Intra-Cluster Coalescing and Distributed-Block Scheduling to Reduce GPU NoC Pressure

Lu Wang, *Student Member, IEEE,* Xia Zhao, *Student Member, IEEE,* David Kaeli *Fellow, IEEE,* Zhiying Wang *Member, IEEE,* and Lieven Eeckhout *Fellow, IEEE*

Abstract—GPUs continue to boost 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. Sending redundant requests to access the same memory location wastes valuable NoC bandwidth — we find on average 19% (and up to 48%) of the requests to be redundant. To remove redundant NoC traffic, we propose distributed-block scheduling, intra-cluster coalescing (ICC) and the coalesced cache (CC) to coalesce L1 cache misses within and across SMs in a cluster, respectively. Our evaluation results show that distributed-block scheduling, ICC and CC are complementary and improve both performance and energy consumption. We report an average performance improvement of 15% (and up to 67%) while at the same time reducing system energy by 6% (and up to 19%) and improving the energy-delay product (EDP) by 19% on average (and up to 53%), compared to state-of-the-art distributed CTA scheduling.

Index Terms—GPU, coalescing, CTA scheduling, inter-CTA locality, NoC pressure

1 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 latest Nvidia Pascal [1] and 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 NoC in a GPU due to its low latency and high bandwidth [1]. However, a crossbar NoC faces scalability issues as hardware costs increase quadratically with 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 with 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.

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 executing on the same cluster will access the same cache lines. According to our experimental results, we find that on average 19% (and up to 48%) of all L1 misses originating from a cluster indeed access the same cache lines. These memory requests are redundant and can be eliminated.

Motivated by this observation, we propose *distributed-block scheduling*. In contrast to prior work in CTA scheduling, distributed-block scheduling exploits cache locality at both the cluster level *and* the SM level. At the cluster level,

This journal submission extends the conference paper 'Intra-Cluster Coalescing to Reduce GPU NoC Pressure' by the same authors published at the 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS). This manuscript makes two additional contributions over the conference paper: (i) proposal of distributed-block scheduling, (ii) proposal of the coalesced cache, and (iii) evaluation of energy efficiency.

- *L. Wang, X. Zhao, L. Eeckhout are with the Department of Electronics and Information Systems (ELIS), Ghent University, Gent, 9000, Belgium. E-mail: {luluwang.wang, xia.zhao, lieven.eeckhout}@ugent.be*
- *Z. Wang is with the School of Computer, National University of Defense Technology, ChangSha, 410073, Hunan, P.R. China E-mail: zywang@nudt.edu.cn*
- *D. Kaeli is with the Department of Electrical and Computer Engineering, Northeastern University, 333 Dana Research Center, Boston, MA, USA. E-mail: kaeli@ece.neu.edu*

consecutive CTAs are mapped to the same cluster, similar to distributed CTA scheduling [6]. At the SM level, CTAs are allocated in groups of two to further exploit L1 cache locality, similar to block scheduling [7]. This two-level approach maximizes the opportunities to exploit inter-CTA locality among CTAs at the L1 cache within an SM by first mapping a group of consecutive CTAs at the cluster level, and subsequently mapping pairs of consecutive CTAs at the SM level. Inter-CTA locality is exploited to improve L1 cache performance (decreasing the reuse distance between accesses to the same memory location, thereby increasing L1 cache hit rate) and to increase the coalescing opportunities in the L1 miss status handling registers (MSHRs).

Although distributed-block scheduling reduces the number of redundant requests at the SM level, it does not tackle the redundant requests originating from different SMs within a cluster. We therefore propose *intra-cluster coalescing (ICC)* to exploit coalescing opportunities across SMs within a cluster. ICC reduces GPU NoC pressure by coalescing memory requests from different SMs in a cluster to the same L2 cache line. In particular, ICC records the memory requests sent to the NoC by all SMs in a cluster, and when subsequent memory requests from other SMs in the cluster access the same cache lines as an outstanding request, ICC coalesces them. By doing so, ICC significantly reduces NoC traffic. To extend the opportunity for coalescing beyond the time window during which a memory request is outstanding, we complement ICC with a *coalesced cache (CC)* to keep track of recently coalesced cache lines. Cache lines are added to the CC when a coalesced cache line with multiple requesters returns from the memory hierarchy. L1 cache misses trigger an access to the CC, which in case of a hit, further reduces NoC traffic.

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 and is an effective technique to reduce NoC pressure. More specifically, intra-warp coalescing merges L1 cache accesses across threads within a warp [8]; WarpPool merges L1 accesses across warps within the same SM [9]; L1 Miss Status Handling Registers (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.

Distributed-block scheduling, intra-cluster coalescing and the coalesced cache operate synergistically and significantly reduce NoC pressure, which improves performance while at the same time reducing energy consumption. We report that distributed-block scheduling by itself improves performance by 4% on average (up to 16%) and reduces system energy by 1.2% (up to 6.5%) over state-of-the-art distributed CTA scheduling [6]. Intra-cluster coalescing further improves performance and energy efficiency, leading to an average performance improvement of 10% (and up to 28%), an average reduction in system energy by 4% (and up to 9%) and improved energy-delay product (EDP) by 13% (and up to 26%). Finally, the coalesced cache yet further improves performance and energy efficiency, to an overall average 15% performance improvement (up to 67%), an average 6% reduction in energy consumption (up to 19%) and an average 19% improvement in EDP (and up to 53%).

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.
- We propose *distributed-block scheduling* to exploit inter-CTA locality at the SM level by mapping groups of consecutive CTAs at the cluster level and then pairs of consecutive CTAs at the SM level.
- We propose *intra-cluster coalescing (ICC)* and the *coalesced cache (CC)* to track and coalesce L1 cache misses from different SMs in a cluster before sending them across the NoC.
- We study the complementarity and interaction between CTA scheduling, ICC and CC as a solution to reduce GPU NoC pressure.
- We comprehensively evaluate our newly proposed distributed-block scheduling and ICC scheme, and report an average 15% (up to 67%) performance improvement while simultaneously reducing system energy by 6% (up to 19%) and EDP by 19% (and up to 53%) over the state-of-the-art distributed CTA scheduling policy. The hardware cost of the ICC unit is limited to 276 bytes per cluster.

2 Background

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

2.1 GPU Thread Hierarchy

Using Nvidia's terminology, a GPU-compute application consists of kernels, grids, CTAs, warps and threads, and they are organized in a hierarchy. A kernel is a parallel code region that runs on a GPU and consists of multiple grids, which in turn consist of multiple CTAs. Each CTA is a batch of threads that can coordinate with each other through synchronization using a barrier instruction [10]. 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.

2.2 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; the SMs are connected through a bus within a cluster. (In the evaluation section, we will study the sensitivity of our design to the number of clusters and the network ports per SM.) Each cluster has

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2019.2895036, IEEE Transactions on Computers

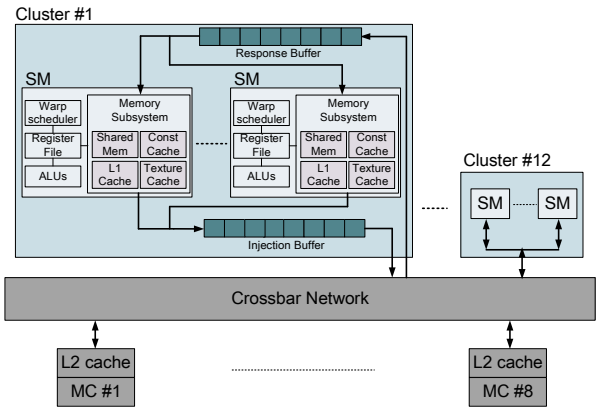

Fig. 1. Clustered GPU architecture: *SMs within a cluster go through the NoC to access the L2 cache and main memory to serve L1 cache misses.*

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.

2.3 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 [11], 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 using a Greedy-Then-Oldest (GTO) policy [12].

3 Motivation and Opportunity

We now further motivate the problem and describe the opportunity.

3.1 NoC Bandwidth Bottleneck

We first demonstrate that the NoC indeed constitutes a performance bottleneck in a clustered GPU architecture. In particular, we study the relationship between performance and NoC bandwidth. Figure 2 quantifies performance as a function of NoC bandwidth. To ensure an overall balanced design, we vary the LLC bandwidth proportionally as we change the NoC bandwidth. This is done by increasing the clock frequency of the NoC and LLC subsystems by the same factor. This enables providing 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 7.) We find that performance is sensitive to NoC and LLC bandwidth for most of the benchmarks. In particular, increasing NoC/LLC bandwidth by a factor 1.5× leads to a substantial performance benefit; doubling the NoC/LLC bandwidth saturates the performance improvement to on average 45% and up to 78%. This illustrates that NoC bandwidth indeed is a severe bottleneck. Limited NoC bandwidth leads to congestion within a cluster for memory requests that need to proceed through the NoC to reach the L2 cache and beyond.

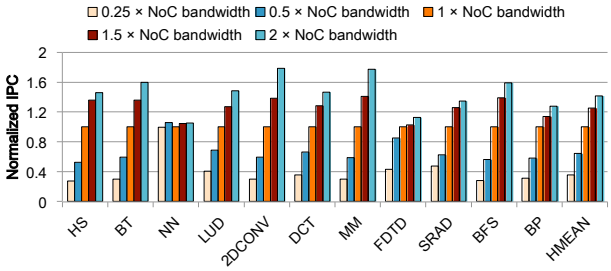

Fig. 2. Quantifying the NoC bottleneck: Normalized IPC when varying the NoC and LLC frequency from $0.25\times$ to $2\times$. *NoC (and LLC) bandwidth is a severe performance bottleneck.*

3.2 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 1 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 [8]. 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. [9] 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.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2019.2895036, IEEE Transactions on Computers

TABLE 1
GPU coalescing techniques and their scope.

Technique	Scope
Intra-warp coalescing [8]	Across threads in a warp
WarpPool [9]	L1 accesses across warps in an SM
L1 MSHR [13]	L1 misses across warps in an SM
Packet coalescing [14]	MC side
ICC (this work)	L1 misses across SMs in a cluster

Miss Status Handling Registers (MSHRs) [13] 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.

Packet coalescing [14] groups memory requests from different SMs at the memory controller (MC) side. The MC then generates a single read reply and relies on a multicast NoC to transfer the reply packet to the requesting SMs. Packet coalescing does not reduce the number of L1 miss requests sent over the NoC.

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.

3.3 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.

3.3.1 Quantifying Intra-Cluster Locality

To quantify the amount of intra-cluster locality, we 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 versus 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.

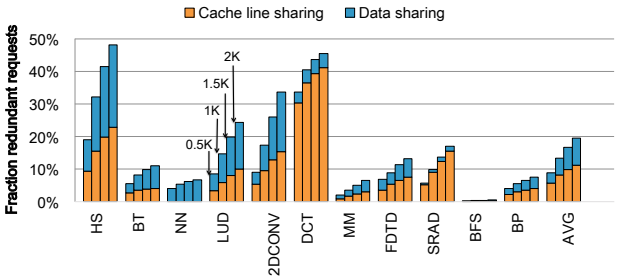

Fig. 3. Intra-cluster locality (fraction of redundant requests versus the total number of requests in a cluster) as a function of a past window of requests under the distributed CTA scheduling policy [6]. A distinction is made between cache line sharing and data sharing. *A substantial fraction of NoC requests are redundant because of intra-cluster locality due to cache line sharing or data sharing.*

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

3.3.2 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 analyze all the benchmarks and identify two categories of inter-CTA locality: cache line sharing versus data sharing. Figure 3 quantifies their relative contribution. For a window size of 2000 cycles, we observe 19.4% intra-cluster locality, with 11.2% due to cache line sharing and 8.2% due to data sharing. We also note that intra-cluster locality increases with increasing window size.

(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 — spatial locality. 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.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2019.2895036, IEEE Transactions on Computers


```
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 from 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).*

(2) Inter-CTA locality due to data sharing. In many GPU-compute applications, we observe that different CTAs access the *same* (read-only) data – temporal locality. 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 of a 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 the submatrices. For example, in the first iteration, submatrices L_{j1} and U_{1i} are 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).

We note that data sharing may happen between CTAs that are relatively far apart from each other. For example, LUD features a 6×6 CTA organization in which CTAs in the same row and same column access the same data. Hence, CTAs that are multiples of 6 away from each other will access the same data, i.e., data sharing within a column. During our workload analysis, we find that locality due to data sharing may happen among CTAs that are relatively far apart. On the other hand, inter-CTA locality due to cache line sharing is typically observed among adjacent CTAs.

4 CTA Scheduling

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. In order to illustrate this point, we consider four previously proposed CTA scheduling policies, which we illustrate using the example shown in Figure 6. The example assumes 10 CTAs in total; we further assume two clusters with two SMs per cluster; each SM can execute two CTAs.

4.1 Scheduling Algorithms

Two-level round-robin scheduling follows the strategy previously described in Section 2.3. CTAs are first distributed across clusters; once all clusters have one CTA assigned, we assign CTAs across SMs within a cluster; finally, when all SMs in all clusters are assigned one CTA, we 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 scheduling, first distributes CTAs across all SMs within a cluster and then across clusters, i.e., it assigns a CTA to the first SM and a second CTA to the second SM in the first cluster; once all SMs in the first cluster are assigned one CTA, we move to the second cluster, and so forth. Once all SMs in 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 possible 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 allocated SMs and clusters. However, for kernels with a limited number of CTAs, this policy may lead to imbalanced 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 scheduling is more limited. However, these three polices 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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2019.2895036, IEEE Transactions on Computers

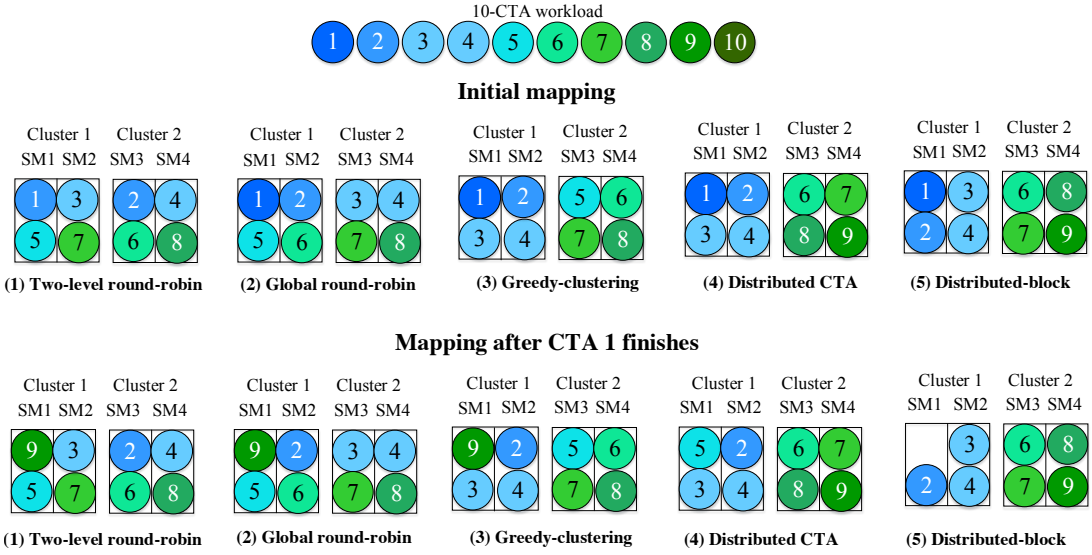

Fig. 6. Illustrating the five CTA scheduling algorithms for a 10-CTA workload. We assume a GPU architecture with 2 clusters with 2 SMs each; we can allocate at most 2 CTAs per SM. The top row shows the initial mapping of CTAs to clusters and SMs; the bottom row shows the mapping of the next CTA to schedule after CTA 1 finishes its execution.

the CTAs already executing on the cluster. This is, when CTA 1 finishes in the example shown in Figure 6, CTA 9 gets scheduled and assigned to the SM previously executing CTA 1. Unfortunately, there may be limited or no inter-CTA locality between CTA 9 and the CTAs already running on the same cluster.

Distributed scheduling as proposed in MCM-GPU [6], addresses this issue by uniformly distributing CTAs across clusters, i.e., all clusters get the same number of CTAs assigned in a pool of CTAs. In the example in Figure 6, there are 10 CTAs in total. Distributed 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 6. As mentioned above, two-level round-robin, global round-robin and greedy-clustering scheduling select and assign the next CTA from the global CTA pool, i.e., CTA 9 is selected and mapped to the SM and cluster where CTA 1 just finished its execution, namely the first SM in cluster #1. Distributed scheduling on the other hand selects the next CTA from the cluster's CTA pool, i.e., CTA 5 is mapped to cluster #1. This is a major difference as it enables distributed CTA scheduling to continuously exploit inter-CTA locality and assign neighboring CTAs to the same cluster during the entire execution.

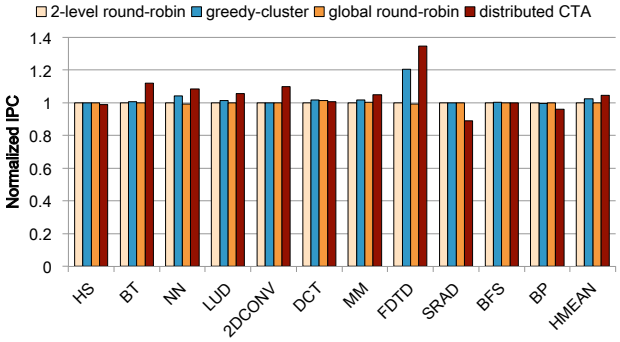

Fig. 7. Normalized IPC for two-level round-robin, greedy-clustering, global round-robin and distributed CTA scheduling. *Distributed scheduling outperforms the other three policies on average.*

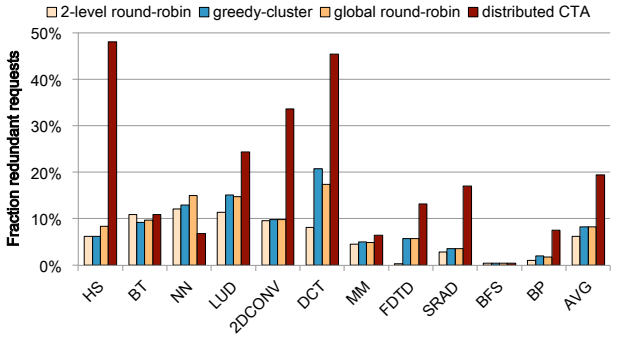

Fig. 8. Intra-cluster locality for the different CTA scheduling policies. *CTA scheduling policies have a substantial impact on the exploitable intra-cluster locality, and distributed scheduling yields the highest opportunity.*

4.2 Performance Analysis

Figure 7 reports performance (IPC) normalized to two-level round-robin. We observe that distributed scheduling is the best performing scheduling policy. Significantly higher performance is achieved for FDTD (35%) and a couple other benchmarks including BT (12%), 2DCONV (10%), NN (9%)

and LUD (6%). The reason for the higher performance is improved L1 cache locality. Distributed scheduling achieves lower performance than the other policies for SRAD (11%) and BP (4%) because of workload imbalance across SMs. On average, we report that distributed scheduling is the best performing policy, i.e., it achieves 6% higher performance on average compared to two-level round-robin scheduling. Because it is the best performing policy, we will consider distributed scheduling as the default CTA scheduling policy for the remainder of the paper.

The comparison becomes even more interesting as we consider intra-cluster locality, see Figure 8 which quantifies intra-cluster locality as previously defined in Section 3.3 for the different CTA scheduling policies with a time window of 2000 cycles. Intra-cluster locality is the highest for distributed scheduling, i.e., we measure that on average 19.5% (and up to 48% for HS and 46% for DCT) of the requests within a cluster are redundant. The reason is that distributed scheduling maintains inter-CTA locality across consecutive CTAs, not only at the beginning of the execution but also during the execution as CTAs finish and new CTAs get launched. This makes distributed scheduling particularly amenable to intra-cluster coalescing, and in addition, reinforces the choice to use distributed scheduling as our baseline.

There are two possibilities to exploit the observed intra-cluster locality. The first approach is based on the observation that a fraction of the locality across SMs within a cluster can be captured within an SM by changing the CTA scheduling policy. This is the approach taken through the newly proposed distributed-block scheduling policy, which we discuss in Section 5. The second approach is to coalesce requests to the same cache lines across SMs in a cluster, which is the approach taken in the newly proposed intra-cluster coalescing mechanism as discussed in Section 6.

5 Distributed-Block Scheduling

Among the state-of-the-art CTA scheduling strategies, distributed scheduling clearly exposes the most intra-cluster locality. It does so by uniformly distributing CTAs across clusters. However, CTAs are allocated following a (default) round-robin strategy within a cluster. As a result, consecutive CTAs may be allocated to different SMs in a cluster, which may lead to reduced L1 cache performance and/or missed opportunities to coalesce requests at the L1 cache MSHRs. We therefore propose *two-level distributed-block CTA scheduling*, or *distributed-block scheduling* for short, to exploit coalescing opportunities at the cluster level and at the same time increase locality benefits at the L1 cache level. At the cluster level, we use distributed scheduling to maximize intra-cluster locality. At the SM level, we leverage block CTA scheduling (BCS) [7] which assigns a block of two CTAs to the same SM. The intuition is to increase the opportunity for exploiting data cache line locality across CTAs if those CTAs get allocated to the same SM at the same time. BCS delays the scheduling of CTAs to an SM until there are two CTA contexts available on the SM to simultaneously schedule two CTAs. This has two potential benefits: (i) the L1 cache hit rate improves because of improved locality, and (ii) there are more coalescing opportunities in the L1 cache MSHRs because the reuse distance between two accesses to the same memory location is reduced. In other words, inter-CTA locality gets exploited within a single SM, which leads to overall higher performance. Note though that delayed CTA scheduling may also incur some performance overhead, i.e., if an SM only has one CTA context left, no new CTA can be allocated until another CTA context becomes available, which leaves SM resources underutilized.

Distributed-block scheduling is illustrated in Figure 6. At the cluster level, distributed-block scheduling performs the same as distributed scheduling. We first split the CTAs evenly and assign the first 5 CTAs to cluster #1 and the next five CTAs to cluster #2. In the next step, rather than allocating CTAs by using a default round-robin strategy, we allocate CTAs at a granularity of 2 to different SMs in a cluster. CTAs 1 and 2 are allocated to the first SM in cluster 1; CTAs 3 and 4 are allocated to the second SM in cluster 1. By doing so, CTAs with higher inter-CTA locality are likely to be allocated to the same SM. Note that distributed-block scheduling follows a delayed mapping strategy once a CTA finishes its execution, as illustrated at the bottom part in Figure 6. New CTAs can only be allocated to an SM if at least two CTA contexts are available. Hence, CTA 5 can be allocated to SM #1 only when both CTAs 1 and 2 have finished their execution.

6 Intra-Cluster Coalescing (ICC)

Distributed-block scheduling exploits inter-CTA locality between adjacent CTAs within the same SM. However, it does not exploit inter-CTA locality across SMs in a cluster. This leaves performance and efficiency benefits on the table, i.e., a significant fraction of data sharing within a cluster may not come from adjacent CTAs running on the same SM. Hence, we propose *intra-cluster coalescing (ICC)* to coalesce inter-CTA locality across SMs within the same cluster. The key idea of ICC is to merge requests from different SMs in a cluster to the same L2 cache line before sending the request to the NoC. By doing so, ICC decreases the number of memory transactions over the network and reduces the contention on the network port shared by multiple SMs in a cluster.

6.1 ICC Unit

Figure 9 illustrates the overall architecture of the proposed 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. 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 (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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2019.2895036, IEEE Transactions on Computers

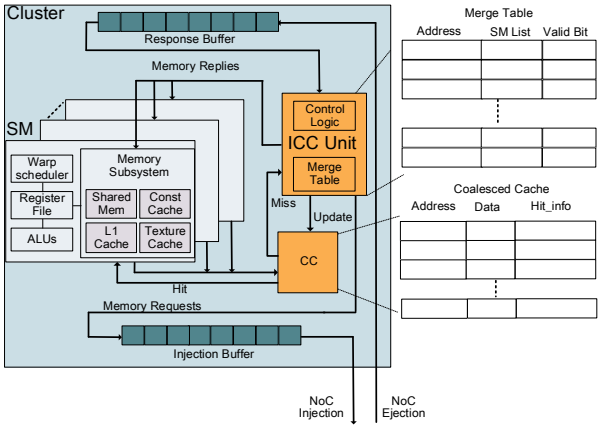

Fig. 9. The intra-cluster coalescing (ICC) unit merges L1 cache misses across SMs within a cluster. The coalesced cache (CC) keeps track of recently coalesced cache lines.

line. If on the other hand, there is no entry allocated in the merge table for that cache line (merge table miss), the ICC unit 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. ICC unit only records read requests to the global address space; read requests to other address spaces bypass the merge table. Write requests also bypass the merge table in order not to complicate the design of the merge table — including writes in the merge table would require storing entire cache lines and implementing write merging within a cache line.

When a cluster receives a reply packet from the network, the ICC unit uses the reply address to index the merge table. If there exists an entry for that address (a merge table hit), the ICC unit reads the corresponding SM list and broadcasts 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, the average memory access latency is reduced 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.

6.2 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 large 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. In case of a miss, 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 line. 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. Hence, 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.

6.3 Coalesced Cache

A limitation of the ICC unit as just proposed is that it can only coalesce memory requests within a limited time window, namely while the initial memory request is outstanding. However, as quantified in Section 3.3, there exists significant inter-CTA locality within large time windows, beyond the latency of a memory request. In other words, there is a high possibility that coalesced cache lines will be accessed again in the near future. We therefore extend the ICC unit with a *coalesced cache* to keep track of recently coalesced cache lines.

Figure 9 illustrates the architecture of the coalesced cache (CC). The CC is accessed upon an L1 cache miss. In case of a hit in the CC, i.e., this is an access to a previously coalesced cache line, the cache line is simply returned, saving a request over the NoC to the next level of cache. In case of a CC miss, the cache line is inserted in the merge table as previously described. Cache lines are added to the CC upon their return from the memory hierarchy *if* there are more than a single requester, i.e., the cache line is only inserted in the CC if it is effectively a coalesced cache line as indicated in the respective entry in the merge table.

6.4 Cost Analysis

As mentioned before, we assume a 48-entry fully-associative merge table. For GPU-compute applications with a 48-bit address space [12] 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 for the merge table amounts to 2,208 bits or 276 bytes. We further assume a 24-entry fully-associative coalescing cache with 41 bit tags and 128 byte cache lines, amounting to a total size of 3.2 KB. (We find that a larger number of entries in the coalesced cache improves performance but we opt for 24 entries in the evaluation to balance performance and hardware cost.)

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2019.2895036, IEEE Transactions on Computers

TABLE 2
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 × 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 3
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

We need a merge table and coalesced cache for each cluster. We use CACTI 6.5 [15] to compute the access latency to the merge table and coalesced cache, and we find it to be less than one cycle at 1.4GHz assuming a 40nm chip technology.

7 Experimental Setup

The evaluation is done using the GPGPU-Sim 3.2.2 simulator [16]. Table 2 shows the simulated baseline GPU configuration. We assume a total of 12 clusters with each cluster containing 5 SMs; hence, there are 60 SMs in total. Each SM features a 48 KB L1 cache. The 12 clusters are connected through a crossbar NoC to 8 memory controllers with a 512 KB L2 cache per memory controller. (We will vary the number of clusters and the number of SMs per cluster in the evaluation.) We further assume Greedy-Then-Oldest (GTO) [12] as the warp scheduling policy within an SM. The merge table in the ICC unit and coalescing cache are configured to hold up to 48 entries and 24 entries respectively; we assume a one-cycle access latency to the merge table and coalescing cache, which we account for in our simulations. We also model the five CTA scheduling algorithms: 2-level round-robin, global round-robin, greedy-clustering, distributed scheduling and the newly proposed

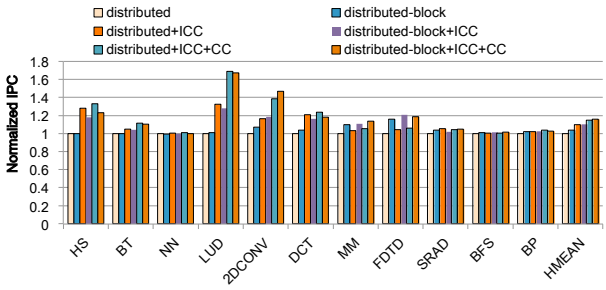

Fig. 10. Normalized IPC for distributed-block scheduling ,intra-cluster coalescing (ICC) and coalesced cache (CC) normalized to distributed scheduling. *Distributed-block scheduling improves performance by 4% on average; ICC significantly improves performance for both CTA scheduling policies and yields an overall average 10% performance improvement; ICC along with CC yields an 15% performance improvement.*

distributed-block scheduling. Our baseline includes intra-warp coalescing in which memory requests are coalesced across threads within a warp before sending them to the L1 cache [8]. We further assume 32 MSHR entries at both the L1 and L2 caches; the MSHRs at the L1 coalesce L1 misses within an SM.

When evaluating GPU energy consumption, we use GPUWattch [17] assuming a 40 nm chip technology. GPUWattch is modified and configured to model the same GPU configuration as the performance simulator. We account for the extra energy consumed in the merge table and coalesced cache, although we find it to be negligible.

Table 3 lists the workloads used to evaluate our proposed solution, taken from CUDA SDK [18], Rodinia [19] and PolyBench [20]; NN comes with GPGPUsim [16]. 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.

8 Results

We now evaluate distributed-block scheduling and intra-cluster coalescing. This is done in a number of steps. We start by investigating the performance improvement and energy consumption reduction, after which we analyze the impact on NoC traffic and memory access latency. Finally, we provide a sensitivity analysis with respect to cluster size and the effective number of NoC ports per SM.

8.1 Performance

Figure 10 reports normalized performance for distributed-block scheduling, intra-cluster coalescing (ICC) and the coalesced cache (CC) normalized to distributed distributed scheduling (without ICC). A couple of interesting observations can be drawn from these results. Distributed-block scheduling outperforms distributed scheduling for all benchmarks, and by 4% on average. For applications that exhibit L1 cache locality such as FDTD, MM and 2DCONV, distributed-block scheduling improves performance by 15%, 10% and 7%, respectively. This is because part of the inter-CTA locality is captured at the L1 cache within an SM. However, for applications such as HS and DCT which exhibit high intra-cluster locality, distributed-block scheduling does not show any IPC improvement. In

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2019.2895036, IEEE Transactions on Computers

HS, the lack of improvement is due to the high L1 cache miss rate, which under the GTO warp scheduling policy leads to cache lines being replaced before being referenced again. Hence, even though two CTAs with cache-line related locality are allocated to the same SM, this does not result in improved L1 cache performance. The reason is different for DCT: this benchmark contains more writes than reads. As a result, reducing the number of L1 read misses has limited impact on overall performance.

Distributed-block scheduling works synergistically with ICC to improve performance by 10% on average. Intra-cluster coalescing leads to an additional performance improvement of 6% on average over distributed-block scheduling. Several benchmarks experience a substantial performance improvement when combining distributed-block scheduling with ICC, see for example LUD (28%), HS (19%), DCT (17%) and 2DCONV (19%). Generally speaking, benchmarks with high intra-cluster locality, see Figure 3, benefit more from ICC. 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). Overall, we report that distributed-block scheduling with ICC improves performance by 10% on average (and up to 28%) compared to state-of-the-art distributed scheduling.

Intra-cluster coalescing improves distributed scheduling performance by 10% on average. For some benchmarks we do observe that distributed scheduling achieves higher performance than distributed-block scheduling, assuming both are complemented with ICC, see for example HS, LUD and DCT. This is due to delayed CTA launch under block scheduling, i.e., a new CTA is only scheduled when two CTA slots are available. For other benchmarks we observe that distributed-block scheduling outperforms distributed scheduling, see for example FDTD, MM and 2DCONV. These benchmarks benefit from improved L1 cache performance and/or coalescing in the L1 MSHRs.

Intra-cluster coalescing when complemented with the coalesced cache (CC) improves performance for both CTA scheduling policies by 15% on average. The coalesced cache extends the opportunity to benefit from coalesced cache lines which leads to an additional average 5% performance improvement beyond ICC. Some cache lines that cannot be serviced by the ICC unit hit in the CC, avoiding an additional access over the NoC. In particular, LUD and 2DCONV experience a substantial performance improvement of up to 67% and 46%, respectively.

We further note that distributed scheduling with ICC yields similar performance benefits as distributed-block scheduling, with an average performance improvement of 10%. However, distributed-block scheduling with ICC is a more robust solution: it leads to substantial performance gains, larger than 10% for six of the benchmarks, whereas distributed scheduling with ICC leads to similarly high performance gains for only four of the benchmarks. The coalesced cache effectively caches coalesced cache lines at the cluster level, leading to an overall performance improvement by 15% on average.

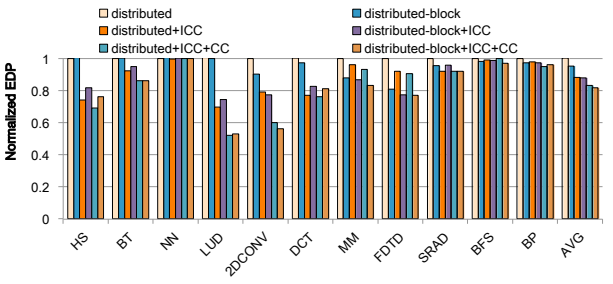

Fig. 12. Normalized EDP for distributed-block scheduling and ICC normalized to distributed scheduling. *Distributed-block scheduling with ICC and CC reduces system EDP by 19% on average.*

8.2 Energy Efficiency

ICC reduces energy consumption by coalescing L1 misses, which reduces the number of requests over the NoC to the L2 cache. Figure 11 quantifies the impact on the overall system (GPU plus DRAM) energy consumption by providing a breakdown of where energy is consumed. We report that distributed-block scheduling by itself reduces energy consumption by 1.2% on average. ICC reduces energy consumption by 4% on average, and up to 9% for 2DCONV. ICC plus CC reduces energy consumption by 6% on average, and up to 19% for 2DCONV. The reduction in energy consumption comes from two sources: reduced NoC energy and reduced L2 energy. The NoC accounts for a significant fraction of total energy consumption, for 25% on average and up to 44% for 2DCONV and BFS. When put together, distributed-block scheduling with ICC and CC reduces NoC energy by 16% on average and up to 30%. The reduction in L2 cache energy is also significant (by 10% on average). However, because of the smaller contribution of the L2 cache to total system energy compared to the NoC, the impact is relatively limited. Overall, we observe significant system energy savings for the benchmarks that benefit from exploiting intra-cluster locality, see for example 2DCONV (19%), LUD (11%), FDTD (8%), HS (6%), BT (5%) and HS (4%).

Figure 12 quantifies the energy-delay product (EDP), a well-established metric for energy efficiency. Distributed-block scheduling by itself improves EDP by 5% on average and up to 20% compared to distributed scheduling. Distributed-block scheduling with ICC leads to an average EDP improvement of 13%, and up to 26%. Distributed-block scheduling with ICC and CC improves EDP by 19% on average, and up to 53% (LUD).

8.3 NoC Traffic

To investigate where the performance improvements and energy savings are coming from, we now report the NoC traffic, which we quantify by counting the number of read requests through the NoC. Figure 13 reports normalized NoC traffic.

Distributed-block scheduling reduces NoC traffic by 7% on average, as a result of coalescing L1 misses in the MSHRs within an SM. We observe a significant reduction in NoC traffic for a number of benchmarks, including FDTD (24%), DCT (15%), MM (10%) and 2DCONV (7%). These

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2019.2895036, IEEE Transactions on Computers

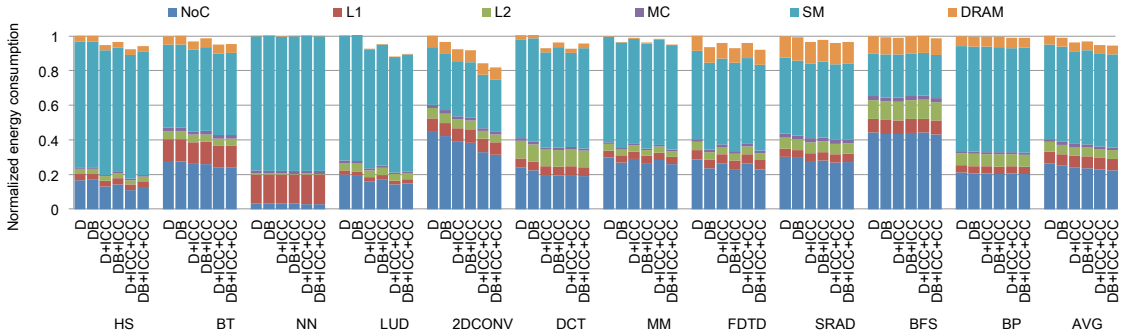

Fig. 11. Energy consumption breakdown normalized to distributed scheduling (D). *Distributed-block scheduling with ICC and CC reduces system energy by 6% on average.*

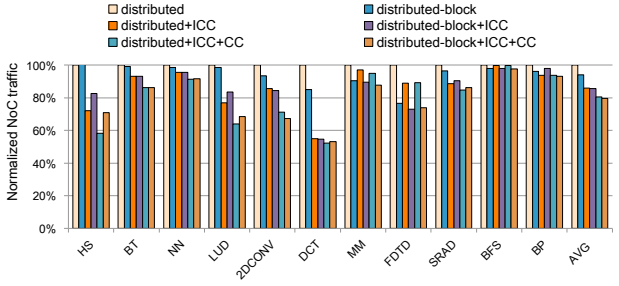

Fig. 13. NoC traffic (number of NoC read requests) for distributed-block scheduling and ICC normalized to distributed scheduling. *Distributed-block scheduling with ICC and CC reduces NoC traffic by 20% on average.*

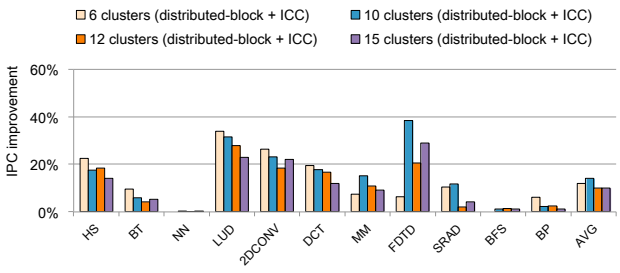

Fig. 14. IPC improvement for distributed-block scheduling with ICC versus distributed scheduling as a function of the number of clusters while keeping total SM count constant at 60 SMs. *Distributed-block scheduling with ICC consistently improves performance across different cluster sizes and effective NoC port count per SM.*

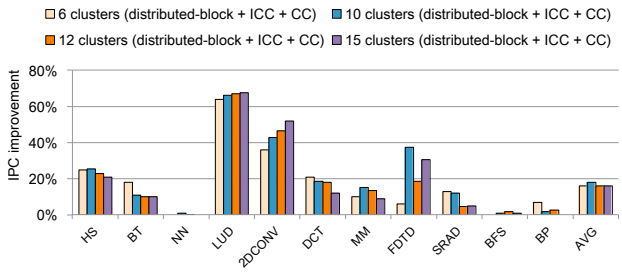

Fig. 15. IPC improvement for distributed-block scheduling with ICC *and CC* versus distributed scheduling as a function of the number of clusters while keeping total SM count constant at 60 SMs. *Distributed-block scheduling with ICC and CC consistently improves performance across different cluster sizes and effective NoC port count per SM; the coalesced cache improves performance consistently.*

benchmarks also experience a significant performance improvement as previously reported in Figure 10. The only exception is DCT, given its high percentage of writes versus reads. Other benchmarks such as HS and BT do not benefit either, due to a high L1 cache miss rate (90% for HS) and limited locality among consecutive CTAs (as is the case for BT).

Some of the inter-CTA locality cannot be exploited through the L1 cache MSHRs within an SM, but can be exploited across SMs within a cluster through the ICC unit, further decreasing NoC traffic by 9% on average. Combining distributed-block scheduling with ICC leads to an average reduction in NoC traffic by 15%. Some benchmarks experience a significant NoC traffic reduction, including DCT (46%), FDTD (27%), HS (18%), LUD (17%) and 2DCONV (16%). These benchmarks are also the workloads experiencing the largest performance and energy improvements. Note though that the correlation is not perfect — NoC traffic reduction also depends on the fraction of read requests. Benchmarks with more write requests than reads do not experience an equally high reduction in NoC traffic.

The coalesced cache keeps track of coalesced cache lines upon eviction from the merge table. This prolongs the opportunity to exploit intra-cluster locality, reducing NoC traffic reduction by an additional 5% on average. The coalesced cache decreases NoC traffic significantly for some benchmarks, including HS (12%), LUD (13%) and 2DCONV (17%). These benchmarks are also the workloads that are more sensitive to the time window, see Figure 3. Distributed-block scheduling with ICC and CC reduces the NoC traffic by 20% on average. A couple of benchmarks obtain a significant NoC traffic reduction, including DCT (47%), 2DCONV (33%), LUD (32%), HS (29%) and FDTD (27%).

8.4 Sensitivity Analysis

Our baseline configuration assumed 12 clusters with 5 SMs each. 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, hence the effective number of NoC ports per SM

effectively increases as we increase the number of clusters. We assume NoC bandwidth is constant across the different configurations — note that NoC bandwidth is bounded by the 8 memory controllers. (For the configuration with 6 clusters, we therefore increase NoC frequency to 1.9 GHz to keep NoC bandwidth constant.) Finally, we also change the number of merge table entries according to the number of SMs in a cluster, i.e., we set the size of the merge table to 96, 60, 48 and 40 entries for 6, 10, 12 and 15 clusters, respectively. We (obviously) set the number of bits in the SM_list field in the merge table to be the same as the number of SMs in a cluster. We assume a 24-entry coalescing cache per cluster.

Figures 14 and 15 report IPC improvement (percentage speedup) as a function of the number of clusters comparing distributed-block scheduling with ICC versus distributed scheduling; Figure 14 considers ICC in isolation whereas Figure 15 considers ICC plus CC. The key observation here is that distributed-block scheduling with ICC and CC is effective across the different GPU architecture configurations. Even with as few as 4 SMs per cluster sharing one NoC port (15 clusters in total), we still observe an average performance improvement of 16% (and up to 68%) from distributed-block scheduling with ICC and CC. The highest performance improvement is achieved for 10 clusters with 6 SMs each. We note an average 14% performance improvement for distributed-block scheduling with ICC and a 18% performance improvement for distributed-block scheduling with ICC and CC.

9 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 proposals exploit inter-CTA locality to improve CTA scheduling. In particular, Lee et al. [7] and Mao et al. [21] propose to dispatch groups of two consecutive CTAs onto the same SM to improve L1 cache performance. Unfortunately, this exploits locality between consecutive CTAs located in a row only. Chen et al. [22] propose a software-hardware cooperative design to exploit spatial locality among different CTAs located in different rows and columns. Li et al. [23] propose software techniques to schedule CTAs with potential reuse on the same SM to exploit inter-CTA locality on real GPU hardware. None of this prior work explores CTA scheduling to improve intra-cluster coalescing opportunities.

Exploiting inter-SM 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. None of prior work considers inter-CTA locality as a potential solution for the GPU NoC bottleneck in clustered GPUs.

GPU NoC optimization. Two recent approaches 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. [14] exploit packet coalescing to reduce data redundancy in GPUs. These two prior approaches 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 this prior work considers 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. [27] 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 a high network bandwidth. Zhao et al. [28] propose a ring-like NoC to provide high bandwidth for servicing reply packets in a cost-effective way. These previous proposals only focused on the NoC topology, but could be combined with our intra-cluster coalescing to further improve their performance.

Memory access coalescing. Coalescing techniques for GPUs have been widely investigated, see for example [8], [9], [14], [29]. Intra-warp coalescing is widely deployed in GPUs to group aligned memory accesses of different threads in a warp [8]. To coalesce memory accesses from different warps, WarpPool [9] merges requests between warps within an SM to increase the effective L1 cache bandwidth. This prior work only targets memory access coalescing within an SM. None notices nor exploits the potential of coalescing redundant memory accesses from different SMs within a cluster.

10 Conclusion

As the number of SMs on next-generation GPUs continues to increase, NoC congestion quickly becomes a key design challenge to scale performance. Clustered GPUs face a severe NoC bottleneck with increasing SM count. To mitigate network congestion, we propose distributed-block CTA scheduling, intra-cluster coalescing (ICC) and the coalesced cache (CC) to exploit inter-CTA locality observed in many GPU-compute applications, coalescing L1 cache misses within and across SMs in a cluster, respectively. Distributed-block scheduling is a two-level CTA scheduling policy that first evenly distributes consecutive CTAs across clusters, and subsequently schedules pairs of consecutive CTAs per SM to maximize L1 cache locality and L1 MSHR coalescing opportunity. ICC groups memory requests from different SMs in a cluster to the same L2 cache line to reduce the overall number of requests sent over the NoC. CC extends the opportunity from coalescing cache lines by caching them at the cluster level for future reference. By

removing redundant NoC traffic, we find that distributed-block scheduling, intra-cluster coalescing and the coalesced cache work synergistically to improve both performance and energy consumption.

Using execution-driven GPU simulation, we find that distributed-block scheduling with ICC and CC improves GPU performance by 15% on average (and up to 67%) while at the same time reducing system energy by 6% (up to 19%) and the energy-delay product by 19% (up to 53%) compared to the state-of-the-art distributed CTA scheduling. 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 and reduce system energy by coalescing memory requests both within and across SMs in a cluster.

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] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

[15] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "CACTI 6.0: A Tool to Model Large Caches," *HP Laboratories*, pp. 22–31, 2009.

[16] 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.

[17] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "GPUWattch: Enabling Energy Optimizations in GPGPUs," in *Proceedings of the Annual International Symposium on Computer Architecture*, 2013.

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[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, pp. 42–45, Jan 2017.

[27] 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.

[28] 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.

[29] B. Pichai, L. Hsu, and A. Bhattacharjee, "Architectural support for address translation on gpus: Designing memory management units for cpu/gpus with unified address spaces," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. New York, NY, USA: ACM, 2014, pp. 743–758. [Online]. Available: http://doi.acm.org/10.1145/2541940.2541942