GPU-Initiated Resource Allocation for Irregular Workloads

Ilyas Turimbetov
iturimbetov18@ku.edu.tr
Koç University
İstanbul, Turkey

Muhammad Aditya Sasongko
msasongko@ku.edu.tr
Koç University
İstanbul, Turkey

Didem Unat
dunat@ku.edu.tr
Koç University
İstanbul, Turkey

ABSTRACT

GPU kernels may suffer from resource underutilization in multi-GPU systems due to insufficient workload to saturate devices when incorporated within an irregular application. To better utilize the resources in multi-GPU systems, we propose a GPU-sided resource allocation method that can increase or decrease the number of GPUs in use as the workload changes over time. Our method employs GPU-to-CPU callbacks to allow GPU device(s) to request additional devices while the kernel execution is in flight. We implemented and tested multiple callback methods required for GPU-initiated workload offloading to other devices and measured their overheads on Nvidia and AMD platforms. To showcase the usage of callbacks in irregular applications, we implemented Breadth-First Search (BFS) that uses device-initiated workload offloading. Apart from allowing dynamic device allocation in persistently running kernels, it reduces time to solution on average by 15.7% at the cost of callback overheads with a minimum of 6.50 microseconds on AMD and 4.83 microseconds on Nvidia, depending on the chosen callback mechanism. Moreover, the proposed model can reduce the total device usage by up to 35%, which is associated with higher energy efficiency.

CCS CONCEPTS

• **Computing methodologies** → **Parallel algorithms**; Parallel programming languages; **Self-organization**; • **Mathematics of computing** → *Graph algorithms.*

ACM Reference Format:
Ilyas Turimbetov, Muhammad Aditya Sasongko, and Didem Unat. 2024. GPU-Initiated Resource Allocation for Irregular Workloads. In *The 3rd International Workshop on Extreme Heterogeneity Solutions (ExHET '24), March 3, 2024, Edinburgh, United Kingdom.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3642961.3643799

1 INTRODUCTION

Irregular applications are difficult to efficiently implement on multi-GPU systems. However, recent advances in GPU hardware and software have allowed performance improvements for these applications [9]. Direct Peer-to-Peer messaging between the devices through NVLink [1, 10, 14] made fine-grained irregular communication inside a single multi-GPU node effective, while staying free from the CPU orchestration [11]. Moreover, it untangles GPU systems from the synchronous model, allowing for asynchronous execution. Together with the concept of persistent kernels [12], where the iteration loop is moved inside the GPU kernel, GPU-sided communication allows to run the entirety of an application on the device [13]. Additionally, authors in [9] show that persistent kernels outperform discrete kernels when the amount of available parallelism per iteration is low.

Despite these advancements, a multi-GPU system may be prone to both underutilization or oversubscription of devices [5] because the workload of irregular applications at every iteration is unpredictable. When the workload is not enough to saturate all the devices for the duration of the execution, dynamic allocation or scheduling can be applied to avoid resource wastes and unnecessary communication overheads. However, in irregular algorithms with low compute intensity (e.g., Breadth-First Search and other graph algorithms) scheduling may be impossible to perform. Moreover, dynamic allocation conflicts with the use of persistent kernels, since all the initially assigned resources in persistent kernels are busy throughout the kernel lifetime. Better resource utilization in persistent kernels is therefore only possible on the device granularity.

This work proposes a methodology to dynamically employ devices on multi-GPU systems both in discrete and persistent kernels. Since it is not possible to directly launch kernels from a GPU kernel on peer devices, we propose a callback-based scheme for GPU-sided offloading. To our knowledge, it is the first method described in literature, for device-sided offloading that works at intra-kernel level. We show that this method suits well for the irregular applications due to their highly unpredictable computation and communication workloads, as it ensures reduced inter-device communication costs, while maintaining the same processing power of continuously using many devices. Moreover, GPU as an offloader model constitutes an important step towards autonomy of GPU devices and frees them from strict reliance on the host for resource management.

We developed three different callback mechanisms based on event waiting on GPU streams (event-based callbacks), CPU thread busy-waiting (busy wait-based callbacks), and non-busy waiting through interrupts (interrupt-based callbacks). We implemented asynchronous BFS using work sharing queues [8] and utilized GPU-initiated offloading with CPU busy-wait callback. The proposed dynamic scheme is compared against usage of fixed device count throughout the execution. The list of contributions this work brings forward is as follows:

- A methodology to dynamically employ devices and a GPU-to-GPU work offloading mechanism for multi-GPU systems.
- Performance analysis of several GPU-to-CPU callback mechanisms on AMD and Nvidia GPUs.
- Example implementation of the proposed offloading mechanism on Nvidia GPUs using BFS.

ExHET '24, March 3, 2024, Edinburgh, United Kingdom
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0537-3/24/03
https://doi.org/10.1145/3642961.3643799

- Performance analysis of BFS with dynamic device allocation in discrete and persistent kernels.

The proposed offloading scheme shows an average 15.7% and 7.3% improvement in terms of time to solution on persistent and discrete kernels, respectively. The resource utilization is improved in 8 out of 13 configurations on persistent kernels and 16 out of 18 on discrete kernels.

2 RELATED WORK

To our knowledge, there is no prior work focusing on GPU-initiated resource management. Thus, we will explore related literature that influenced the components of this work.

Dynamic Resource Allocation. Though dynamic resource allocation is widely-used in cloud computing [4, 27], there have been few works for heterogeneous systems. Park et al. [16] developed a technique that dynamically partitions GPUs among multiple kernels by allocating one or more streaming multiprocessors (SM) to a kernel. Vaishnav et al. [24] developed a resource-elastic scheduling algorithm that schedules workload on FPGAs. Their scheduler can determine the types of devices, i.e. CPUs or FPGAs, number of compute units, and the types of accelerators that need to be allocated to each task. Mandal, et al. [15] propose techniques that manage the resource allocation of heterogeneous system-on-chips at runtime. Different from the existing works, we dynamically allocate computational resources in our scheme by adding or removing GPU devices as needed.

Persistent kernels. Authors in [12] discuss potential features that can be gained by usage of persistent kernels, such as in-kernel device synchronization, better load balancing and kernel fusion. An example of uberkernel resulting from fusion of multiple stages of a GPU application is proposed in WhippleTree [19] and [23]. The benefit of maintaining the device state is exemplified in [28], where efficient usage of on-chip memory led to substantial speedup over non-persistent implementations on a single-GPU. The CPU-free execution model is introduced in [13] for iterative solvers running on multi-GPUs, where authors eliminated the CPU involvement in both data and control paths of the program execution.

Load balancing and GPU-initiated Communication. Gunrock [26] is a graph analytics library that can execute on multi-GPUs. However, it does not fully utilize the GPU-sided communication. Additionally, it follows the BSP model, which suffers from intra-GPU load imbalance due to device-wide synchronization of unevenly occupied threads. Groute [6] however, uses the NVLink interconnect and takes load balancing one step further by introducing asynchronous execution with the help of the work queue. Yet, Groute still uses CPU resources for scheduling and initiating the GPU-to-GPU communications. This CPU involvement is potentially the most important bottleneck of the model. ATOS [8] removes this bottleneck from the model, introducing in-kernel communication. Similarly to Groute [6] and Whippletree [19], it features a work queue for better load balancing. For multi-GPU execution [8], ATOS uses intra-kernel communication over NVLink. It does not require any CPU involvement, since a device directly pushes work to remote devices.

GPU-to-CPU callbacks. In [20], for the first time, authors called for the introduction of GPU-to-CPU callbacks and motivated addition of such a feature to GPU devices with several use-cases such as debugging. Their solution is based on CPU polling to wait for the notifications. Moreover, there have been techniques that enable GPUs to leverage OS system calls in CPUs to perform filesystem I/O operations [7, 17] and to create socket/channel for networking [18]. Vesely et al. [3, 25] developed a framework for GPU-sided system call invocation that uses POSIX APIs. It enables GPUs to send interrupts to invoke OS kernel-level services from CPUs. By leveraging interrupts, this approach does not keep CPU threads busy-waiting for notifications from GPUs. Sun et al. [21] extended AMD ROC platform to allow GPUs to assign computation tasks to CPUs. Tomoutzoglou et al. [22] proposes a hardware, packet-based communication scheme that allows GPUs to notify CPUs and offload jobs to CPUs at user space without involving costly system calls.

Existing work on CPU callbacks and system calls lacks a crucial capability: the initiation of new kernels from one GPU to others. While CUDA dynamic parallelism enables such launches within the same device, it is limited to a subset of active threads. Additionally, GPUs cannot utilize additional devices beyond those already in use. Enabling a GPU device to notify a CPU for launching kernels on other devices allows unused GPUs to remain inactive until needed. Moreover, utilizing interrupts for CPU notifications from GPUs eliminates the need for CPU polling, improving efficiency.

3 METHODOLOGY

Figure 1 shows the overall workflow of the proposed GPU offloading model. It starts with an example workload in an irregular application. Stage *a* refers to the phase with low workload and only one device in use. During stage *b* the workload increases thus can be split between all the available devices. In stage *c*, the workload shrinks thus all the devices delegate the remaining work to a smaller number of GPUs. Although the example shows a pattern with a single high workload section, some applications can have multiple or be completely unpredictable. In such cases stage *c* can be followed by stage *b* repeatedly until the execution is completed. The dashed line between the stages represents a decision-making process.

To start execution on a single device, all the application data has to be available, as shown in Figure 1B. It ensures that while the workload is not enough to occupy the whole GPU, the communication costs are reduced, as the work items are not yet being shared with other devices; all work items are processed by GPU0 as shown in Figure 1C *a*. If the workload exceeds a certain threshold, it would mean that usage of additional GPUs would improve the overall running time at the cost of inter-device communication. The threshold depends on a specific scenario. In our case, we use a fixed threshold, so that when a device has more work items than maximum concurrent threads, the threshold is exceeded. As shown in Figure 1C *b*, every GPU sends work items to their corresponding oweners' work queues. In case, workload shrinks again (Figure 1C *c*), queues of inactive devices are emptied, all remaining items are sent to GPU0.

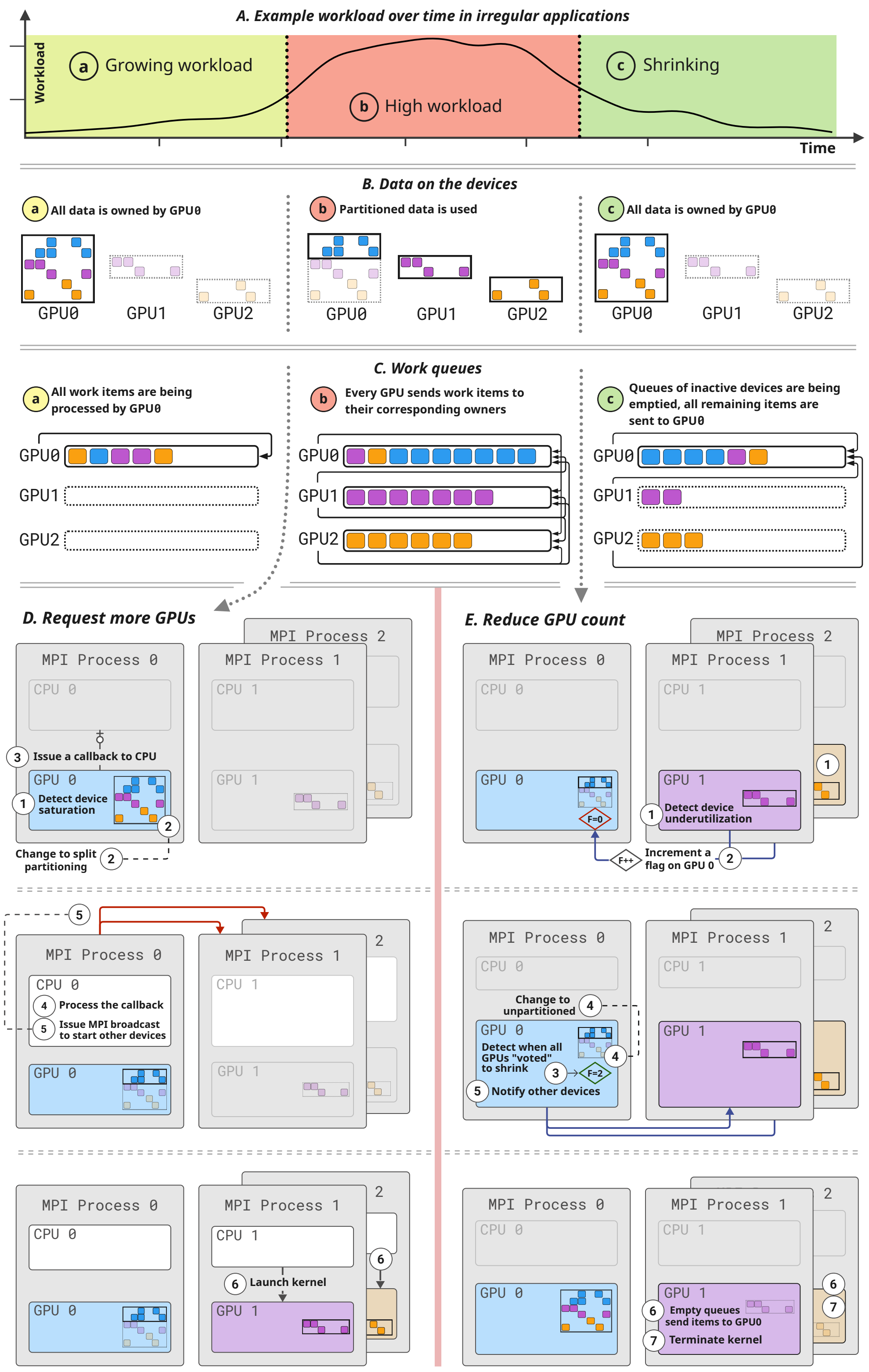
A. Example workload over time in irregular applications
Workload
a Growing workload
b High workload
c Shrinking
Time
B. Data on the devices
a All data is owned by GPU0
b Partitioned data is used
c All data is owned by GPU0
GPU0
GPU1
GPU2
C. Work queues
a All work items are being processed by GPU0
b Every GPU sends work items to their corresponding owners
c Queues of inactive devices are being emptied, all remaining items are sent to GPU0
D. Request more GPUs
MPI Process 0
MPI Process 1
MPI Process 2
CPU 0
CPU 1
GPU 0
GPU 1
3 Issue a callback to CPU
1 Detect device saturation
2 Change to split partitioning
4 Process the callback
5 Issue MPI broadcast to start other devices
6 Launch kernel
E. Reduce GPU count
1 Detect device underutilization
2 Increment a flag on GPU 0
F=0
F++
Change to unpartitioned 4
Detect when all GPUs "voted" to shrink 3
F=2
5 Notify other devices
6 Empty queues send items to GPU0
7 Terminate kernel

Figure 1: Workflow of the proposed model.

3.1 Offloading for Discrete Kernels

The basic scenario, which allows dynamic allocation of GPU resources is when every algorithm iteration corresponds to a separate kernel launch. This so-called *discrete kernel* model is used to be a standard mode of execution. This model assumes that in every algorithm iteration, the CPU would launch a GPU kernel that only handles the computation, followed by a data movement step, also initiated on the CPU side. The fact that the application execution path goes through the CPU after each iteration, despite the overheads, can be used in irregular applications for efficient allocation of GPU resources. Typically, in a graph algorithm the workload of the next iteration is known after a current one has been completed. Therefore, the exact amount of needed threads can be allocated by the CPU for each discrete kernel launch.

We can extend the resource allocation approach to consider the number of GPUs in use. For that, each GPU would need to have the partitions for each potential GPU count that can be used in memory. The CPU can then determine, based on the workload, the number of GPUs to be used and instruct the devices to employ the corresponding data partitioning. Although this incurs a memory cost for the allocation, it allows fine-tuning the number of GPUs in use at every iteration, similar to adjusting the number of GPU threads.

3.2 Offloading for Persistent Kernels

A *persistent kernel* is long running kernel where the iteration loop is moved to the device code from the host code. Unlike the discrete kernels, the persistent kernels are challenging due to the absence of direct device-initiated invocation of GPU kernels on peer devices. We therefore use CPU-initiated kernel invocation through signaling, while attempting to keep CPU usage minimized. Because of the differences in hardware and the corresponding instruction set, the signaling mechanism varies.

Regardless of the underlying mechanism, the idea behind this approach is to signal the CPU to employ additional devices. We allocate an MPI process for each GPU device used. As shown in Figure 1D, in *1* the device detects it is saturated thus it needs to offload some of its workload to its peers. In *2*, it then switches to split partitioning to start filling up other devices' queues and issues a callback to the CPU *3*. After a callback has been received by the host *4*, there is an MPI signaling step *5* to inform other devices about the start of the execution, which can be avoided if all the devices are managed within a single process. Step *6* launches the kernels on other devices.

Reducing the number of devices in use (Figure 1E) requires all devices to reach a low-workload phase. This step does not require CPU involvement, because now it is the GPUs' responsibility to coordinate the shrinking. When underutilization is detected *1*, devices notify GPU0 *2* by updating a flag to make a common decision *3* to send all the remaining work items to GPU0. Without a coordinated decision, it is possible that a device would send work items to an already terminated device. Therefore data partitioning needs to be changed on all devices, starting with GPU0 *4*, followed by other devices afterwards *5*. Potentially, a model with completely dynamic number of devices is possible, but would require either higher memory consumption to store additional data on devices or communication overheads to send such data in case the owner decides to terminate. Such costs may be tolerable only in compute-intensive applications.

3.3 GPU-to-CPU Callback

We devise three callback mechanisms to facilitate inter-GPU computation offloading as illustrated in Figure 2. While the first callback mechanism could only work on discrete kernels, the second and third callback mechanisms could work on both discrete and persistent kernels. In the first mechanism, referred as *event-based callback* (Figure 2A), a GPU stream records an event, i.e. a CUDA event or a HIP event, after its kernel finishes running. By recording this event, the stream causes the other GPU streams that wait on the event to launch additional kernels to unused GPUs. In the second mechanism, referred as *busy wait-based callback* (Figure 2B), a CPU thread busy-waits on a globally accessible memory region until a GPU writes a signaling value to the address being busy-waited [20]. The third mechanism, named as *interrupt-based callback* (Figure 2C), leverages interrupt delivery from device to host on AMD GPUs. However, such a functionality is not available to end-users on Nvidia GPUs. Even though we utilize these callback mechanisms for CPU to launch new kernels on other devices, these mechanisms are general and can be used to signal the host to do other tasks.

Event-based callbacks. In Nvidia and AMD GPUs, we implement event-based callbacks that work on discrete kernels. These callbacks leverage `cudaStreamWaitEvent` in Nvidia GPUs and `hipStreamWaitEvent` in AMD GPUs. These functions work in conjunction with `cudaEventRecord` and `hipEventRecord`, respectively, to enable a discrete kernel to launch kernels to other GPU devices by having its stream record an event after its completion.

Busy-wait callbacks. This callback employs busy-wait in CPUs by leveraging CUDA and HIP's zero-copy memory to detect modification of flags in persistent kernels. After launching a kernel, a CPU thread busy-waits on a flag. Once the kernel changes the value of the flag using a zero-copied pointer of the flag, the CPU thread detects the change and launches kernels to other GPUs.

This callback mechanism, however, keeps CPU resources busy by polling on a shared memory region to wait for notifications from GPUs. Yet, due to the lack of any alternative solutions, we can use only this mechanism on persistent kernels running on Nvidia GPUs. One possible alternative for busy wait in CPUs is by having CUDA streams running on CPUs wait on an update by GPUs to a memory location by using `cuStreamWaitValue32` or `cuStreamWaitValue64`. After the memory location is updated, the streams can launch kernels to unused GPUs. Though the mechanism is similar to busy-wait by CPU threads, it might incur lower latency due to CUDA's built-in support. However, we cannot implement this mechanism because the support for stream memory operation is disabled by default and there is a lack of documentation on how to use it.

Interrupt-based callbacks. We implement this type of callbacks only in AMD GPUs since AMD has a documented instruction, i.e. `s_sendmsg`, that allows GPUs to send interrupts to CPUs. After receiving an interrupt from a GPU, the interrupt handler in the OS kernel sends a signal to a CPU thread in the user space. Once

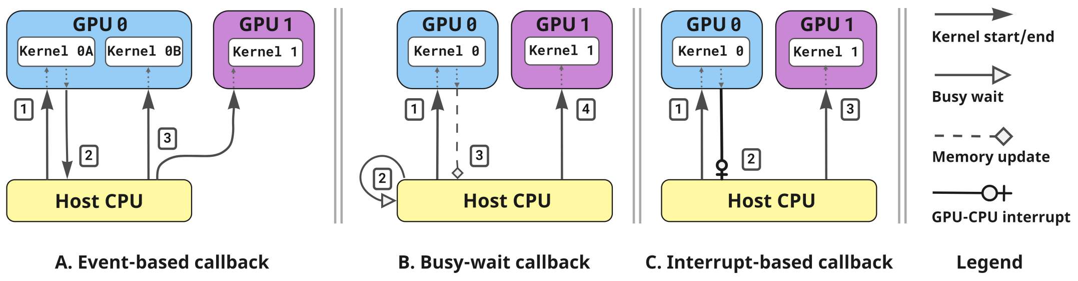

Figure 2: Callback methods to launch a kernel on another GPU

Name	Vertices	Edges	Depth	Reference
uk-2005	39.5M	936.4M	200	SuiteSparse
it-2004	41.3M	1,150M	61	SuiteSparse
webbase-2001	118M	1,012M	623	SuiteSparse
hgg_15m_ud	15M	889M	689	KaGen
hgg_20m_ud	20M	791M	145	KaGen
rmat_100m_200m_di	100M	200M	67	PaRMAT
rmat_90m_450m_di	90M	450M	168	PaRMAT

Table 1: List of the graphs used in experiments. Graphs from KaGen and PaRMAT are generated.

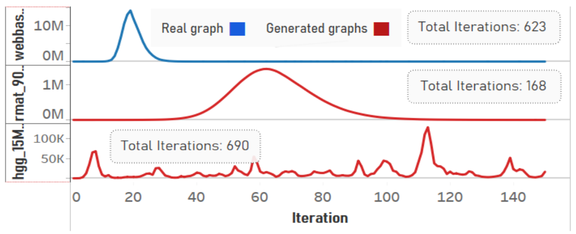

Figure 3: Frontier size at every BFS iteration for three of the graphs. Note that the Y-axis is non-uniform.

receiving the signal, the CPU thread runs a signal handler that launches computation kernels to other idle GPU devices.

4 EVALUATION

This section evaluates the performance of the different call-back mechanisms and proposed GPU-initiated resource allocation on BFS. Table 1 shows the list of graphs used in the evaluation. All graphs are randomly partitioned.

Experiments were conducted on two machines. First machine has 4 V100 NVIDIA GPUs connected to a 2-socket Intel Xeon Gold 6148 CPUs with 20 physical cores/socket. The other machine has 8 A100 NVIDIA GPUs connected to a 2-socket AMD EPYC 7742 CPUs with 64 physical cores/socket. For callback latency, we also carried out experiments in a machine with 2 AMD Instinct MI100 GPUs, which are connected to a 2-socket AMD EPYC 7313 CPUs with 16 logical cores/socket. The measurements are performed five times on all setups.

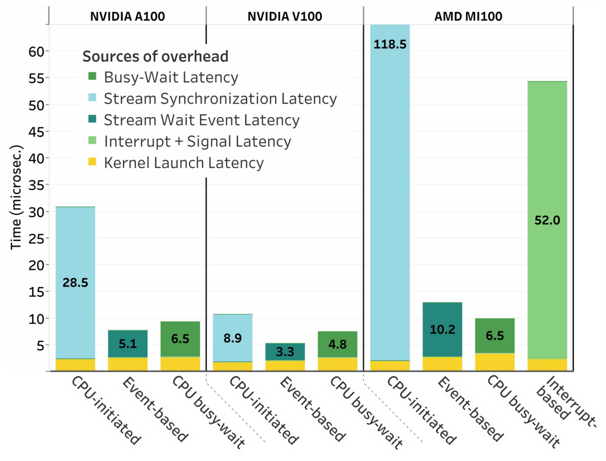

Figure 4: Callback Latency (lower is better)

4.1 Callback Latency

We compare the overhead of callback implementations against a baseline that performs a `cudaDeviceSynchronize` call after a discrete GPU kernel launch. We label our baseline as *CPU-initiated*. The GPU kernels of the *CPU-initiated* and *event-based* schemes in this experiment are empty kernels, while the kernel of the *CPU busy-wait* scheme does nothing other than updating the zero-copied flag that is busy-waited by a CPU thread. For *CPU-initiated*, we measure the latency from the moment the kernel is launched until after the stream synchronization finishes. For *event-based*, the latency is from the moment the kernel is launched in stream 0 until the moment `cudaStreamWaitEvent` in stream 1 detects the event recorded in stream 0. For *CPU busy-wait*, the latency is from kernel launch until the moment an update to the zero-copied flag is detected by the busy-waiting CPU thread. The GPU kernel in the *interrupt-based* callback code does nothing other than sending a software interrupt to the OS by using the `s_sendmsg`[2] instruction. We measure the latency of the interrupt-based callback code from the moment the kernel is launched until the moment the OS signal sent by the interrupt handler is detected by the CPU thread.

Figure 4 displays the latency of the callback schemes on NVIDIA A100 and V100 GPUs. *CPU busy-wait* and *event-based* mechanisms outperform the baseline consistently across different GPUs. Furthermore, *CPU busy-wait* is slower than the *event-based* callback.

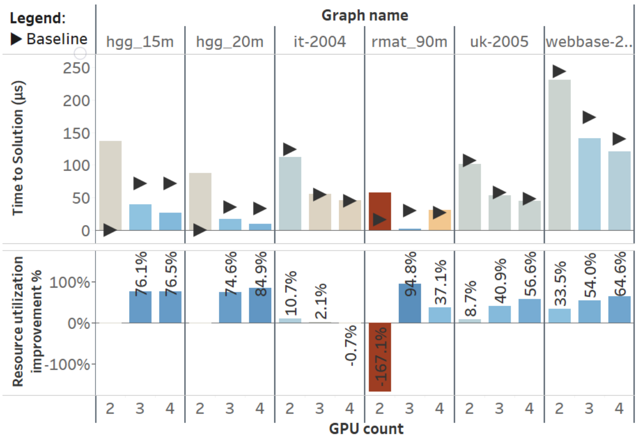

Figure 5: Top: BFS time to solution (lower is better). Bottom: Resource utilization improvement in percentage (positive is better) in discrete kernels

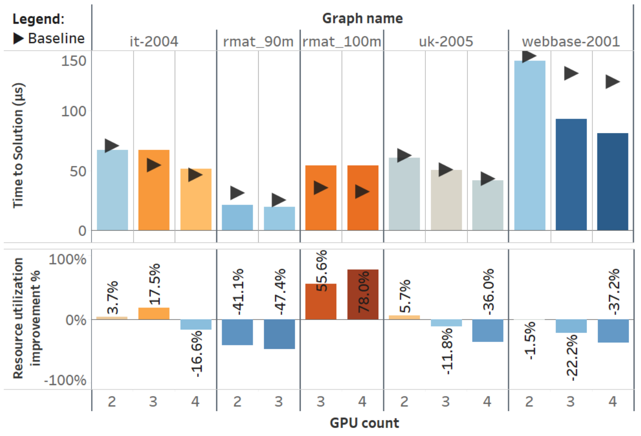

Figure 6: Top: BFS time to solution (lower is better). Bottom: Resource utilization improvement in percentage (positive is better) in persistent kernels

Figure 4 also presents the latency of our callback implementations on AMD MI100 GPUs. All of the three callback mechanisms outperform the baseline. Among three mechanisms, *interrupt-based* callback incurs the highest latency, potentially caused by the extra time lag between interrupt delivery and interrupt handling, in addition to the time lag between signal delivery in the interrupt handler and the signal handling by the host thread. Differently than the results from the NVIDIA GPUs, *event-based* callback has slightly higher latency than *CPU busy-wait*. Additionally, a lower callback latency can be observed on an older generation NVIDIA V100. As previous research indicates [29], in NVIDIA V100 the `cudaDeviceSynchronize` latency has increased compared to the older NVIDIA P100. This trend seems to continue with the newer generation NVIDIA A100, likely due to the increasing number of SMs.

4.2 BFS Performance

We compare our BFS implementation that uses GPU-initiated offloading against ATOS [9]. BFS in ATOS is a state-of-the-art multi-GPU implementation that outperforms the prior-art from Groute[5] and Gunrock [26]. Our implementation is also based on ATOS but capable of dynamically adjusting the devices in use through callbacks to offload work to other GPUs. Since interrupts are not openly available on Nvidia GPUs, we choose to use the CPU busy-waiting callback. In our evaluation we consider two metrics: *performance* and *resource utilization*. Performance will be indicated by the time to solution (when all the GPUs complete their respective kernels). The resource utilization metric, reported as a percentage, is reflected by the total GPU time of all the devices in a system. The second metric is an indication of improvement in energy usage.

We provide the experiment results for input graphs using 2, 3, and 4 GPUs for discrete and persistent kernels on Figure 5 and Figure 6, respectively. The performance of baseline is shown as a black triangle. In discrete kernels, dynamic allocation of GPUs during kernel invocation yields better performance in 14 out of 18 data points, with an average 7.3% improvement. The resource efficiency is improved in 16 out of 18 data points, with an average 35% improvement.

In persistent kernels, dynamic allocation of GPUs during kernel invocation yields better performance in 9 out of 13 data points, with an average 15.7% improvement. The resource utilization is improved in 8 out of 13 data points, with an average 4.1%. Note that the unreported result for certain graphs are due to ATOS (baseline) not being able to produce correct solution. While in discrete kernels a device can terminate easily after each BFS iteration, in persistent kernel resource utilization can be worse due to the overhead introduced by the need to wait for the NVSHMEM operations to complete. Overall, results for both figures show that dynamic allocation of GPUs works best in scenarios where the high workload phase is characterized by a large amount of parallelism, while the low-workload phase spans a large amount of iterations. An example is shown on Figure 4, where webbase-2001 is the graph that most clearly manifests both of the required features.

5 CONCLUSION

We have implemented device-sided work offloading in multi-GPU systems for both discrete and persistent kernels. In contrast to existing approaches, our scheme dynamically allocates computational resources by adding or removing GPU devices as needed. We have explored and assessed three distinct callback mechanisms for notifying the CPU to launch kernels on additional devices. On AMD GPUs, incorporating additional devices is achievable through an interrupt-based callback, effectively eliminating busy-waiting from the offloading process. Furthermore, multi-GPU systems can gain both performance and improved resource utilization advantages from dynamically assignable devices in both discrete and persistent kernels, as demonstrated in the case of BFS. However, the performance improvement is limited for BFS because it exhibits an extreme case with very low computational work. We anticipate that other dynamic algorithms with higher compute intensity would benefit more from dynamic resource allocation. Moreover, enabling

GPUs to seamlessly launch kernels on other devices could mitigate some of the callback overheads.

ACKNOWLEDGMENTS

This work was supported in part by the Royal Society-Newton Advanced Fellowship and by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 949587).

REFERENCES

[1] Palwisha Akhtar, Erhan Tezcan, Fareed Mohammad Qararyah, and Didem Unat. 2021. ComScribe: Identifying Intra-node GPU Communication. In *Benchmarking, Measuring, and Optimizing*, Felix Wolf and Wanling Gao (Eds.). Springer International Publishing, Cham, 157–174.

[2] AMD. 2020. *"AMD Instinct MI100" Instruction Set Architecture Reference Guide.* AMD.

[3] Arkaprava Basu, Joseph L. Greathouse, Guru Venkataramani, and Ján Veselý. 2018. Interference from GPU System Service Requests. In *2018 IEEE Int'l Symposium on Workload Characterization (IISWC)*. 179–190.

[4] Leander Beernaert, Miguel Matos, Ricardo Vilaça, and Rui Oliveira. [n. d.]. Automatic Elasticity in OpenStack. In *Proceedings of the Workshop on Secure and Dependable Middleware for Cloud Monitoring and Management* (Montreal, Quebec, Canada) *(SDMCMM '12)*. ACM, New York, NY, USA, Article 2, 6 pages.

[5] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. 2017. Groute: An Asynchronous Multi-GPU Programming Model for Irregular Computations. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Austin, Texas, USA) *(PPoPP '17)*. ACM, New York, NY, USA, 235–248.

[6] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. 2017. Groute: An Asynchronous Multi-GPU Programming Model for Irregular Computations. *SIGPLAN Not.* 52, 8 (jan 2017), 235–248. https://doi.org/10.1145/3155284.3018756

[7] Shai Bergman, Tanya Brokhman, Tzachi Cohen, and Mark Silberstein. 2017. SPIN: Seamless Operating System Integration of Peer-to-Peer DMA Between SSDs and GPUs. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 167–179.

[8] Yuxin Chen, Benjamin Brock, Serban Porumbescu, Aydın Buluç, Katherine Yelick, and John D. Owens. 2022. Atos: A Task-Parallel GPU Scheduler for Graph Analytics. In *Proceedings of the International Conference on Parallel Processing (ICPP 2022)*. arXiv:2112.00132

[9] Yuxin Chen, Benjamin Brock, Serban Porumbescu, Aydın Buluç, Katherine Yelick, and John D. Owens. 2022. Scalable irregular parallelism with GPUs: Getting CPUs out of the way. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '22)*.

[10] D. Foley and J. Danskin. 2017. Ultra-Performance Pascal GPU and NVLink Interconnect. *IEEE Micro* 37, 2 (2017), 7–17.

[11] Taylor Groves, Ben Brock, Yuxin Chen, Khaled Z. Ibrahim, Lenny Oliker, Nicholas J. Wright, Samuel Williams, and Katherine Yelick. [n. d.]. Performance Trade-offs in GPU Communication: A Study of Host and Device-initiated Approaches. In *2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*. 126–137.

[12] Kshitij Gupta, Jeff A. Stuart, and John D. Owens. 2012. A study of Persistent Threads style GPU programming for GPGPU workloads. In *2012 Innovative Parallel Computing (InPar)*. 1–14. https://doi.org/10.1109/InPar.2012.6339596

[13] Ismayil Ismayilov, Javid Baydamirli, Doğan Sağbili, Mohamed Wahib, and Didem Unat. 2023. Multi-GPU Communication Schemes for Iterative Solvers: When CPUs Are Not in Charge. In *Proceedings of the 37th International Conference on Supercomputing* (Orlando, FL, USA) *(ICS '23)*. Association for Computing Machinery, New York, NY, USA, 192–202. https://doi.org/10.1145/3577193.3593713

[14] A. Li, S. Song, J. Chen, J. Li, X. Liu, N. R. Tallent, and K. J. Barker. 2020. Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. *IEEE Transactions on Parallel and Distributed Systems* 31, 01 (jan 2020), 94–110.

[15] Sumit K. Mandal, Umit Y. Ogras, Janardhan Rao Doppa, Raid Z. Ayoub, Michael Kishinevsky, and Partha P. Pande. 2020. Online Adaptive Learning for Runtime Resource Management of Heterogeneous SoCs. In *Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference* (Virtual Event, USA) *(DAC '20)*. IEEE Press, Article 176, 6 pages.

[16] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. 2017. Dynamic Resource Management for Efficient Utilization of Multitasking GPUs. *SIGPLAN Not.* 52, 4 (apr 2017), 527–540. https://doi.org/10.1145/3093336.3037707

[17] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. 2014. GPUfs: Integrating a File System with GPUs. *ACM Trans. Comput. Syst.* 32, 1, Article 1 (feb 2014), 31 pages.

[18] Mark Silberstein, Sangman Kim, Seonggu Huh, Xinya Zhang, Yige Hu, Amir Wated, and Emmett Witchel. 2016. GPUnet: Networking Abstractions for GPU Programs. *ACM Trans. Comput. Syst.* 34, 3, Article 9 (sep 2016), 31 pages. https://doi.org/10.1145/2963098

[19] Markus Steinberger, Michael Kenzel, Pedro Boechat, Bernhard Kerbl, Mark Dokter, and Dieter Schmalstieg. 2014. Whippletree: Task-Based Scheduling of Dynamic Workloads on the GPU. *ACM Trans. Graph.* 33, 6, Article 228 (nov 2014), 11 pages. https://doi.org/10.1145/2661229.2661250

[20] Jeff A. Stuart, Michael Cox, and John D. Owens. 2010. GPU-to-CPU Callbacks. In *Proceedings of the 2010 Conference on Parallel Processing* (Ischia, Italy) *(Euro-Par 2010)*. Springer-Verlag, Berlin, Heidelberg, 365–372.

[21] Yifan Sun, Saoni Mukherjee, Trinayan Baruah, Shi Dong, Julian Gutierrez, Prannoy Mohan, and David Kaeli. 2018. Evaluating Performance Tradeoffs on the Radeon Open Compute Platform. In *2018 IEEE Int'l Symposium on Performance Analysis of Systems and Software (ISPASS)*. 209–218.

[22] Othon Tomoutzoglou, Dimitris Mbakoyiannis, George Kornaros, and Marcello Coppola. 2020. Efficient Job Offloading in Heterogeneous Systems Through Hardware-Assisted Packet-Based Dispatching and User-Level Runtime Infrastructure. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 5 (2020), 1017–1030.

[23] Stanley Tzeng, Anjul Patney, and John D. Owens. 2010. Task Management for Irregular-Parallel Workloads on the GPU. In *Proceedings of the Conference on High Performance Graphics* (Saarbrucken, Germany) *(HPG '10)*. Eurographics Association, Goslar, DEU, 29–37.

[24] Anuj Vaishnav, Khoa Dang Pham, and Dirk Koch. 2019. Heterogeneous Resource-Elastic Scheduling for CPU+FPGA Architectures. In *Proceedings of the 10th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies* (Nagasaki, Japan) *(HEART 2019)*. ACM, New York, NY, USA, Article 1, 6 pages. https://doi.org/10.1145/3337801.3337819

[25] Ján Veselý, Arkaprava Basu, Abhishek Bhattacharjee, Gabriel H. Loh, Mark Oskin, and Steven K. Reinhardt. 2018. Generic System Calls for GPUs. In *Proceedings of the 45th Annual International Symposium on Computer Architecture* (Los Angeles, California) *(ISCA '18)*. IEEE Press, 843–856.

[26] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. 2016. Gunrock: A High-Performance Graph Processing Library on the GPU. *SIGPLAN Not.* 51, 8, Article 11 (feb 2016), 12 pages. https://doi.org/10.1145/3016078.2851145

[27] Jinyu Yu, Dan Feng, Wei Tong, Pengze Lv, and Yufei Xiong. 2021. CERES: Container-Based Elastic Resource Management System for Mixed Workloads. In *50th International Conference on Parallel Processing* (Lemont, IL, USA) *(ICPP 2021)*. ACM, New York, NY, USA, Article 13, 10 pages.

[28] Lingqi Zhang, Mohamed Wahib, Peng Chen, Jintao Meng, Xiao Wang, and Satoshi Matsuoka. 2022. Persistent Kernels for Iterative Memory-bound GPU Applications. https://arxiv.org/abs/2204.02064

[29] Lingqi Zhang, Mohamed Wahib, Haoyu Zhang, and Satoshi Matsuoka. 2020. A study of single and multi-device synchronization methods in Nvidia GPUs. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 483–493.