra-cluster locality. On the flip side, a larger merge table also implies higher hardware cost and access latency; access latency is something to consider since it is on the critical path for every L1 cache miss.

The maximum possible size of the merge table is determined by the maximum number of in-flight memory requests. Memory read requests in each SM first access the L1 cache, and in case of a cache hit, the data is sent to the register file. Otherwise, the memory request is sent to the next level of cache. In the L1 cache, the MSHRs track the in-flight L1 cache misses and merge duplicate requests accessing the same L2 cache lines. The number of MSHR entries controls the number of memory requests that can be injected into the NoC, i.e., when all MSHR entries are occupied, L1 cache misses can no longer be serviced. From this point of this view, the maximum size of the merge table is bounded by the number of SMs per cluster multiplied by the number of L1 MSHR entries per SM. This amounts to a maximum size of $5 \times 32 = 160$ entries for our clustered architecture.

Obviously, the size of the merge table can be set to a smaller value to reduce the hardware cost and/or access latency. This trade-off impacts our ability to coalesce memory requests across the NoC. We set the size of the merge table to 48 entries in our setup. We find that whereas a maximum sized merge table can coalesce 14.5% of the L1 cache misses, a 48-entry merge table captures the vast majority of those by coalescing 12% of the L1 cache misses.

C. Cost Analysis

In our setup, we assume a 48-entry fully-associative merge table. For GPU-compute applications with a 48-bit address space [11] and a 128-byte cache line size, we need 41 bits to record the address of the cache line. We further assume 5 bits to record the SM list, i.e., the SMs waiting for that particular cache line to come back from the memory subsystem. The total hardware cost amounts to 2,208 bits or 276 bytes per cluster. We use CACTI 6.5 [13] to compute the access latency of the merge table and we find it to be less than one cycle at 1.4 GHz assuming a 40 nm chip technology. This is also what we assume in our simulations, i.e., every L1 cache miss incurs an additional one-cycle latency for accessing the merge table.

V. CTA Scheduling versus ICC

Intra-cluster locality is not only a function of the algorithm or its implementation. It is also greatly affected by how CTAs are mapped to clusters. We consider four CTA scheduling algorithms here, and we illustrate them using the example shown in Figure 7. The example assumes 10 CTAs in total. We further assume 2 clusters with 2 SMs per cluster; each SM can execute two CTAs.

Two level round-robin follows the procedure previously described in Section II-C. CTAs are first distributed across clusters; once all clusters have one CTA assigned, we then assign CTAs across SMs within a cluster; finally, when all SMs across all clusters are assigned one CTA, we then assign additional CTAs per SM — the assignment of additional CTAs is done the same way. This CTA scheduling algorithm has the advantage of distributing the CTAs uniformly across all clusters and SMs in the system.

Global round-robin, or one-level round-robin, first distributes CTAs across all SMs within a cluster and then across clusters, i.e., it assigns a CTA to the first SM in the first cluster, then a

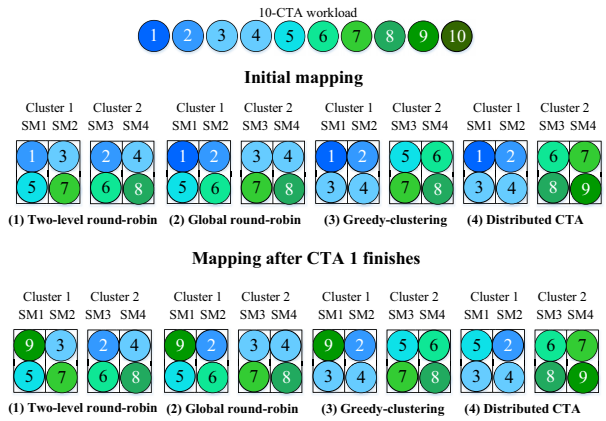

Fig. 7. Illustrating the four CTA scheduling algorithms for a 10-CTA workload. We assume a GPU architecture with 2 clusters with 2 SMs each; we can allocate 2 CTAs at most per SM. The top row shows the initial mapping of CTAs to clusters and SMs; the bottom row shows the mapping of the next CTA to schedule after CTA 1 finishes its execution.

second CTA is assigned to the second SM in the first cluster; once all SMs in a given cluster are assigned one CTA, we then move to the second cluster. Once all SMs across all clusters have one CTA assigned, we then assign additional CTAs to the SMs. The assignment of additional CTAs per SM is done in the same manner.

Greedy-clustering assigns as many CTAs as possibly to the first cluster before proceeding to the next, i.e., the first CTA is assigned to the first SM and the second CTA is assigned to the second SM in the first cluster; once all SMs in the cluster have one CTA assigned, additional CTAs are assigned to the cluster until all SMs can take no more additional CTAs. It then moves to the next cluster. This greedy-clustering algorithm has the advantage of fully utilizing the clusters and SMs that it uses. However, for kernels with a limited number of CTAs, this policy may lead to unbalanced execution, i.e., not all clusters are assigned the same workload. While this is not a concern for GPU-compute workloads that consist of a large number of CTAs, it may be problematic for others.

These three CTA scheduling policies share the common limitation that they expose limited intra-cluster locality. As mentioned before, inter-CTA locality typically occurs between neighboring CTAs. Compared to the other two policies, greedy-clustering may be advantageous because it assigns neighboring CTAs to the same cluster. The number of neighboring CTAs assigned to the same cluster under two-level round-robin and global round-robin is more limited. However, these three policies do not make any guarantees to exploit intra-cluster locality during the execution. In particular, when a CTA on an SM finishes execution, a new CTA needs to be launched and this is done without considering the locality between the new CTA and the CTAs already executing on the cluster.

Distributed CTA scheduling, proposed in MCM-GPU [8], addresses this issue by uniformly distributing CTAs across

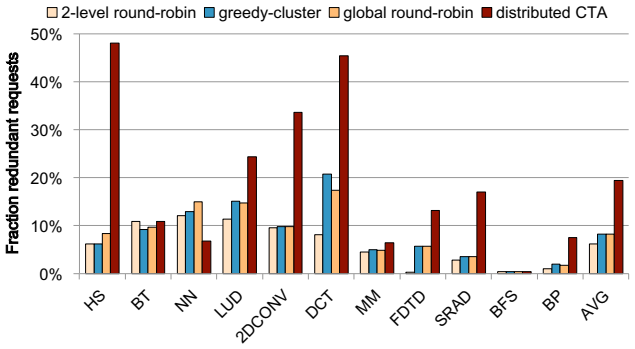

Fig. 8. Intra-cluster locality for the different CTA scheduling polices. *CTA scheduling policies have a substantial impact on the exploitable intra-cluster locality; distributed CTA scheduling yields the highest opportunity.*

clusters, i.e., all clusters get the same number of CTAs assigned in a pool of CTAs. In the example from Figure 7, there are 10 CTAs in total. Distributed CTA scheduling first splits up the set of CTAs evenly across the two clusters, i.e., CTAs 1 through 5 are assigned to cluster #1, and CTAs 6 through 10 are assigned to cluster #2. In the next step, it maps a block of neighboring CTAs to each cluster from the respective pools, i.e., CTAs 1 through 4 are mapped to cluster #1, and CTAs 6 through 9 are mapped to cluster #2. This is similar to greedy-clustering except that greedy-clustering does this from a global pool of CTAs whereas distributed CTA scheduling considers a per-cluster pool of CTAs. The key difference with the other CTA scheduling policies appears when a CTA finishes its execution, e.g., CTA 1 at the bottom in Figure 7. The two-level round-robin, global round-robin and greedy-clustering scheduling policies will then select and assign the next CTA from the global CTA pool, i.e., CTA 9 is selected and mapped to the cluster and SM where CTA 1 just finished its execution, namely SM#1 in cluster #1. Distributed CTA scheduling on the other hand selects the next CTA from the cluster's CTA pool to schedule, i.e., CTA 5 is mapped to cluster #1. This is a major difference because this enables distributed CTA scheduling to continuously optimize locality and assign neighboring CTAs to the same cluster during the entire execution.

Comparing CTA scheduling algorithms. Figure 8 quantifies intra-cluster locality, as previously defined in Section III-C, for the different CTA scheduling policies with a time window of 2000 cycles. Intra-cluster locality is the highest for distributed CTA scheduling. The reason is because distributed CTA scheduling maintains locality across neighboring CTAs, not only at the beginning of the execution, but also when new CTAs are launched.

Figure 9 reports performance (IPC) normalized to two-level round-robin. We observe that the distributed CTA scheduling policy significantly outperforms the other policies for a couple benchmarks. On average, the difference is modest. In the results section, we will report that ICC substantially improves performance for the distributed CTA scheduling policy, mak-

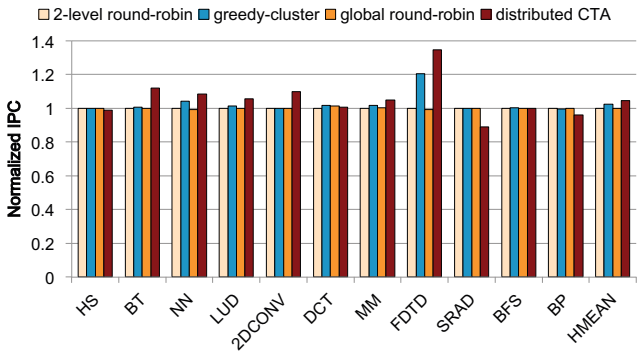

Fig. 9. Normalized IPC for the four CTA scheduling policies considered in this paper: two-level round-robin, greedy-clustering, global round-robin and distributed CTA scheduling. *Distributed CTA scheduling outperforms the other three policies on average.*

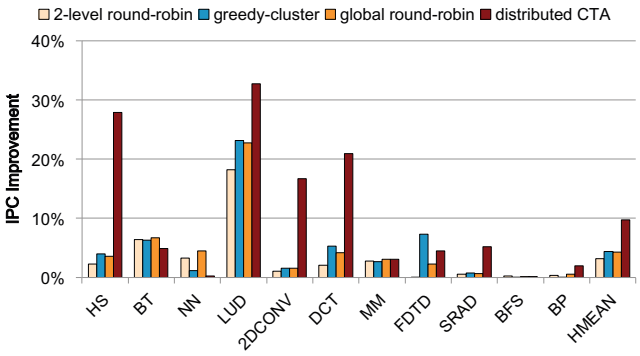

Fig. 10. IPC improvement for intra-cluster coalescing (ICC) for the four CTA scheduling policies considered in this paper: two-level round-robin, greedy-clustering, global round-robin and distributed CTA scheduling. *ICC is an effective optimization for all four CTA scheduling policies but the highest improvement is observed for distributed CTA scheduling.*

TABLE II
SIMULATED GPU CONFIGURATION.

Parameter	Value
Clock Frequency	1.4 GHz
Number of Clusters	12
Number of SMs per Cluster	5
Numbers of MC	8
Warp Schedulers / SM	2 (GTO)
L1 Cache / SM	48 KB 128 B line, 4-way assoc LRU, 32-entry MSHR
Shared Memory / SM	64 KB
L2 Unified Cache	512 KB per MC 128 B line, 8-way assoc LRU, 32-entry MSHR
NoC Topology	12 $\times$ 8 crossbar
NoC Channel width	64 B
NoC Bandwidth	716.8 GB/s
DRAM Bandwidth	720 GB/s
GDDR5 DRAM	1.4 GHz t_{CL}=12, t_{RP}=12, t_{RC}=40, t_{RAS}=28, t_{RCD}=12, t_{RRD}=6, t_{CCD}=2, t_{WR}=12

TABLE III
BENCHMARKS CONSIDERED IN THIS STUDY.

Benchmark	Suite	Abbr.
hotspot	Rodinia	HS
b+trees	Rodinia	BT
backprop	Rodinia	BP
bfs	Rodinia	BFS
srad	Rodinia	SRAD
lud	Rodinia	LUD
2Dconv	Polybench	2DCONV
matrixmul	SDK	MM
neuralnetwork	GPGPUsim	NN
FDTD3d	SDK	FDTD
dct8×8	SDK	DCT

ing it the winner across the board.

VI. EXPERIMENTAL SETUP

We faithfully model the proposed ICC unit in the GPGPU-Sim 3.2.2 simulator [14]. The merge table is set to hold up to 48 entries; we assume a one-cycle access latency to the merge table, which we account for in our simulations. We also model the four CTA scheduling algorithms: 2-level round-robin, global round-robin, greedy-cluster and distributed CTA scheduling. Table II shows the simulated GPU configuration. Our baseline includes intra-warp coalescing in which memory requests are coalesced across threads within a warp before sending them to the L1 cache [6]. We further assume 32 MSHR entries at both the L1 and L2 caches; the MSHRs at the L1 cache coalesce L1 misses within an SM.

Table III lists the workloads used to evaluate our proposed solution, and are taken from CUDA SDK [15], Rodinia [16] and PolyBench [17]; NN comes with GPGPUsim [14]. We choose a mix of high intra-cluster locality and low intra-cluster locality applications to properly evaluate the performance impact across a broad range of workloads.

VII. RESULTS

We now evaluate intra-cluster coalescing (ICC). This is done in a number of steps. We start by quantifying overall performance. We then investigate the main sources leading to the performance improvements. We finally provide a sensitivity analysis with respect to cluster size and the effective number of NoC ports per SM.

A. Overall Performance

Figure 10 reports overall performance (IPC) improvements through ICC for the four CTA scheduling policies considered in this paper. We observe modest improvements for ICC under two-level round-robin, greedy-clustering and global round-robin scheduling, i.e., performance improves by 3.2%, 4.4% and 4.3% on average, although LUD experiences a more substantial improvement by 18.2%, 23.2% and 22.8%, respectively.

Significantly higher performance improvements are observed for ICC under the distributed CTA scheduling policy, by 9.7% on average. Several benchmarks experience a substantial performance improvement, i.e., LUD (33%), HS (30%), 2DCONV (16.8%) and DCT (21%). The high performance

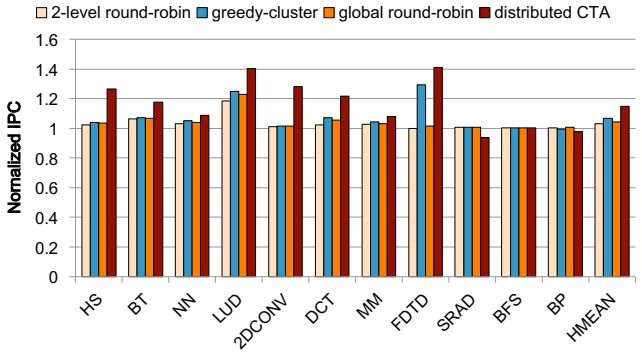

Fig. 11. IPC for the four CTA scheduling policies with ICC normalized to two-level round-robin scheduling without ICC. *Distributed CTA scheduling with ICC yields the best performance overall.*

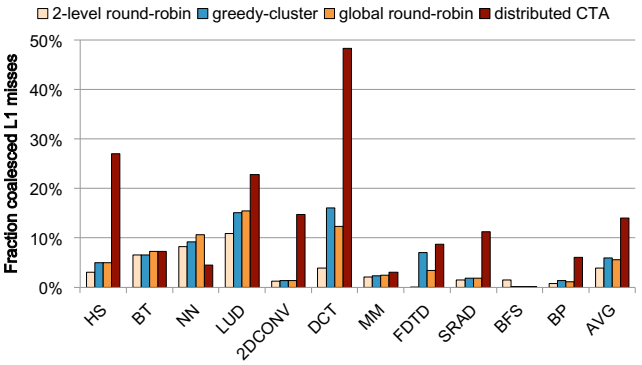

Fig. 12. Fraction coalesced L1 cache misses through ICC for the four CTA scheduling policies. *ICC coalesces a significant fraction of the L1 cache misses; this is especially the case for distributed CTA scheduling.*

achieved for ICC under distributed CTA scheduling, as compared to the alternative CTA scheduling policies, is due to the fact that the distributed scheme optimizes inter-CTA locality within a cluster during the entire execution. This fact creates more opportunities to apply intra-cluster coalescing, resulting in better performance.

Generally speaking, benchmarks with high intra-cluster locality, see Figure 8, benefit more from intra-cluster coalescing. However, the correlation is not perfect. This is due to the fact that intra-cluster locality quantifies the redundancy in read requests only. Applications that have a relatively high fraction of writes versus reads, e.g., DCT, do not benefit as much as the intra-cluster locality metric would suggest (although the improvement is still significant).

Figure 11 quantifies performance (IPC) for the four CTA scheduling algorithms *with* ICC, relative to two-level round robin without ICC. The key message is that distributed CTA scheduling with ICC is the overall winner. We report an average improvement by 15% and up to 40.8%. This is an important result because it shows that distributed CTA scheduling not only has the greatest opportunity for exploiting intra-cluster locality, as shown in Figure 10, it also yields the highest performance overall when deployed in conjunction with ICC, see Figure 11.

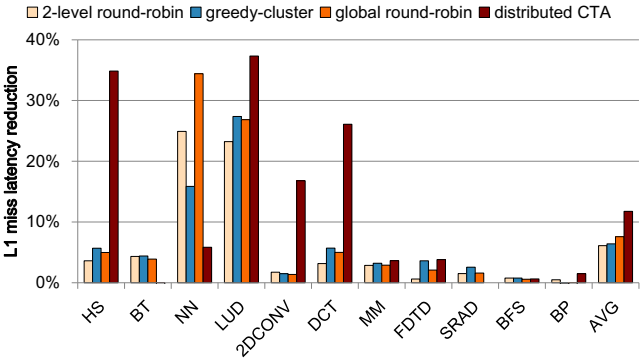

Fig. 13. Average normalized L1 cache miss latency reduction for ICC under the four CTA scheduling policies. *ICC reduces the average L1 cache miss latency significantly.*

B. L1 Cache Miss Coalescing

We next investigate where the performance improvements are coming from. To this end, we first quantify the fraction of L1 cache misses that get coalesced through ICC, see Figure 12. In line with the performance results just described, we observe a relatively modest fraction of coalesced L1 misses for two-level round-robin scheduling (4% on average), greedy-clustering (6% on average) and global round-robin CTA scheduling (5.5% on average). We obtain substantially better results under distributed CTA scheduling: 14% of the L1 cache misses get coalesced on average, and up to 48.3% (DCT), 27% (HS), 22.8% (LUD) and 14.7% (2DCONV). These are also the benchmarks for which we observed the highest performance improvement under ICC, see Figure 10. Coalescing L2 accesses reduces NoC pressure, which in turn leads to higher performance. Note the correlation is not perfect though — this is a result of whether the coalesced L1 cache misses are on the critical path and/or affect bandwidth saturation in the NoC and/or memory subsystem.

In contrast to what the intra-cluster locality metric reported in Figure 8 suggests, we observe that for some applications, e.g., HS and 2DCONV, ICC fails to coalesce a large fraction of the redundant accesses. This is due to the fact that the lifetime of an entry in the merge table is smaller than the 2000 cycles we assume for quantifying the amount of intra-cluster locality.

C. L1 Cache Miss Latency Reduction

Next, we investigate this further by quantifying the L1 cache miss latency. Coalescing L1 cache misses not only reduces NoC pressure, it also reduces the average L1 cache miss latency, i.e., a request to the cache line of an already outstanding cache line only sees the remaining latency, which reduces the average L1 cache miss latency.

The average L1 cache miss latency reduction is quantified in Figure 13. ICC, under the distributed CTA scheduling policy, reduces the average L1 cache miss latency by 11.7% on average. We observe good correlation with the fraction of coalesced L1 cache misses as shown in Figure 12. We observe the largest reduction in L1 cache miss latency for HS,

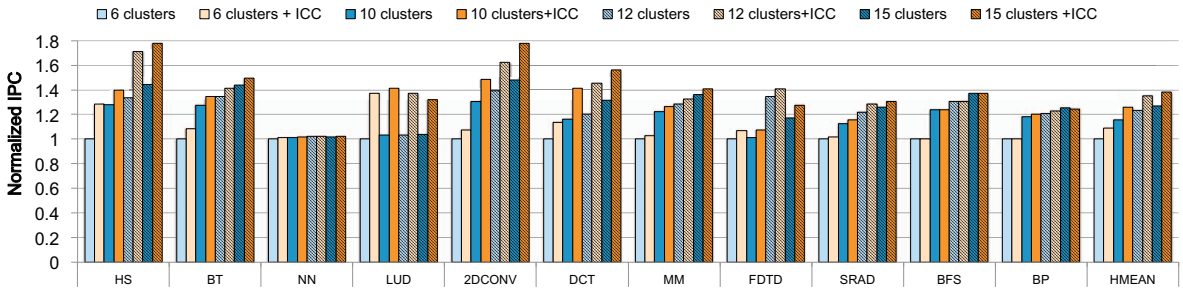

Fig. 14. Evaluating ICC while varying the number of clusters for a total of 60 SMs assuming distributed CTA scheduling; IPC is reported normalized to 6 clusters. *ICC consistently improves performance across different cluster sizes and across different effective NoC ports per SM.*

LUD, 2DCONV and DCT. These are also the benchmarks for which we observe the largest performance improvement. Interestingly, we obtain a large reduction in L1 cache miss latency for NN for most of the CTA scheduling policies, which does not seem to translate into a significant performance improvement. This is because NN has a relatively small L1 cache miss rate; hence, although we reduce the L1 cache miss latency significantly, the impact on performance is not as big.

D. Sensitivity Analysis

Our baseline configuration assumed 12 clusters with 5 SMs each and one NoC port per cluster. We now vary the number of SMs per cluster and include configurations with 6, 10, 12 and 15 clusters. To keep the total number of SMs constant at 60, each cluster consists of 10, 6, 5 and 4 SMs, respectively. We assume one NoC port per cluster, so the number of NoC ports per SM effectively increases as we increase the number of clusters.

Figure 14 reports normalized IPC for the four cluster configurations, assuming distributed CTA scheduling. The key message is that ICC is effective across different clustered GPU architecture configurations. Even with as little as 4 SMs per cluster sharing one NoC port (15 clusters in total), we still observe an average performance improvement of 9% (and up to 27.3%). We also observe the general trend that performance increases as we increase the number of clusters. This is a result of less NoC congestion, as there are more NoC ports and fewer SMs competing for NoC bandwidth. Yet, we do observe a significant performance improvement from intra-cluster coalescing even when the NoC is less congested.

VIII. Related Work

To the best of our knowledge, this is the first paper to target coalescing memory requests across SMs within a cluster to mitigate the NoC bottleneck in GPUs. We now discuss the most closely related work in CTA scheduling, inter-SM locality, GPU NoC optimization and memory access coalescing.

CTA scheduling. Several prior works exploit inter-CTA locality to improve CTA scheduling. In particular, Lee et al. [18] and Mao et al. [19] dispatch groups of two consecutive CTAs onto the same SM to improve L1 cache performance by exploiting locality between consecutive CTAs located in a row. Chen et al. [20] propose a software-hardware cooperative design to exploit spatial locality among different CTAs located in different rows and columns. Li et al. [21] propose software techniques to schedule CTAs with potential reuse on the same SM to exploit inter-CTA locality on real GPU hardware. None of these prior works explore CTA scheduling to improve intra-cluster coalescing opportunities [22], [23].

Exploiting inter-SM locality. A couple papers exploit inter-CTA locality. Tarjan and Skadron [24] propose a central sharing tracker (ST) to exploit data sharing among SMs. They consider a GPU architecture that lacks an on-chip last-level cache (LLC). Through the ST, L1 misses are sent to other SMs to obtain the data from another L1 cache (if available) instead of accessing off-chip main memory. Li et al. [25] prioritize memory requests to data that is shared across SMs. Neither of these approaches consider inter-CTA locality as a potential solution for the GPU NoC bottleneck in clustered GPUs.

GPU NoC optimization. Two recent works address the GPU NoC bottleneck by exploiting inter-SM locality. In particular, Zhao et al. [26] propose an inter-SM locality aware LLC design to transfer few-to-many NoC traffic into many-to-many traffic to increase the effective network bandwidth utilization. Kim et al. [27] exploit packet coalescing to reduce data redundancy in GPUs. These two prior works focus on a mesh NoC. Although the latter work also exploits packet coalescing, it coalesces redundant replies on each MC. This only alleviates the MC bottleneck but the traffic caused by a multicast operation to transfer the data back to the requesting SMs is not addressed, which may lead to serialization delays in the NoC routers. None of these prior works consider intra-cluster locality to reduce GPU NoC pressure.

Bakhoda et al. [3] propose a checkerboard router to reduce the NoC cost while providing multiple input ports for the MCs to increase the injection rate. The bandwidth-efficient NoC design by Jang et al. [28] leverages asymmetric virtual channel (VC) partitions to assign more VCs to reply packets which occupy a large portion of network traffic. Ziabari et al. [5] propose asymmetric NoCs where the reply network features high network bandwidth. Zhao et al. [29] propose a ring-like NoC to provide high bandwidth for reply packets in a cost-effective way. These previous works only focus on the NoC topology, but could be combined with our intra-cluster coalescing to further improve their performance.

Memory access coalescing. Intra-warp coalescing is widely deployed in GPUs to group aligned memory accesses of different threads in a warp [6]. To coalesce memory accesses from different warps, WarpPool [7] merges requests between warps within an SM to increase the effective L1 cache bandwidth. These prior works only target memory access coalescing within an SM. None of these notice and exploit the potential of coalescing duplicate memory accesses from different SMs within a cluster.

IX. Conclusion

Clustered GPUs face a severe NoC bottleneck with increasing SM count. To mitigate network congestion, we propose intra-cluster coalescing (ICC) by exploiting inter-CTA locality observed in many GPU-compute applications. ICC coalesces memory requests from different SMs in a cluster to the same L2 cache line to reduce the overall number of requests and replies sent over the NoC. We find that ICC coalesces 14% of all L1 cache misses on average (and up to 48.3%). This leads to an average 9.7% (and up to 33%) performance improvement over a set of benchmarks with varying degrees of inter-CTA locality. The overarching contribution of this paper is the exploitation of inter-CTA locality, an inherent GPU-compute workload characteristic, to tackle the emerging NoC congestion bottleneck in clustered GPUs to improve overall system performance by coalescing memory requests across SMs within a cluster.

X. Acknowledgements

We thank the anonymous reviewers for their valuable feedback. This work is supported through the European Research Council (ERC) Advanced Grant agreement No. 741097, Research Foundation Flanders (FWO) grants No. G.0434.16N and G.0144.17N, and the National Natural Science Foundation of China through grants No. 61572508 and 61672526.

References

[1] NVIDIA GP100 Pascal Architecture. NVIDIA Corporation. [Online]. Available: https://www.nvidia.com/object/pascal-architecture-whitepaper.html

[2] NVIDIA Tesla V100 Volta Architecture. NVIDIA Corporation. [Online]. Available: http://www.nvidia.com/object/volta-architecture-whitepaper.html

[3] A. Bakhoda, J. Kim, and T. M. Aamodt, "Throughput-Effective On-Chip Networks for Manycore Accelerators," in *Proceedings of the 43rd International Symposium on Microarchitecture (MICRO)*, Dec 2010.

[4] H. Kim, J. Kim, W. Seo, Y. Cho, and S. Ryu, "Providing Cost-Effective On-Chip Network Bandwidth in GPGPUs," in *Proceedings of International Conference on Computer Design (ICCD)*, Sept 2012.

[5] A. K. Ziabari, J. L. Abellán, Y. Ma, A. Joshi, and D. Kaeli, "Asymmetric NoC Architectures for GPU Systems," in *Proceedings of the International Symposium on Networks-on-Chip (NOCS)*, Sept 2015.

[6] J. Hestness, S. W. Keckler, and D. A. Wood, "A Comparative Analysis of Microarchitecture Effects on CPU and GPU Memory System Behavior," in *Proceedings of the International Symposium on Workload Characterization (IISWC)*, Oct 2014.

[7] J. Kloosterman, J. Beaumont, M. Wollman, A. Sethia, R. Dreslinski, T. Mudge, and S. Mahlke, "WarpPool: Sharing Requests with Inter-Warp Coalescing for Throughput Processors," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, Dec 2015.

[8] A. Arunkumar, E. Bolotin, B. Cho, U. Milic, E. Ebrahimi, O. Villa, A. Jaleel, C.-J. Wu, and D. Nellans, "MCM-GPU: Multi-Chip-Module GPUs for Continued Performance Scalability," in *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, Jun 2017.

[9] O. Kayıran, A. Jog, M. T. Kandemir, and C. R. Das, "Neither More nor Less: Optimizing Thread-level Parallelism for GPGPUs," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sept 2013.

[10] A. Lopes, F. Pratas, L. Sousa, and A. Ilic, "Exploring GPU Performance, Power and Energy-Efficiency Bounds with Cache-Aware Roofline Modeling," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2017.

[11] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-Conscious Wavefront Scheduling," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, Dec 2012.

[12] D. Kroft, "Lockup-free Instruction Fetch/Prefetch Cache Organization," in *Proceedings of the Annual Symposium on Computer Architecture (ISCA)*, May 1981.

[13] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "CACTI 6.0: A Tool to Model Large Caches," *HP Laboratories*, 2009.

[14] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Apr 2009.

[15] NVIDIA CUDA SDK Code Samples. NVIDIA Corporation. [Online]. Available: https://developer.nvidia.com/cuda-downloads

[16] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *Proceedings of the International Symposium on Workload Characterization (IISWC)*, Oct 2009.

[17] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a High-Level Language Targeted to GPU Codes," in *Innovative Parallel Computing (InPar)*, May 2012.

[18] M. Lee, S. Song, J. Moon, J. Kim, W. Seo, Y. Cho, and S. Ryu, "Improving GPGPU Resource Utilization Through Alternative Thread Block Scheduling," in *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2014.

[19] M. Mao, J. Hu, Y. Chen, and H. Li, "VWS: A Versatile Warp Scheduler for Exploring Diverse Cache Localities of GPGPU Applications," in *Proceedings of the Design Automation Conference (DAC)*, June 2015.

[20] L. J. Chen, H. Y. Cheng, P. H. Wang, and C. L. Yang, "Improving GPGPU Performance via Cache Locality Aware Thread Block Scheduling," *IEEE Computer Architecture Letters*, 2017.

[21] A. Li, S. L. Song, W. Liu, X. Liu, A. Kumar, and H. Corporaal, "Locality-Aware CTA Clustering for Modern GPUs," in *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Apr 2017.

[22] Y. Wang, D. Wang, S. Chen, Z. Liu, S. Chen, X. Chen, and X. Zhou, "Iteration Interleaving-Based SIMD Lane Partition," *TACO*, vol. 12, no. 4, 2016.

[23] C. Li, S. Ma, S. Chen, Y. Guo, and P. Wang, "Express Ring: A Multi-Layer and Non-Blocking NoC Architecture," *IEICE Electronic Express*, vol. 12, no. 3, 2015.

[24] D. Tarjan and K. Skadron, "The Sharing Tracker: Using Ideas from Cache Coherence Hardware to Reduce Off-Chip Memory Traffic with Non-Coherent Caches," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, Nov 2010.

[25] D. Li and T. M. Aamodt, "Inter-Core Locality Aware Memory Scheduling," *IEEE Computer Architecture Letters*, vol. 15, no. 1, pp. 25–28, Jan 2016.

[26] X. Zhao, Y. Liu, A. Adileh, and L. Eeckhout, "LA-LLC: Inter-Core Locality-Aware Last-Level Cache to Exploit Many-to-Many Traffic in GPGPUs," *IEEE Computer Architecture Letters*, vol. 16, no. 1, Jan 2017.

[27] K. H. Kim, R. Boyapati, J. Huang, Y. Jin, K. H. Yum, and E. J. Kim, "Packet Coalescing Exploiting Data Redundancy in GPGPU Architectures," in *Proceedings of the International Conference on Supercomputing (ICS)*, June 2017.

[28] H. Jang, J. Kim, P. Gratz, K. H. Yum, and E. J. Kim, "Bandwidth-Efficient On-Chip Interconnect Designs for GPGPUs," in *Proceedings of the Design Automation Conference (DAC)*, June 2015.

[29] X. Zhao, S. Ma, C. Li, L. Eeckhout, and Z. Wang, "A Heterogeneous Low-Cost and Low-Latency Ring-Chain Network for GPGPUs," in *Proceedings of the International Conference on Computer Design (ICCD)*, Oct 2016.