Analyzing and Leveraging Remote-core Bandwidth for Enhanced Performance in GPUs

Mohamed Assem Ibrahim*, Hongyuan Liu*, Onur Kayiran†, Adwait Jog*
*William & Mary †Advanced Micro Devices, Inc.
Email: {maibrahim,hliu08}@email.wm.edu, onur.kayiran@amd.com, ajog@wm.edu

***Abstract*—Bandwidth achieved from local/shared caches and memory is a major performance determinant in Graphics Processing Units (GPUs). These existing sources of bandwidth are often not enough for optimal GPU performance. Therefore, to enhance the performance further, we focus on efficiently unlocking an *additional* potential source of bandwidth, which we call as remote-core bandwidth. The source of this bandwidth is based on the observation that a fraction of data (i.e., L1 read misses) required by one GPU core can also be found in the local (L1) caches of other GPU cores. In this paper, we propose to efficiently coordinate the data movement across cores in GPUs to exploit this remote-core bandwidth. However, we find that its efficient detection and utilization presents several challenges. To this end, we specifically address: a) which data is shared across cores, b) which cores have the shared data, and c) how we can get the data as soon as possible. Our extensive evaluation across a wide set of GPGPU applications shows that significant performance improvement can be achieved at a modest hardware cost on account of the additional bandwidth received from the remote cores.**

***Keywords*-Bandwidth; GPUs; Network-on-Chip**

I. Introduction

Graphics Processing Unit (GPU) architectures are becoming an inevitable part of every computing system [1] because of their ability to provide orders of magnitude faster execution. They have become the default choice for accelerating innovations in various fields [2]–[10] such as high-performance computing (HPC), artificial intelligence, deep learning, and virtual/augmented reality. Traditionally, GPUs have relied on bandwidth to achieve high throughput [11]–[16]. However, the current sources of bandwidth such as local/shared caches, scratchpad, and memory are often not sufficient for achieving the peak GPU throughput [11], [17]–[21]. In this paper, we focus on dynamically identifying and exploiting an additional source of bandwidth in GPUs, which we call as *remote-core bandwidth*. The source of this additional bandwidth stems from *inter-core locality* [22]–[25] that allows the data required by one of the GPU cores (i.e., L1 read misses) to be also found in the local L1 caches of remote GPU cores. Our analysis shows that this additional source of bandwidth leads to significant improvement in performance, however, can only be leveraged if an efficient inter-core communication is enabled. However, there are several challenges towards designing efficient inter-core communication, which have not been addressed by prior works. In particular, this paper addresses the following research questions.

I) How to determine which data can also be found in the local caches of remote cores? Traditionally, a cache line requested by a core is always found in the GPU memory, as it stores the data required by the kernel(s). However, the requested data may or may not be found in the L1 cache of the remote cores due to static data sharing characteristics or runtime state of the caches [22]–[26]. A mechanism that correctly predicts if the data is shared would reduce unnecessary inter-core communication.

II) How to determine which cores have the data of the requester core? Even if it is known that the data is shared across cores, determining which cores have the shared data is critical. A naive approach of sending request *probes* to all the cores to fetch the data can incur significant latency and consume interconnect bandwidth. Therefore, it is important to determine which cores are likely to have the requested data to reduce the communication overhead.

III) How to get the data as soon as possible without congesting the interconnect? Finally, it is important to search the cores such that we do not saturate the interconnect bandwidth while still reducing the search latency. This latency can be tolerated to a certain extent; however, long latencies can hurt performance [11]. Moreover, long search delays decrease the probability of finding the shared data due to cache evictions at the remote core.

To the best of our knowledge, this is the first work that systematically addresses these questions. Specifically, this paper makes the following contributions:

• We observe a bi-modal distribution of inter-core locality across different load instructions – some instructions use data that is shared across cores and some do not. We leverage this observation and use the program counter (PC) to predict which L1 read misses are likely to be satisfied by the L1 caches of remote cores.

• We develop a low-overhead mechanism that can locally predict which cores are likely to have the shared data. It is based on our key observation that the data required by a core is generally shared across *only a few* cores, which can be detected via sampling a limited number of core replies.

• We develop a novel two-level probing mechanism that searches the identified cores in parallel while considering the interconnect bandwidth consumption.

• Our combined schemes take advantage of the untapped remote-core bandwidth, leading to 21% improvement (up to 40%) in performance if the data is *a priori* known to be shared, and 10% (up to 26%) with our PC-based predictor. These results are averaged across 11 diverse GPGPU applications that exhibit inter-core locality and achieved at a modest area overhead of 0.058 mm^2 per core (determined by

978-1-7281-3613-4/19/$31.00 ©2019 IEEE
DOI 10.1109/PACT.2019.00028

detailed RTL synthesis). Additionally, our proposed schemes do not affect the performance of applications that possess low inter-core locality.

II. Motivation and Analysis

Many important graph and HPC applications are known to be cache sensitive with significant reuse. To capture this reuse, much attention has been given to improving local cache performance in GPUs (e.g., [11], [17], [19], [27], [28]). However, limited focus is given to another type of locality, called as inter-core locality [22]–[25] (i.e., the data required by a core can be found in the local L1 caches of other cores). Inter-core locality primarily results from each core independently requesting data without consulting the L1 cache of nearby cores. We find that in many cases, other GPU cores have previously requested the same data (exact sharing) or nearby data in the same cache line (false sharing) and placed it in their local caches [22]. Consequently, they are also capable of supplying the data and a potential source of memory bandwidth, which we refer to as *remote-core bandwidth*. To unlock this additional bandwidth, efficient inter-core communication is essential.

A. Inter-core Communication Message Flow

We first provide a high-level overview of how L1 read miss requests are routed to other cores to exploit inter-core locality. Under a baseline GPU where inter-core communication is not enabled (Figure 1(a)), a read request which misses in L1 goes through the Network-on-Chip (NoC) and accesses L2 cache. L2 cache either responds with data or forwards the request to its associated memory channel. When inter-core communication is enabled (Figure 1(b)), a read request which misses in L1 (i.e., the requester L1) can probe other L1 caches (i.e., supplier L1s).[1] An L1 read miss goes through the NoC to probe other L1 caches. If a supplier L1 has the data, it will respond with data; if not, it will send a NACK. If no supplier L1 responds with the data (or NACK) in a given amount of time (we define this as *Timeout*), the requester L1 will fall back to the default scenario shown in Figure 1(a) to probe the L2 cache.

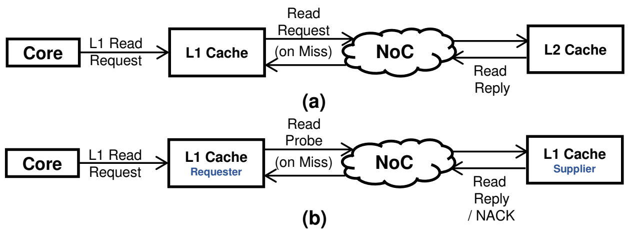


Figure 1: Illustrating the L1 read miss handling when inter-core communication is (a) disabled and (b) enabled.

[1]The inter-core communication in our proposal is enabled for the read requests only and thus can co-exist with the existing cache coherence mechanism. A write request to a shared data in L1 is handled by the default cache coherence mechanism.

B. Potential Benefits of Remote-core Bandwidth

To illustrate the benefits of inter-core communication in GPUs, we consider three different scenarios for probing other GPU cores, as tabulated in Table I. These scenarios are formed based on the questions we raised in Section I: (1) is the data shared?; (2) which remote cores have the data?; and (3) how should the data be fetched? We start with the assumption that the answer to the first question is known *a priori* (we will relax this assumption later in Section III). In other words, we assume a perfect predictor that determine if the required data exists in the L1 cache of at least one remote core.

Table I: Probing/Communication Scenarios.

Scenario	**Is the data shared?**	**Which remote cores have the data?**	**How is the data fetched?**
Perfect Probing (PP)	Known	Known	Zero-cycle communication
Direct Probing (DP)	Known	Known	Direct communication with the nearest supplier
Naive Indirect Probing (n-IP)	Known	Search all the cores	Sequentially search the cores one-by-one

The first scenario, called as *Perfect Probing* (PP), assumes that we oracularly know which cores have the shared data, and this data can be fetched in zero cycles (i.e., no communication overhead). In the next scenario, called as *Direct Probing* (DP), we still assume that the location of the shared data is known, but a mechanism is required to probe the nearest core that shares the data and fetch it. Finally, in the *Indirect Probing* mechanism (IP), we assume that the location of the shared data is unknown, and a single probe request has to sequentially search all remote cores one-by-one to fetch the data. This is a naive implementation of IP, and hence mentioned as *Naive IP* (n-IP) in Table I. Section III discusses our final probing scenario (not shown in Table I), called as *Realistic Probing* (RP), which adopts intelligent IP mechanisms to efficiently fetch data from the remote cores, and also a technique to determine if a cache line is shared by other remote cores.

Figure 2 shows the reply bandwidth received by each core in terms of L2 reply bandwidth and remote-core reply bandwidth, and the performance in terms of IPC (both normalized to the baseline with no inter-core communication) under the aforementioned probing scenarios. Four observations are in order. First, on average, the total reply bandwidth is higher under PP scenario compared to other scenarios. Therefore, IPC is also the maximum in this scenario. Specifically, because $IPC \propto BW/MPKI$, where MPKI is misses-per-kilo-instruction [14], [29], unlocking the remote-core bandwidth shall increase the overall available bandwidth, which in turn improves IPC. Thus, even if the overall memory bandwidth can be increased by adding more memory partitions, the additional on-chip bandwidth from remote cores can further enhance performance.

Second, the remote-core bandwidth under DP is lower

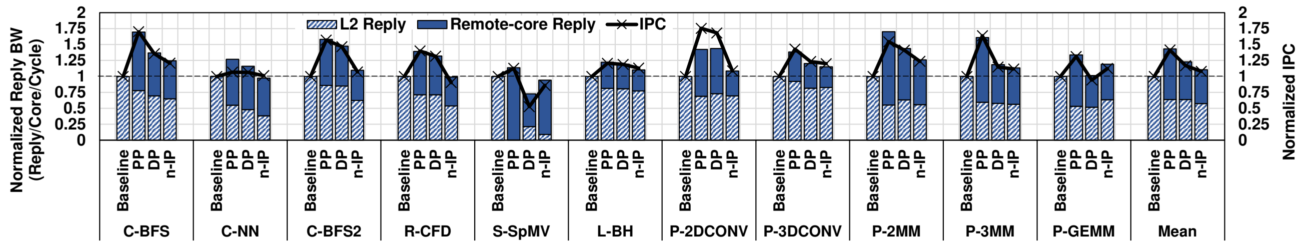


Figure 2: Illustrating the performance benefits of remote-core bandwidth for various scenarios. Section IV-A has the details on the experimental methodology.

in many applications compared to PP. This is due to the overhead of fetching the data from remote cores. This overhead is not only in terms of latency of fetching the data; in some cases, the data is no longer present in the cache by the time a probe reaches the remote destination. As shown in Figure 3, this results in a loss in remote hit rate (i.e., inter-core locality), which is defined as the ratio of replies received from the remote cores to L1 read misses. Figure 3 results are normalized to PP with the raw inter-core locality numbers of PP shown at the top of each application. Third, with n-IP, the overhead of naive searching is more significant because of the NoC contention, which further decreases the remote-core bandwidth of n-IP, and thus its performance.

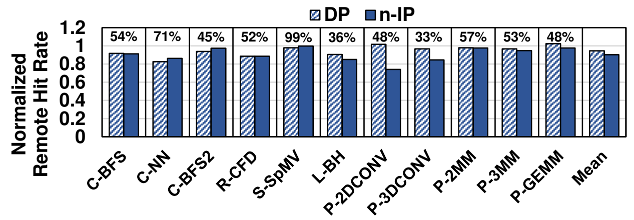


Figure 3: Illustrating the loss of inter-core locality (remote hit rate) for various scenarios.

Finally, the reply bandwidth for `P-2DConv` is slightly higher with DP than PP, however, IPC with PP is higher than DP. This is attributed to the runtime state of the caches such as cache evictions [22]. Specifically, using $IPC \propto BW/MPKI$, the runtime state of the cache affects $MPKI$, which may decrease IPC. Also, using zero-cycle communication is the main performance booster in PP. In summary, utilizing remote-core bandwidth boosts overall performance and is complementary to the bandwidth received from the memory partitions.

III. Inter-core Communication in GPUs

In this section, we discuss the design of inter-core communication policies, which are required to exploit the inter-core locality opportunities discussed before.

A. Baseline Architecture and Communication Fabric

Our baseline GPU consists of 28 cores (also called Compute Units (CUs) or Streaming Multiprocessors (SMs)) connected to 8 L2 slices and memory channels via NoC. Each core has a local L1 cache, which is connected to its associated NoC interface. There is a shared L2 cache that is interleaved across 8 banks. Each L2 bank is connected to a NoC interface for the incoming L2 requests and to its corresponding memory controller (MC) for forwarding the requests to memory in case of L2 misses. We use two separate NoCs: request and reply NoCs to avoid protocol deadlock [12]. The L2 requests, probes, and the NACKs use the request NoC, while the replies from cores or L2 use the reply NoC. Similar to recent works [30]–[33] in GPUs, we model a 2D mesh NoC for connecting cores to memory channels because it inherently enables core-to-core communication. Additionally, a 2D mesh NoC is scalable as the number of cores increases because it is modular and easier to lay out on a chip [12], [34], [35].

B. Communication Knobs: Probe Coverage and Probe Rate

To address the performance overheads of inter-core communication discussed in Section II, we consider modulating the number of cores to search (i.e., controlling the *probe coverage*) and/or the rate at which the cores are searched (i.e., controlling the *probe rate*). Formally, we define IP(C,S,P), where S probes are sent per read miss with a probability of P ($0 <= P <= 1$), or $S-1$ probes per read miss are sent with a probability of $1-P$, to search C cores in the GPU system. For example, IP(15,2,0.2) implies that a core searches 15 remote cores by sending 2 probes per request for around 20% of its L1 read misses and 1 probe per request for the rest. In the case of two (or more) probes per request, the target cores (i.e., the cores to be probed) are disjointly divided among the probes as equally as possible to be searched in parallel. For example, under IP(15,2,0.2), the first probe searches 8 cores and the second probe searches 7 cores. Probe coverage is determined by the value of C and the probe rate is determined by the value of the pair (S,P). Note that both probe coverage and rate affect the consumption of request NoC bandwidth (Request/Core/Cycle), which is inherently limited. Therefore, it is important to control each of these parameters carefully (C, S, and P) to optimize performance.

C. Which Remote Cores Have the Data?

Effect of Probe Coverage. Figure 4 shows the effect of probe coverage on the remote hit rate and the request

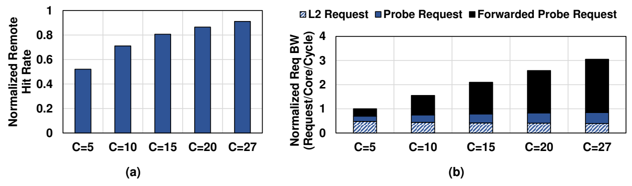


Figure 4: Illustrating (a) inter-core locality (normalized to the PP scenario) and (b) request bandwidth (normalized to the IP(5,1,1)) under IP(C,1,1) averaged across the evaluated applications.

bandwidth under IP(C,1,1). The request bandwidth has three components: a) requests sent to L2, b) probe requests sent to remote cores, and c) forwarded probe requests from remote cores. We observe that probing a limited number of cores can reduce the consumption of the request bandwidth at the cost of reducing inter-core locality. Therefore, it is important to carefully select the number of target cores that balances the available inter-core locality and the NoC overhead (e.g., $C = 15$ in Figure 4).

Which Cores to Probe? Our next goal is to identify the target cores. This step consists of predicting which cores have a high probability of providing the shared data and selecting a subset of them to probe. Figure 5 shows the heat map of cores that can supply data to requester cores for representative applications. Each cell in the heat map represents how many times a particular core is able to respond to an incoming probe with data. A requester core is any core that had at least one remote request during execution. This data is collected assuming that probes can be sent in zero cycles. We observe from this figure that some cores can provide the data more than the others. For example, in `C-BFS`, the highlighted core is more likely to provide the data. Similar behavior is observed in the other applications as shown in Figure 5. Therefore, probing the cores that have a higher probability of responding with data is potentially beneficial because it would maintain inter-core locality, and reduce the request NoC bandwidth consumption.

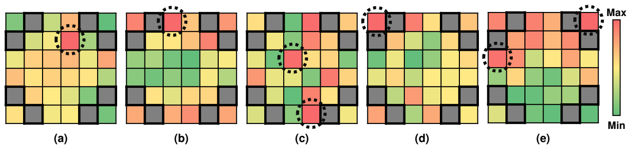


Figure 5: Supplier heat map for (a) `C-BFS`, (b) `R-CFD`, (c) `S-SpMV`, (d) `L-BH`, and (e) `PP-2MM` under the baseline 6×6 mesh NoC. L2 partitions (and MCs) are highlighted using thick borders. For these applications, the maximum value in the heat map is 1.94× the minimum, on average.

Selection Criteria. There are multiple design choices when selecting the set of target cores. Figure 6 shows the performance of IP(C=27,1,1) under two selector mechanisms, where 27 is the maximum number of cores that can be searched in our 28-core baseline architecture. In *index-based*, which is used in n-IP, a probe sequentially searches the cores assigned to it based on the core index in ascending order. We propose a *supplier-based* selector. In this mechanism, each core locally and periodically collects the number of data replies received from other cores. This information is then used to assign probability values for selecting the target cores.[2] To reduce the bias in the selection process, (1) the collected data is reset at the end of each period, and (2) the cores that have not replied with data during the current period are given a very small probability (half of the lowest collected non-zero probability) to be selected as target cores. Then, C target cores are selected for probing based on the collected and modified probability of finding data in each core. We observe from Figure 6 that our *supplier-based* selector outperforms the *index-based* selector because of its ability to adapt to the dynamic changes in the sharing patterns.

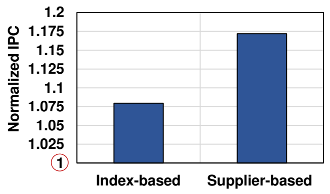


Figure 6: Performance of selection criteria under IP(27,1,1) averaged across the evaluated applications. Results are normalized to the baseline with no inter-core communication.

D. How is the Data Fetched?

Effect of Probe Rate. We study the effect of probe rate with the help of Figure 7 that shows the performance of IP(27,S,P) for `C-BFS` under index-based and supplier-based selection criteria. In the index-based case, we obtain the highest IPC when $S = 1$ and $P <= 1$. In other words, if we send only one probe for a portion of the read miss requests, while the rest are directly sent to L2, then performance can improve; with multiple probes per request, performance drops.

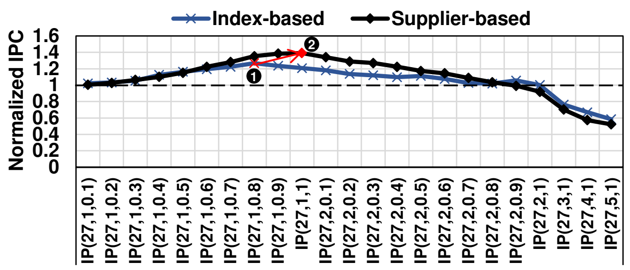


Figure 7: Performance of `C-BFS` with index-based and supplier-based selectors under IP(27,S,P). Results are normalized to a baseline with no inter-core communication.

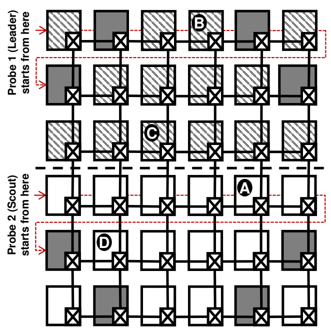
[2]For example, in a four-core system, if Core1, Core2, and Core3 responded to Core0 with data 5, 3, and 2 times during a period, respectively, then Core0 will select Core1, Core2, and Core3 as target cores with 50%, 30%, and 20% probability, respectively.

In the supplier-based case, we observe that the peak performance for C-BFS is shifted to the right (from ❶ to ❷). This confirms that selecting which cores to search first has a positive impact on performance. However, performance still drops when using $S > 1$. This is because multiple probes can cause contention in the request NoC resources (e.g., links, buffers, virtual channel (VC) allocation, switch (SW) allocation). In addition, multiple parallel probes may lead to redundant replies, thereby congesting the reply NoC further. Therefore, it is important to modulate the probe rate carefully while handling the redundant replies.

One way to improve performance in the presence of parallel probes is to limit the number of data replies to one, so that reply NoC is not further congested. Based on this idea, we propose a novel *Two-level Probing* scheme.

Two-level Probing. Our two-level probing scheme overcomes the issue of redundant replies by leveraging two probe types. The first type is the *Leader Probe*, which looks for the data in its assigned target cores and returns once the data is found (similar to a normal probe). The second type is the *Scout Probe*, which also looks for data within its target cores; however, once it finds the data, it does not return with data. Instead, it appends the core identifier to the candidate suppliers list and then searches the rest of the assigned cores. The scout probe returns once it completes searching. If the leader does not return with the data, then the requester initiates the *second-level of probing* by injecting a leader-like probe to search all the candidate suppliers sequentially and return if it finds the data (or failed). There is a singular leader probe in our scheme, while the rest of the parallel probes are scouts.

To illustrate how two-level probing works, let us consider an example in Figure 8. Assume that $S = 2$; the leader probe searches the shaded cores, while the scout probe searches the others. Assuming that the data is present in cores 🅐, 🅑, 🅒, and 🅓, the leader returns with data (from 🅑) after searching three cores, and the scout searches all the assigned fourteen cores and returns with candidate suppliers 🅐 and 🅓. However, because the data is found by the leader, these candidates are ignored. In another scenario, assume that data is only found in 🅐 and 🅓. In this case, the leader searches all the assigned cores and returns with a NACK back to the requester. The scout returns with the candidate suppliers (🅐 and 🅓), so the requester injects a leader-like probe that searches 🅐. On failing to find the data (for example, evicted by the time the probe reaches 🅐), it searches 🅓. In summary, the advantage of two-level probing is the elimination of redundant replies from different remote L1 caches.


Probe 1 (Leader) starts from here
Probe 2 (Scout) starts from here
B
C
A
D
Figure 8: Illustrating how two-level probing works. The dotted red lines represent the order of searching the cores in this scenario. Gray nodes are connected to L2s and MCs.

Discussion. Figure 9 shows the average performance under IP(C, S, P) when S and P (probe rate) are varied, while C (probe coverage) is set to 5, 10, 15, 20, or 27. Since the request NoC bandwidth is a function of the number of probes sent and the number of cores to search, decreasing the number of target cores is expected to release more NoC resources to accommodate more probes. In that case, we observe a further shift to the right in the peak performance (i.e., we observe better performance when more than one probe search in parallel). Using $C >= 20$, we barely observe any benefits from using $S >= 2$. We can still get benefits from sending a mix of one or two probes, but not beyond two probes. On the other hand, using $C = 15$, we observe a lower reduction in performance even with $S >= 2$. Both $C = 10$ and $C = 5$ lead to better performance with $S >= 2$ compared to $C >= 15$. To summarize, a trade-off between the number of cores to search and the parallel probes to inject is required to balance the overall request bandwidth and to control the forward request bandwidth.

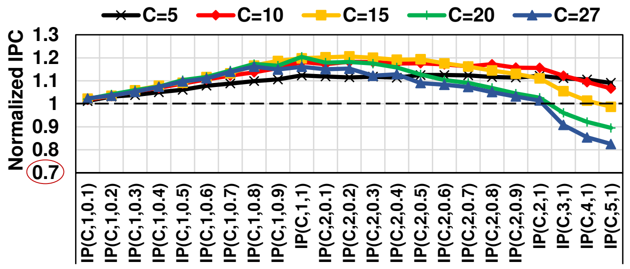


Figure 9: Performance with supplier-based selector and two-level probing under IP(C, S, P) averaged across the evaluated applications. Results are normalized to the baseline with no inter-core communication.

E. Is the Data Shared?

We have so far assumed that a requester core had *a priori* knowledge of whether the data it requests is cached by remote cores. In this section, we propose a two-bit predictor that uses the Program Counter (PC) information to predict, locally at each core, if the required data exists in a remote L1 cache. If our predictor anticipates that the data is shared, the supplier-based core selector and the two-level probing techniques are utilized to search for the required data. Otherwise, the request is sent directly to L2.

Why Prediction? We start by studying the need for a predictor. From Figure 3, we observe that the raw volume of inter-core locality is not 100% of the read misses. Additionally, falsely assuming that a read miss is shared causes latency overhead for the request sent to L2, as probing remote L1 caches imposes a search delay. As a result, if we assume every read miss is shared, it will cause unnecessary search overhead in the cases when the data is not shared. For example, in C-BFS, the percentage of shared read miss request is around 54%. Thus, if we probe remote L1 caches on every read miss, we will end up with a failed search for 46% of the requests. In other words, almost half of the requests will endure unnecessary delay and consume request NoC bandwidth whereas the data is not shared.

PC and Inter-core Locality. As a first step to designing a sharing predictor, we need to identify a simple local parameter to use. We investigated multiple parameters, and we found that request origin PC is a good metric to consider. Figure 10(a) shows the volume of remote hits for each PC value in C-BFS. We observe that out of nine PCs, only two have inter-core locality ($PC = 80$, $PC = 288$), and one of them ($PC = 288$) features $> 90\%$ remote hits out of 350120 remote read accesses. We observe similar behavior in other evaluated applications. This observation leads to the design of our PC-based predictor. If we keep track of the number of probe requests sent and the core replies received per PC, then we can develop a local scheme that predicts if the data is shared.

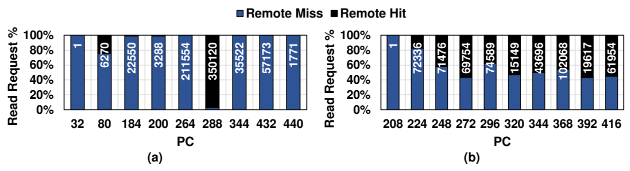


Figure 10: Remote hits vs. Remote misses for different PC under (a) C-BFS, and (b) P-2DCONV. The numbers on each bar represent the total remote read accesses per PC.

Two-bit PC-based Predictor. Figure 11 shows the finite state machine for our proposed predictor. It keeps track of four different states (hence two-bit) per PC. Specifically, the states are *Strong Shared*, *Weak Shared*, *Weak Non-shared*, and *Strong Non-shared*. The predictor optimistically assumes sharing and starts from a *Strong Shared* state. If a given PC fails to show a dominant sharing behavior, it will end up in the most restrictive state *Strong Non-shared*. Each state utilizes three variables (W, S, and T). These variables are used along with the inter-core replies count (R) to decide the next state. Given state *i*, W_i sets the number of read misses to be considered during state *i*. S_i sets the number of read misses that are assumed to be shared out of W_i requests ($W_i >= S_i$). Once W_i requests are processed, we compare the number of core replies R_i to the threshold T_i and based on that, the next state is determined. Based on the current state, if $R_i \geq T_i$, then the next state is set as the state that provides more sharing. On the contrary, if $R_i < T_i$, then the next state is the more restrictive state.

Discussion. We will discuss the effectiveness of the proposed predictor and its accuracy in Section IV. However, we want to point out one possible concern with our predictor.

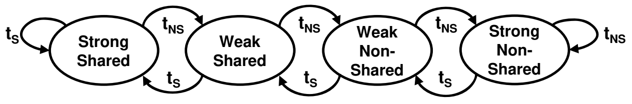


Figure 11: Two-bit PC-based sharing predictor. t_S refers to a $Sharing$ transition, while t_{NS} refers to a $Non\text{-}Sharing$ transition.

In Figure 10(b), we show the volume of remote hits for each PC value in P-2DCONV. In contrast to C-BFS, P-2DCONV does not have a few dominant PC values. Specifically, eight out of ten PCs have around 50% remote hits. Additionally, such behavior is spread throughout the execution (not shown). As a result, it is difficult to have high accuracy under such application behavior.

F. Implementation Details

Figure 12 shows the architectural diagram of our proposal. We start by explaining the design choices and scenarios in our system. Then, we study the area, power, and communication overheads.

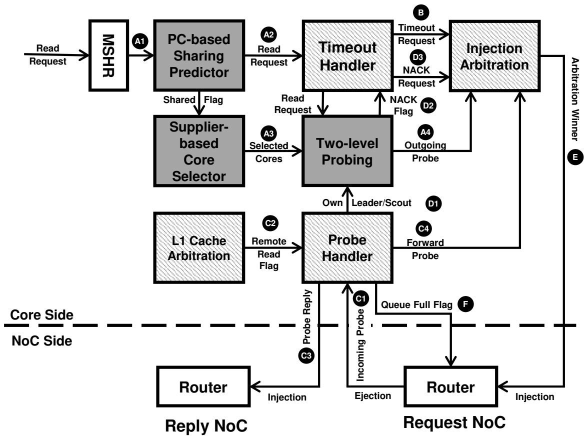


Figure 12: Hardware organization of our proposal. The shaded components are used for inter-core communication. The gray components are added to support our proposal.

Probe Injection. On an L1 read miss, a request is added to MSHR to be passed down the memory hierarchy. First, the request is sent to the *PC-based Sharing Predictor* A1 to locally predict if the data is present in remote L1 caches. If the request is predicted to be shared, it will be (1) added to a queue (*Selective L2 Requests*) in the *Timeout Handler* A2 that selectively sends the request to L2 if needed, and (2) sent to the *Supplier-based Core Selector* to select the target cores for probing A3. Then, the *Two-level Probing* mechanism determines how many probes to send (based on S and P), assigns the target cores to the generated probes, and adds the probes to a queue (*Outgoing Probe Requests*) holding the core's own probes for injection arbitration A4.

Selective L2 Request Timeout. In some cases, probe requests take a long time to return (with data or NACK). This might be due to several reasons related to NoC congestion and queuing. We need a failsafe mechanism to ensure forward progress. Therefore, for every read miss predicted as shared, a corresponding L2 request is also generated, and placed into *Selective L2 Requests* queue. Every cycle, the *Timeout Handler* checks if the head of the queue timed out. Timeout means that the injected probe(s) failed to retrieve the data from the target cores in a timely manner. In that case, the head of the *Selective L2 Requests* queue competes for injection to be sent to L2 B.

Handling Other Cores' Probes. On receiving an incoming probe from a remote core, the probe is added to a queue (*Incoming Probe Requests*) in the *Probe Handler* module C1. The forwarded probe is processed to differentiate between a leader probe, a scout probe, or a received NACK. In case of a leader or a scout, the *Probe Handler* consults the *L1 Cache Arbitration* module that prioritizes the local cache accesses over remote reads.[3] In case of no local cache access, the *L1 Cache Arbitration* module informs the *Probe Handler* C2 to check the L1 cache if the required data is cached.

If the incoming probe is a leader, and the data is not found, the probe is added to a queue (*Forwarded Incoming Probes*) to forward it to the next target core (or the requester if no more target cores). However, if the data is found locally, then a probe reply is added to a queue (*Replies to Incoming Probes*) holding the replies to be sent to the requester cores. The rationale behind this queue is to mitigate the head-of-line blocking that can occur in the *Incoming Probe Requests* queue if the reply failed to find space for injection into the reply NoC. The head of the *Replies to Incoming Probes* is pushed into the reply NoC C3. On the other hand, a scout probe updates its candidate supplier list if the data is found, and is always added to the *Forwarded Incoming Probes* queue to be sent to the next target core (or the requester if no more target cores). The head of the *Forwarded Incoming Probes* contends for injection into the request NoC C4.

In case of a returning own leader/scout, the *Probe Handler* notifies the *Two-level Probing* module D1 to keep track of the injected probes per request. If all outstanding probes are received without data reply or candidate suppliers, then the *Two-level Probing* module informs the *Timeout Handler* D2. If the timeout of the failed request has not fired yet, it is retrieved from the *Selective L2 Requests* queue to compete for injection to be sent to L2 D3.

Injection Arbitration. Our design supports different types of messages to be injected into the request NoC. Consequently, to keep the system stable, we must maintain the injection rate into the NoC. We do so by arbitrating between five different request types (ordered from the highest to the lowest priority): non-shared requests, selective L2 requests, forwarded probes, processed NACKs, and outgoing probes. The *Injection Arbitration* selects the winner of the arbitration to be injected into the request NoC based on the priorities of the competing requests E.

Deflection of Incoming Probes. To control the queuing delay at the core, a mechanism is required to limit the number of probes received by a given core. If the *Incoming Probe Requests* queue is full, we deflect the incoming probes at the NoC level by passing a signal from the core to the NoC router to convey the unavailability of queue space F. The router then deflects the probe request to its next target cores or to its requester if no more target cores exist.

Overhead. The *PC-based Sharing Predictor* supports up to 64 PC values. We empirically select the values of W, S, and T based on the following, $W_i = 32 \times 2^i$, $S_i = W_i/4^i$, $T_i = ceil(S_i/8)$, where $0 \leq i \leq 3$. Both *Timeout Handler* and the *Two-level Probing* modules track up to 32 outstanding requests, which is the MSHR size. The *Supplier-based Core Selector* monitors the replies from 27 remote cores (in our 28-core baseline GPU) over a period of 8192 cycles. Finally, we empirically choose 2048 cycles as the timeout value in the *Timeout Handler*. Under this timeout, only 0.7% of the probe requests fail to return with a reply (or a NACK).

To estimate the area overhead, we differentiate between the hardware used to enable inter-core communication (shaded components in Figure 12), and the hardware used to optimize such communication (gray components in Figure 12). We faithfully synthesized the RTL design of the hardware required for the inter-core communication and our schemes using the 65nm TSMC libraries in the Synopsys Design Compiler. We use these synthesized Verilog models for the area and leakage power. Additionally, we use DSENT [37] to estimate the NoC dynamic power assuming a 45nm technology. The area overhead for inter-core communication is 0.089 mm^2 per core, while the area overhead for our schemes is 0.058 mm^2 per core. The total leakage power overhead is 2.022 mW per core. The difference in the dynamic power compared to the baseline is 0.05794 W.

In terms of communication overhead, we add 1-bit in the request to mark as a probe, and 1-bit to identify as a leader or scout. A 32-bit group identifier is added to uniquely identify the probes belonging to the same request. Additionally, up to fifteen target cores need to be searched, and each core needs $ceil(log_2 27)$ bits, that is 75 bits required in total. All this overhead in the request fits in the baseline flit size of 32 bytes.

IV. Experimental Evaluation

A. Evaluation Methodology

We model our schemes and inter-core communication using a cycle-level simulator – GPGPU-Sim v.3 [12]. A detailed platform configuration is described in Table II. We use sixteen applications from five benchmarks suites (CUDA SDK (C) [38], Rodinia (R) [39], SHOC (S) [40], Lonestar (L) [41], and PolyBench (P) [42]) for evaluation. Eleven out of sixteen applications have inter-core locality greater than

[3]Dual ported caches may be needed for applications where L1 bandwidth is not sufficient [36]. However, we do not observe L1 bandwidth as a bottleneck in our applications and hence arbitration is sufficient.

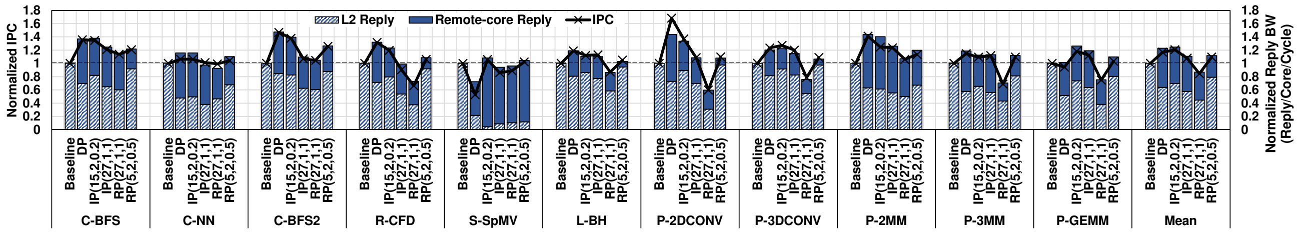


Figure 13: Illustrating the benefits of the proposed schemes in terms of IPC and reply bandwidth.

Table II: Configuration parameters of the simulated GPU.

Core Features	1400MHz core clock, 28 cores, SIMT width = 32 (16×2)
Resources / Core	48KB scratchpad, 32KB register file, Max. 1536 workitems (48 wavefronts, 32 workitems/wavefront)
L1 Caches / Core	16KB 4-way L1 data cache, 12KB 24-way texture cache, 8KB 2-way constant cache, 2KB 4-way I-cache, 128B cache block size
L2 Cache	8-way 128 KB/memory channel (1MB in total), 128B cache block size
Features	Memory coalescing and inter-wavefront merging enabled, immediate post dominator based branch divergence handling
Memory Model	8 GDDR5 Memory Controllers (MCs), FR-FCFS scheduling, 16 DRAM-banks, 4 bank-groups/MC, 924 MHz memory clock, Global linear address space is interleaved among partitions in chunks of 256 bytes Hynix GDDR5 Timing [43], t_{CL} = 12, t_{RP} = 12, t_{RC} = 40, t_{RAS} = 28, t_{CCD} = 2, t_{RCD} = 12, t_{RRD} = 6, t_{CDLR} = 5, t_{WR} = 12
Interconnect	6×6 mesh topology, 700MHz interconnect clock, 32B flit size, 1 VC per port, 8 flits/VC, iSLIP VC and switch allocators

30% (Figure 2). The rest of the applications have inter-core locality less than 10%.

B. Experimental Results

In Section III, we studied the effect of both probe coverage C and probe rate (S, P) on the efficiency of the inter-core communication under a mesh-based system. We proposed three techniques (supplier-based core selector, two-level probing, and PC-based sharing predictor) to exploit the remote-core bandwidth via efficient inter-core communication. We evaluate IP(C=15,S=2,P=0.2), an IP scenario that incorporates supplier-based core selector and two-level probing under a perfect sharing predictor. Although IP knows the sharing information *a priori*, we investigate it thoroughly as it gives an attainable upper bound of the inter-core communication benefits via our schemes. In order to reach such upper bound, we evaluate RP(C=5,S=2,P=0.5), a *Realistic Probing* scenario that does not need any software support and adopts PC-based sharing predictor in addition to supplier-based core selector and two-level probing.

We choose IP(15,2,0.2) as it balances the trade-off between losing inter-core locality (due to searching fewer cores) and incurring latency (due to searching more cores). In general, given an arbitrary GPU, searching 35%-55% of the cores is a valid choice to maintain the required balance under IP scenario. Also, using two probes parallelizes the search process without overwhelming the request NoC resources. For RP(5,2,0.5), we reduce the number of target cores ($C = 5$) because we use a realistic PC-based predictor. Specifically, if we use $C = 15$, any misprediction will result in searching fifteen cores even though the data is not shared. This leads to unnecessary latency overhead for the whole data fetching process. In general, under RP, searching 15%-25% of the cores balances the inter-core locality and the request NoC bandwidth consumption. Also, to further reduce the search overhead, RP(5,2,0.5) uses a higher probe rate. We compare these mechanisms against:

- **DP** utilizes a perfect sharing predictor and sends a probe request to the oracularly known nearest sharer (Section II).
- **IP(27,1,1)**, which is equivalent to n-IP, uses a perfect sharing predictor, however, it searches all the cores sequentially based on core index to find the shared data (Section II).
- **Cooperative Caching Network (CCN)** [24] uses a ring NoC to connect all the cores. On a read miss, CCN traverses the ring and searches the cores sequentially. To limit the search overhead, a throttling scheme based on the ratio between replies received and requests sent, over a sampling window, is used. Since CCN NoC is a crossbar augmented with a ring, we emulate it by using index-based core selector under RP(27,1,1).
- **Locality-Aware Last-Level Cache (LA-LLC)** [44] utilizes a locality-aware L2 that records the last sharer core. Upon receiving a read request from a core, the locality-aware L2 forwards the request to the last sharer in case of a hit, instead of serving the request.

Effect on Performance. Figure 13 shows the performance of our proposed schemes in terms of IPC and total reply bandwidth received by a core (in terms of L2 reply bandwidth and remote-core reply bandwidth), respectively. The results are normalized to the baseline architecture with no inter-core communication. We draw five main observations. First, IP(15,2,0.2) achieves 21% and 8% IPC improvement over the baseline and IP(27,1,1), respectively. The superiority of IP(15,2,0.2) over the baseline comes from unlocking the remote-core bandwidth, thus increasing the total available on-chip bandwidth. However, higher performance compared to IP(27,1,1) comes from searching fewer cores for the required data with higher confidence. Also, the possibility of sending two parallel probes helps in improving the performance as it cuts down the search latency. Second, DP yields better performance compared to IP(15,2,0.2) for almost all evaluated applications except `S-SpMV` and `P-GEMM` (also observed in Figure 2). Such counter-intuitive behavior for these two applications is due to the existence of only a few supplier cores for the majority of the requests (Section III-C), leading to NoC hotspots near some cores under DP. Consequently, the remote-core bandwidth is reduced. In contrast, under IP(15,2,0.2), if a given target core is busy, the request is deflected to the next target core (Section III-F) thereby alleviating hotspots. Moreover, DP is dependent on a single target core, thus it risks the possibility of not finding the data due to eviction and falls back to probing L2/memory. On the other hand, IP(15,2,0.2) searches more cores, so even if a target supplier core evicted the data, the probe moves to the next core in its supplier list.

Third, the performance of RP(27,1,1) is lower than the baseline. This is because of the misprediction overhead. The overhead of searching 27 cores for each misprediction causes a 15% drop in IPC. Therefore, searching less number of cores mitigates the misprediction overhead. Fourth, RP(5,2,0.5) performs better than IP(27,1,1), that utilizes perfect sharing predictor, because of its lower search overhead. Specifically, RP(5,2,0.5) searches only 5 cores compared to 27 cores in case of IP(27,1,1). Also, RP(5,2,0.5) divides the search process among two probes. As a result, even in case of failing to find the data, the smaller search space and the parallel search lessens the overhead. Fifth, the total reply bandwidth follows the same trend as IPC. This conforms to what we discussed in Section II. Additionally, the reply bandwidth from the remote cores in RP(5,2,0.5) is less compared to the other schemes. This is because RP(5,2,0.5) searches 5 cores only, thus perceives lower inter-core locality (refer to Figure 4(a)).

Figure 14(a) shows the precision and recall of RP(5,2,0.5).[4] In general, we find precision and recall to be high for many applications, except a few ones. These applications do not have a few dominant *PC* values as previously discussed in Figure 10(b). On average, RP(5,2,0.5) achieves 72% precision and 88% recall. Since the precision controls the misprediction volume, we investigate the sensitivity to different precision values by studying an imperfect IP. Figure 14(b) shows the effect on IPC using imperfect IP(5,2,0.5) and imperfect IP(15,2,0.2), respectively, under different precision values (100%, 95%, 90%, 80%, and 70%). These precision values are achieved by injecting non-shared requests into the NoC. A precision of $X\%$ under IP means that $(100 - X)\%$ of the non-shared requests are considered as shared. We observe that the drop in IPC in IP(15,2,0.2) increases with less precise predictors (up to 85% performance loss). This is because the unnecessary overhead per mispredicted request is high (searching 15 cores). However, in IP(5,2,0.5), the drop is less severe (up to 45%) due to lower misprediction overhead (searching 5 cores).

We can further bridge the gap between RP(5,2,0.5) and

[4]Precision measures the percentage of the shared predictions that were truly shared. Recall measures the percentage of the truly shared cases the predictor identified.

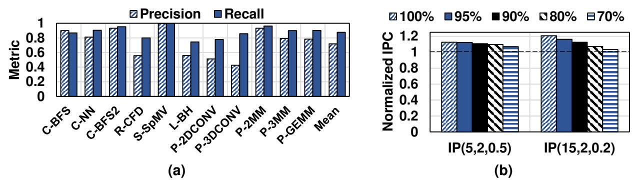


Figure 14: Illustrating (a) Precision and Recall for RP(5,2,0.5) and (b) Effect of prediction precision on IP.

IP(15,2,0.2) if a software-based technique or a programmer input is utilized to provide sharing insight. For example, if a software-based mechanism provides the sharing PC information (instead of using the PC-based predictor), we can achieve performance improvement more than RP(5,2,0.5). Specifically, for `C-BFS2` and `S-SpMV`, an IPC improvement of 37% and 5% is achieved respectively, compared to 38% and 6% in the case of IP(15,2,0.2). To conclude, any increase in the prediction precision helps improving the performance of RP(5,2,0.5).

Finally, we evaluate RP(5,2,0.5) against LA-LLC. On average, RP(5,2,0.5) achieves 10% IPC improvement compared to 2% from LA-LLC. LA-LLC uses the existence of the data in L2 as sharing indicator and forwards the read request to the last sharer core instead of serving at L2. However, the data may be evicted by the time the request reaches the last sharer. This degrades LA-LLC overall prediction precision to an average of 60% and as low as 40% for applications like `P-3MM`, and `P-GEMM`. Also, considering only the last sharer, vs. five cores in RP(5,2,0.5), in the search space decreases the chances of finding the data.

In summary, using IP(15,2,0.2) allows for higher performance as it balances the trade-off between searching more cores vs. sending more probes. However, searching fewer cores as in RP(5,2,0.5) is favored if a low-overhead option is required to balance out any penalty due to mispredictions.

Effect on Link Utilization. Figure 16 shows the effect of IP(15,2,0.2) and RP(5,2,0.5) on the request and reply NoC link utilization. We choose three applications as representatives and compare both mechanisms to baseline and DP. Two observations are in order. First, in the request NoC, both IP(15,2,0.2) and RP(5,2,0.5) have higher link utilization compared to baseline and DP. This is a result of utilizing the links to communicate among cores for searching and retrieving the required data. IP(15,2,0.2) achieves better link utilization in a couple of applications (e.g., `C-BFS`) due to searching more cores. Second, in the reply NoC, IP(15,2,0.2) and RP(5,2,0.5) have similar behavior in the highly utilized links, however, the lowest utilization in IP(15,2,0.2) is higher than in RP(5,2,0.5). This is because IP(15,2,0.2) searches more cores compared to RP(5,2,0.5), thus enabling more sources to deliver replies. Subsequently, more links are used to retrieve data from the target cores.

Performance Impact on Applications with low Inter-core Locality. Some applications have either low inter-core locality or none. Figure 17 shows the performance

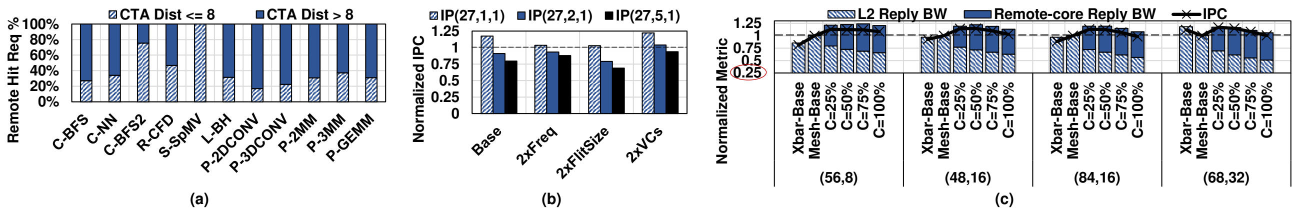


Figure 15: Sensitivity studies on (a) CTA Scheduling, (b) NoC Resources, and (c) NoC Size.

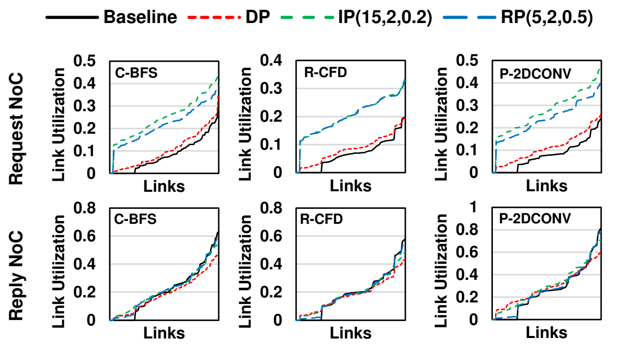


Figure 16: Illustrating the effect of the proposed schemes on the request and reply NoC link utilization.

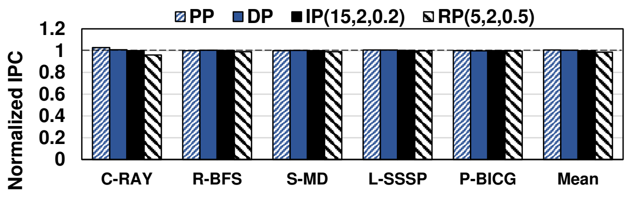


Figure 17: Illustrating the effect of the proposed schemes on applications with low inter-core locality. Results are normalized to the baseline with no inter-core communication.

of five applications, from different benchmarks suites, with $< 10\%$ inter-core locality under PP, DP, IP(15,2,0.2), and RP(5,2,0.5). Two observations are in order. First, the performance gain from PP, DP, or IP(15,2,0.2) is less than 1%. This is due to the reduced scope of inter-core locality. Second, our RP(5,2,0.5) does not affect the evaluated applications negatively. On average, IPC under RP(5,2,0.5) drops 1% for these applications. This is because the small scope of sharing drives the PC-based sharing predictor towards the most restrictive *Strong Non-shared* state which assumes less shared requests over a larger window of requests. This shows that our predictor can handle the absence of inter-core locality without degrading performance.

C. Sensitivity Studies

Effect of CTA Scheduling. We use the widely-used round-robin CTA scheduler to achieve better load balancing of CTAs across cores [11]. However, our proposal should still be effective under different CTA scheduling mechanisms. For example, a CTA scheduler that assigns nearby CTAs on the same core [27] still leaves a significant room to exploit inter-core locality. Figure 15(a) shows the portion of remote hit requests that have CTA distance ≤ 8 (with the nearest supplier core) and above. We observe that for nine out of eleven applications, the portion with CTA distance > 8 is more than 50% of the requests with at least one remote hit. We conclude that even with a CTA scheduler that assigns up to eight consecutive CTAs on the same core, we still have a large scope for inter-core communication to unlock the remote-core bandwidth.

Effect of NoC Resources. Figure 15(b) shows the sensitivity when increasing the NoC resources. We consider three configurations; double the NoC frequency, double the flit size, and double the virtual channels. We also show the results of the baseline NoC used so far (Section III-A), denoted as *Base*. IP(27,S,1) is evaluated under each of them and normalized to the corresponding configuration baseline. First, we observe that our schemes are still beneficial even with double the NoC resources. Second, increasing the number of probes (S) under 27 cores is still not helpful. Third, our schemes benefit the most under double the VCs. This is because searching cores and pushing more probes cause contention at the VC allocator and SW allocator. Thus, doubling the VCs may mitigate the VC allocation contention but at the cost of extra hardware.

Effect of NoC Size. We study the scalability of our schemes using 8×8 mesh and 10×10 mesh under two different configurations. Figure 15(c) shows the IPC and reply bandwidth (both normalized to the configuration mesh baseline) under IP(C%,1,1), where C% represents the percentage of cores to be searched. The used notation in the figure is (number of cores, number of L2/memory partitions). We observe that the IPC follows a similar trend to what we observed using the baseline 6×6 mesh. Specifically, searching 25% or 50% of the cores leads to higher performance in terms of both IPC and reply bandwidth.

Effect of Additional Memory Partitions. Figure 15(c) shows the effect of increasing the number of memory partitions (this increases the total L2 capacity, L2 bandwidth, and memory bandwidth) in the system. For an 8×8 mesh, we study systems with 8 and 16 memory partitions. For a 10×10 mesh, we study systems with 16 and 32 memory partitions. We observe that even with more memory partitions, our proposal enhances IPC due to efficiently unlocking the remote-core bandwidth.

Effect of Core to Memory Partition Ratio. Figure 15(c) studies varying the ratio of core to memory partition count.

We observe that our schemes can boost IPC in all systems. Even in a large (68,32) system, IP(C=25%,1,1) achieves 17% IPC improvement over the baseline 10×10 mesh.

Comparison against a Crossbar-based Baseline. In Figure 15(c), we observe that our schemes perform better than a crossbar-based baseline in terms of both IPC and reply bandwidth under (56,8), (48,16), and (84,16) systems. Under a large (68,32) system, a crossbar-based baseline performs close to, but still not as good as, our schemes. Note that for such large systems, the complexity of the crossbar is high. Also, the performance difference between the mesh-based baseline and the crossbar-based baseline is in line with a simple bisection bandwidth analysis for both systems.[5]

We conclude that our design is robust and can perform well across a wide range of hardware mechanisms and system configurations, such as CTA scheduling policies, L2/memory bandwidth, and core to memory partition ratio. It also outperforms the crossbar-based baseline.

V. Related Work

In this section, we briefly discuss works that are the most relevant to this paper.

Intra-core Locality in GPUs. There is a large body of work that focuses on exploiting the locality that exists within a GPU core [11], [13], [17]–[19], [27], [31], [45]–[52]. In this work, we specifically focus on the locality that exists across cores. Multiple prior CTA schedulers [26], [53]–[56] used different heuristics to exploit the locality across CTAs. However, as shown by prior works [54], [57], [58], there is no single ideal CTA scheduling policy that benefits all applications. This is because inter-CTA locality, data access pattern, and execution time of CTAs are hard to know at compile time, which increases the complexity of the CTA scheduling problem. Hence, we choose the round-robin CTA scheduler as it is the most commonly used. Our analysis shows that the data sharing across L1 caches is pervasive and hence our solutions are effective.

Inter-core Locality in GPUs. Prior works proposed mechanisms to exploit inter-core locality in GPUs by allowing communication between multiple L1s by connecting the cores via a ring NoC [24] or using the L2 cache to forward the read request to a supplier L1 [44]. Other works proposed coherence-like mechanisms [59] to enable communication across L1 caches. Inter-core locality information has also been used to propose a packet coalescing mechanism to reduce NoC pressure [25]. Although these works either identify inter-core locality, propose architectures to enable inter-core communication, or utilize coherence-like mechanisms, they do not provide a way to (1) probe multiple L1 caches in parallel, and (2) identify which L1 caches to probe for high probe success rate. Our schemes allow the inter-core communication to be low-latency due to parallel probes, and low bandwidth-demanding due to the reduced number of useless probes sent. Finally, previous works studied coherence communication predictors based on address [60], [61], instruction [62], or both [63], [64]. These works focused on tracking coherence events at the directories. Our work uses an effective PC-based predictor to filter the read misses that have less probability of sharing across the GPU cores.

[5]For the systems we consider in this paper, the ratio of crossbar bisection bandwidth to 2D mesh bisection bandwidth is equal to the ratio of the number of memory partitions to twice the mesh dimension.

VI. Conclusions

Traditionally, GPUs have been depending on the bandwidth from local/shared caches and memory to achieve high performance. Going forward, other sources of bandwidth need to be explored and leveraged given that the issue of bandwidth is going to be even more critical in large-scale GPU-based systems. Our detailed analysis in this paper shows that remote-core bandwidth can significantly improve the GPU performance within a single GPU node. However, there are several challenges in unlocking this remote-core bandwidth, which this paper systematically addresses. First, we leverage the bi-modal distribution of inter-core locality across PCs to determine which data is expected to be shared across cores. Second, we dynamically generate an inter-core locality map that guides the probing mechanism to determine which cores to probe for increasing the probability of finding the shared data. Finally, we develop a novel two-level probing technique to get the data as soon as possible without saturating the interconnect. We conclude that our efficient inter-core communication provides a significant improvement in performance and on-chip bandwidth at a modest hardware cost.

Acknowledgment

The authors thank Nuwan Jayasena, Jieming Yin, the anonymous reviewers, and members of the Insight Computer Architecture Lab at William & Mary for their feedback. This material is based upon work supported by the National Science Foundation (NSF) grants (#1657336 and #1750667). This work was performed in part using computing facilities at William & Mary. AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

References

[1] TOP500, "Top500 Supercomputer Sites," June 2019. [Online]. Available: http://www.top500.org/lists/2019/06/

[2] A. Eklund, P. Dufort, D. Forsberg, and S. M. LaConte, "Medical Image Processing on the GPU-Past, Present and Future," *Medical Image Analysis Journal*, 2013.

[3] G. Pratx and L. Xing, "GPU Computing in Medical Physics: A Review," *The Journal of Medical Physics Research and Practice*, 2011.

[4] S. S. Stone, J. P. Haldar, S. C. Tsao, W. mei W. Hwu, B. P. Sutton, and Z.-P. Liang, “Accelerating Advanced MRI Reconstructions on GPUs,” *The Journal of Parallel and Distributed Computing*, 2008.

[5] NVIDIA, “How to Harness Big Data for Improving Public Health.” [Online]. Available: http://www.govhealthit.com/news/how-harness-big-data-improving-public-health

[6] I. Schmerken, “Wall street accelerates options analysis with GPU technology,” 2009. [Online]. Available: https://www.hpcwire.com/2009/03/12/wall_street_accelerates_options_analysis_with_gpu_technology/

[7] NVIDIA, “NVIDIA Tesla GPUs Used by J.P. Morgan Run Risk Calculations in Minutes, Not Hours,” 2011. [Online]. Available: https://nvidianews.nvidia.com/news/nvidia-tesla-gpus-used-by-j-p-morgan-run-risk-calculations-in-minutes-not-hours

[8] NVIDIA, “Computational Finance.” [Online]. Available: https://www.nvidia.com/object/computational_finance.html

[9] NVIDIA, “Researchers Deploy GPUs to Build World’s Largest Artificial Neural Network,” 2013. [Online]. Available: https://nvidianews.nvidia.com/news/researchers-deploy-gpus-to-build-world-s-largest-artificial-neural-network

[10] S. I. Park, S. P. Ponce, J. Huang, Y. Cao, and F. Quek, “Low-Cost, High-Speed Computer Vision Using NVIDIAs CUDA Architecture,” in *Proceedings of the Applied Imagery Pattern Recognition Workshop (AIPR)*, 2008.

[11] A. Jog, O. Kayiran, N. C. Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, “OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance,” in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[12] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, “Analyzing CUDA Workloads Using a Detailed GPU Simulator,” in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2009.

[13] A. Jog, E. Bolotin, Z. Guz, M. Parker, S. W. Keckler, M. T. Kandemir, and C. R. Das, “Application-aware Memory System for Fair and Efficient Execution of Concurrent GPGPU Applications,” in *Proceedings of the Workshop on General Purpose Processing Using GPU (GPGPU)*, 2014.

[14] A. Jog, O. Kayiran, T. Kesten, A. Pattnaik, E. Bolotin, N. Chatterjee, S. Keckler, M. T. Kandemir, and C. R. Das, “Anatomy of GPU Memory System for Multi-Application Execution,” in *Proceedings of the International Symposium on Memory Systems (MEMSYS)*, 2015.

[15] B. Wu, Z. Zhao, E. Z. Zhang, Y. Jiang, and X. Shen, “Complexity Analysis and Algorithm Design for Reorganizing Data to Minimize Non-coalesced Memory Accesses on GPU,” in *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2013.

[16] G. Chen, B. Wu, D. Li, and X. Shen, “Porple: An extensible optimizer for portable data placement on gpu,” in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2014.

[17] O. Kayiran, 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 Architecture and Compilation Techniques (PACT)*, 2013.

[18] A. Pattnaik, X. Tang, A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, and C. R. Das, “Scheduling Techniques for GPU Architectures with Processing-In-Memory Capabilities,” in *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2016.

[19] T. G. Rogers, M. O’Connor, and T. M. Aamodt, “Cache-Conscious Wavefront Scheduling,” in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2012.

[20] G. Koo, Y. Oh, W. W. Ro, and M. Annavaram, “Access Pattern-Aware Cache Management for Improving Data Utilization in GPU,” in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2017.

[21] N. Agarwal, D. Nellans, M. O’Connor, S. W. Keckler, and T. F. Wenisch, “Unlocking Bandwidth for GPUs in CC-NUMA Systems,” in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2015.

[22] G. Koo, H. Jeon, and M. Annavaram, “Revealing Critical Loads and Hidden Data Locality in GPGPU Applications,” in *Proceedings of the International Symposium on Workload Characterization (IISWC)*, 2015.

[23] D. Li and T. M. Aamodt, “Inter-Core Locality Aware Memory Scheduling,” *IEEE Computer Architecture Letters (CAL)*, 2016.

[24] S. Dublish, V. Nagarajan, and N. Topham, “Cooperative Caching for GPUs,” *ACM Transactions on Architecture and Code Optimization (TACO)*, 2016.

[25] 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)*, 2017.

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

[27] A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, “Orchestrated Scheduling and Prefetching for GPGPUs,” in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2013.

[28] W. Jia, K. A. Shaw, and M. Martonosi, “MRPB: Memory Request Prioritization for Massively Parallel Processors,” in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2014.

[29] H. Wang, F. Luo, M. Ibrahim, O. Kayiran, and A. Jog, "Efficient and Fair Multi-programming in GPUs via Effective Bandwidth Management," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2018.

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

[31] O. Kayiran, N. C. Nachiappan, A. Jog, R. Ausavarungnirun, M. T. Kandemir, G. H. Loh, O. Mutlu, and C. R. Das, "Managing GPU Concurrency in Heterogeneous Architectures," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2014.

[32] J. Zhan, O. Kayran, G. H. Loh, C. R. Das, and Y. Xie, "OSCAR: Orchestrating STT-RAM Cache Traffic for Heterogeneous CPU-GPU Architectures," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2016.

[33] A. Pattnaik, X. Tang, O. Kayiran, A. Jog, A. Mishra, M. T. Kandemir, A. Sivasubramaniam, and C. R. Das, "Opportunistic Computing in GPU Architectures," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2019.

[34] G. Yuan, A. Bakhoda, and T. Aamodt, "Complexity Effective Memory Access Scheduling for Many-core Accelerator Architectures," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2009.

[35] Cerebras, "Cerebras Wafer Scale Engine," August 2019. [Online]. Available: https://www.cerebras.net/wp-content/uploads/2019/08/Cerebras-Wafer-Scale-Engine-Whitepaper.pdf

[36] 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)*, 2015.

[37] C. Sun, C. H. O. Chen, G. Kurian, L. Wei, J. Miller, A. Agarwal, L. S. Peh, and V. Stojanovic, "DSENT - A Tool Connecting Emerging Photonics with Electronics for Opto-Electronic Networks-on-Chip Modeling," in *Proceedings of the International Symposium on Networks-on-Chip (NOCS)*, 2012.

[38] NVIDIA, "CUDA C/C++ SDK Code Samples," 2011. [Online]. Available: http://developer.nvidia.com/cuda-cc-sdk-code-samples

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

[40] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The Scalable HeterOgeneous Computing (SHOC) Benchmark Suite," in *Proceedings of the Workshop on General Purpose Processing Using GPU (GPGPU)*, 2010.

[41] M. Burtscher, R. Nasre, and K. Pingali, "A Quantitative Study of Irregular Programs on GPUs," in *Proceedings of the International Symposium on Workload Characterization (IISWC)*, 2012.

[42] L.-N. Pouchet, "Polybench: The Polyhedral Benchmark Suite," 2012. [Online]. Available: http://web.cs.ucla.edu/ pouchet/software/polybench/

[43] Hynix, "Hynix GDDR5 SGRAM Part H5GQ1H24AFR Revision 1.0," 2009. [Online]. Available: http://www.hynix.com/datasheet/pdf/graphics/H5GQ1H24AFR(Rev1.0).pdf

[44] 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 (CAL)*, 2017.

[45] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Divergence-Aware Warp Scheduling," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2013.

[46] A. Sethia and S. Mahlke, "Equalizer: Dynamic Tuning of GPU Resources for Efficient Execution," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2014.

[47] D. Li, M. Rhu, D. R. Johnson, O. Mike, M. Erez, D. Burger, D. S. Fussell, and S. W. Redder, "Priority-Based Cache Allocation in Throughput Processors," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2015.

[48] O. Kayiran, A. Jog, A. Pattnaik, R. Ausavarungnirun, X. Tang, M. T. Kandemir, G. H. Loh, O. Mutlu, and C. R. Das, "μC-States: Fine-grained GPU Datapath Power Management," in *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2016.

[49] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving GPU Performance via Large Warps and Two-level Warp Scheduling," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2011.

[50] S.-Y. Lee and C.-J. Wu, "CAWS: Criticality-aware Warp Scheduling for GPGPU Workloads," in *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2014.

[51] U. Milic, O. Villa, E. Bolotin, A. Arunkumar, E. Ebrahimi, A. Jaleel, A. Ramirez, and D. Nellans, "Beyond the Socket: NUMA-Aware GPUs," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2017.

[52] V. Young, A. Jaleel, E. Bolotin, E. Ebrahimi, D. Nellans, and O. Villa, "Combining HW/SW Mechanisms to Improve NUMA Performance of Multi-GPU Systems," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2019.

[53] 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)*, 2014.

[54] 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 International Symposium on Computer Architecture (ISCA)*, 2017.

[55] A. Tabbakh, M. Annavaram, and X. Qian, "Power Efficient Sharing-Aware GPU Data Management," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2017.

[56] L. Wang, X. Zhao, D. Kaeli, Z. Wang, and L. Eeckhout, "Intra-Cluster Coalescing to Reduce GPU NoC Pressure," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2018.

[57] N. Vijaykumar, E. Ebrahimi, K. Hsieh, P. B. Gibbons, and O. Mutlu, "The Locality Descriptor: A Holistic Cross-Layer Abstraction to Express Data Locality In GPUs," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2018.

[58] X. Zhao, A. Adileh, Z. Yu, Z. Wang, A. Jaleel, and L. Eeckhout, "Adaptive Memory-Side Last-Level GPU Caching," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2019.

[59] 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 for High Performance Computing, Networking, Storage and Analysis (SC)*, 2010.

[60] S. S. Mukherjee and M. D. Hill, "Using Prediction to Accelerate Coherence Protocols," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 1998.

[61] A.-C. Lai and B. Falsafi, "Memory Sharing Predictor: The Key to a Speculative Coherent DSM," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 1999.

[62] S. Kaxiras and J. R. Goodman, "Improving CC-NUMA Performance Using Instruction-Based Prediction," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 1999.

[63] M. M. K. Martin, P. J. Harper, D. J. Sorin, M. D. Hill, and D. A. Wood, "Using Destination-Set Prediction to Improve the Latency/Bandwidth Tradeoff in Shared-Memory Multiprocessors," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2003.

[64] S. Kaxiras and C. Young, "Coherence Communication Prediction in Shared-Memory Multiprocessors," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2000.