CPUs as Co-processors of GPUs: Running GPGPU Applications at the Full Speed with PullKernels

[Position Paper]

Yusuke Suzuki
Keio University
yusuke.suzuki
@sslab.ics.keio.ac.jp

Hiroshi Yamada
TUAT
hiroshiy@cc.tuat.ac.jp

Shinpei Kato
The University of Tokyo
shinpei@pf.is.s.u-tokyo.ac.jp

Kenji Kono
Keio University
kono@ics.keio.ac.jp

1. INTRODUCTION

Leveraging massively parallel computing capabilities of graphics processing units (GPUs) for general-purpose computing (GPGPU) is widely accepted in various applications (apps). GPGPU is applicable to various apps such as deep learning [1, 4, 10, 18], scientific simulations [15, 22], file systems [25], complex control systems [12, 20], autonomous vehicles [8,16], network systems [5,9], web servers [2], key-value stores [7], and databases [6, 11, 13, 21].

Deriving substantial performance benefits of GPGPU is non trivial. We need to elaborate multi-stage pipelines in the GPGPU apps to overlap data transfers to and from GPUs and their computation. In addition, we carefully adjust performance-related factors, such as the size of data chunks, the data transfer intervals, and the pipeline length, to the underlying GPU characteristics. These engineering efforts, almost all of which are app-specific, are not one-shot. Since GPU's performance characteristics are sensitively defined by its generation and internal resources such as the memory size and core clock, developers have to conduct these efforts to their apps on all the target type of GPUs. For example, SSLShader [9] requires a heroic effort to overlap networking, data transfer, and computation to maximize its packet processing performance.

The accelerator-centric app design [14, 23, 24, 26, 27] is emerging and promising to mitigate the above engineering hurdles. The philosophy behind the current accelerator centric design is of *"making GPUs as the same-level computational unit as CPUs"*. The accelerator-centric runtime systems allow GPU kernels to access host-side resources, such as disks and networks, via remote procedure call- (RPC-) based APIs where GPU- and host-side mechanisms interact with each other. Since the GPU kernels request host resources by themselves on demand, the CPU-side app code is released from the laborious efforts such as complex pipelining and asynchronous data copy. For example, GPUfs [24] and GPUnet [14] provides file and RDMA socket interfaces to GPU kernels, respectively. GPUpIO [27] achieves I/O-driven preemption in GPU kernels by instrumenting code with save and restore procedures.

However, existing accelerator centric approaches still remain two big hurdles to fully obtain GPU benefits: *effective GPU utilization* and *efficient accesses to diverse host resources*. Some runtime systems waste the GPU computing resources due to the high latency of the host-device interaction. During accessing host resource requested by a GPGPU app, the GPU code commonly polls the completion of the host-device RPC and thus any GPU tasks cannot be executed. For example, GPUfs- and GPUnet-based apps poll I/O completion to avoid costly GPU kernel launches. The runtime systems also carry out the host resource accesses of the GPU kernels in an ad-hoc manner. GPUfs and GPUnet offer the specific mechanisms to access the target host resources, namely files and RDMA sockets, respectively. These are therefore not applicable to other host resources such as other accelerators and operating systems (OS) services. Although GLoop [26] exposes general interfaces to access host resources, interaction between the GPU and CPU is inefficient; GLoop wastes the GPU computing resources by polling on the completion of the host-device RPC and the app developers have to carefully tune performance-related factors described above.

We argue that we can overcome these limitations by embracing a new concept, *GPU centric design* that pushes forward the accelerator centric design, *"using CPUs as co-processors of GPUs"*. This position paper proposes *PullKernels* that allows the GPU kernels to utilize various host resources, including CPUs, memory, and OS services, as co-processors in an efficient manner. To accelerate GPGPU apps as much as possible, PullKernels effectively utilizes the GPU without polling the CPU request completion. PullKernels also offers general interfaces for accessing host resources and systematically performs efficient communication between GPU- and host-side mechanisms.

PullKernels, running on the current hardware and technology stack, is tackling two key challenges; how do we avoid wasteful GPU utilization? and how do we achieve efficient accesses to the host resource without developer's involvement? To address these issues, PullKernels introduces two techniques, (1) *host latency hiding* and (2) *adaptive host operation batching*. The host latency hiding is a technique to hide high latency of the host-device interaction in a GPU centric app. When issuing host-side operations, the PullKernels runtime system retires the current thread block (TB) and runs other ready TB, instead of polling the completion of the host operation.

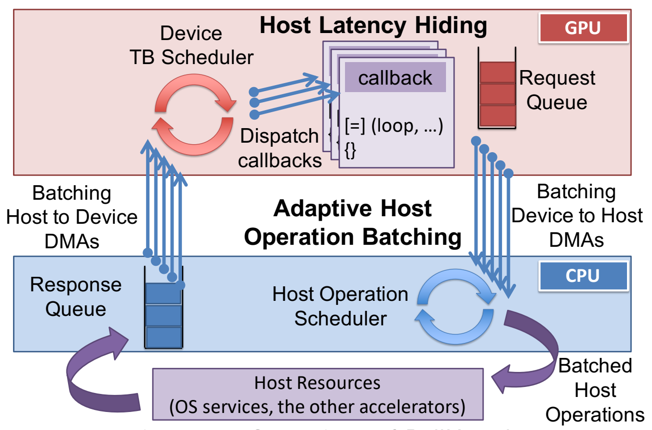

Figure 1: Overview of PullKernels.

To achieve efficient accesses to host resources without developer's involvement, we introduce adaptive host operation batching. A GPU centric app can request numerous host operations from many TBs. Furthermore, since the host latency hiding increases the number of the running TBs, more opportunities for the TBs to issue host resource accesses are given. As a result, the host-side mechanism receives multiple requests of the host operations at a time. PullKernels batches data transfers of these operations to amortize the cost of each Direct Memory Access (DMA). The size of the request queue is adaptively adjusted by the speed of GPU computation and the host-device connection.

2. DESIGN

The design of PullKernels runtime system is based on GLoop [26] that provides an *event-driven programming model* to GPGPU apps. In this model, PullKernels treats host resource requests as *events*, and PullKernels-based apps register their own *callbacks* of events of interest. PullKernels dispatches a callback when the corresponding event has completed. PullKernels-based apps can issue the requests in a non-blocking manner.

Different from GLoop, PullKernels offers *host latency hiding* and *adaptive host operation batching* to enhance GPU centric apps. Fig. 1 shows an overview of PullKernels. The PullKernels runtime system mainly consists of two mechanisms: Device TB Scheduler and Host Operation Scheduler, which run on the GPU and CPU respectively. While a host resource access from the TB is being processed, the PullKernels-based app can proceed simultaneously, instead of polling its completion. The device TB scheduler dispatches other ready TBs from the host access waiting TBs. The host operation scheduler, running on the CPU, batches the host resource requests involving data transfer and automatically adjusts the batch size.

2.1 Host Latency Hiding

When a TB issues a request of the host resources, the PullKernels TB scheduler retires this TB and runs another ready TB instead, based on a practical assumption that GPGPU apps typically launch tremendous number of TBs at once. After the host operation completes, PullKernels makes the requesting TB ready. The host latency hiding inherits an idea of the latency hiding mechanism implemented in GPU hardware; when a warp in GPU starts operations with high latency, the GPU warp scheduler retires this warp and dispatches other ready warps. Our mechanism is a software-level approach to run another ready TB during execution of high-latency host operations.

```
1  __device__ void doRead(DeviceLoop* loop,
2    int fd, size_t offset, size_t size){
3    if (offset < size){
4      size_t sizeToRead = min(PAGE_SIZE, size-offset);
5      auto callback =
6        [=](DeviceLoop* loop, uint8_t* buf, int read){
7        // ...
8        memory::release(loop, buf);
9        doRead(loop, fd, offset + PAGE_SIZE * gridDim.x, size);
10     };
11     fs::read(loop, fd, offset, sizeToRead, callback);
12     return;
13   }
14   fs::close(loop, fd, [=](DeviceLoop* loop, int err){ });
15 }
```

Figure 2: File Read Program on PullKernels.

Fig. 2 shows an example CUDA [17] program where TBs read a file. The function, doRead, reads a specified file up to the specified bytes, size. We define a callback function, callback, in the C++ lambda style at line 5. This callback processes read data and then calls doRead() again (lines 5–10). The program executes fs::read(..., callback), which requests a file read from the host and registers the passed callback. PullKernels executes the associated callback once the requested read has completed. Since doRead() is called in the callback, the running callback registers itself again via fs::read() (line 11). At the point of calling a non-blocking API, PullKernels stores the continuation of the TB as callback. PullKernels device side scheduler retires this TB until the host operation completes, running the other TBs by (re)-starting stored continuation.

2.2 Adaptive Host Operation Batching

GPU centric apps issue numerous requests of the host resource accesses, e.g., stream processing. Batching the data transfer of these requests is a key to amortize the cost of DMA and effectively utilize the bandwidth of the host-device connection. The configuration of the batch size is quite difficult because the best batch size depends on the target GPGPU app logic, the computing performance of the GPU, and the bandwidth of the host-device connection.

PullKernels employs adaptive host operation batching to dynamically find and use an appropriate batch size of the host operations. This mechanism is inspired from adaptive batching in IX [3] that targets TCP commutations, not communications between GPU and host resources. PullKernels-based apps issue host requests whose pace depends on the speed of the GPU-side code. To adaptively configure the size of batching and the height of the queue, PullKernels buffers requests until the current processing batch completes. These requests are queued in the host side and the host operation scheduler batches the buffered requests and data transfer.

3. CURRENT STATUS

We are prototyping PullKernels on the GLoop scheduling framework [26] with CUDA 9.0 on Pascal P100 GPU [19]. We extend GLoop device scheduler to implement host latency hiding mechanism. To effectively perform a DMA batch, we are tackling a design challenge of how we should design APIs to make the target data of the DMA continuous on memory. The design of the APIs is a key since DMA batching fails if the target data is scattered on memory.

4. ACKNOWLEDGMENTS

This work was supported in part by the Japan Society for the Promotion of Science (JSPS KAKENHI 15J09761) and Japan Science and Technology Agency (JST CREST JPMJCR1683).

5. REFERENCES

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *Proc. of the 12th USENIX Conf. on Operating Systems Design and Implementation*, pages 265–283. USENIX, 2016.

[2] S. R. Agrawal and A. R. Lebeck. Rhythm : Harnessing Data Parallel Hardware for Server Workloads. In *Proc. of the 19th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 19–34. ACM, 2014.

[3] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proc. of the 11th USENIX Conf. on Operating Systems Design and Implementation*. USENIX, 2014.

[4] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing. GeePS: scalable deep learning on distributed GPUs with a GPU-specialized parameter server. In *Proc. of the 11th European Conference on Computer Systems*, pages 1–16. ACM Press, 2016.

[5] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: A GPU-Accelerated Software Router. In *Proc. of the ACM SIGCOMM 2010 Conf.*, pages 195–206. ACM, 2010.

[6] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. Relational Joins on Graphics Processors. In *Proc. of the 2008 ACM SIGMOD Int'l Conf. on Management of Data*, pages 511–524. ACM, 2008.

[7] T. H. Hetherington, M. O'Connor, and T. M. Aamodt. MemcachedGPU: Scaling-up Scale-out Key-value Stores. In *Proc. of the 6th ACM Symp. on Cloud Computing*, pages 43–57. ACM, 2015.

[8] M. Hirabayashi, S. Kato, M. Edahiro, K. Takeda, T. Kawano, and S. Mita. GPU Implementations of Object Detection using HOG Features and Deformable Models. In *Proc. of the 1st Int'l Conf. on Cyber-Physical Systems, Networks, and Applications*, pages 106–111. IEEE, 2013.

[9] K. Jang, S. Han, S. Han, S. Moon, and K. Park. SSLShader: Cheap SSL Acceleration with Commodity Processors. In *Proc. of the 8th USENIX Conf. on Networked Systems Design and Implementation*, pages 1–14. USENIX, 2011.

[10] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. In *Proc. of the 22nd ACM International Conference on Multimedia*, pages 675–678, New York, New York, USA, 2014. ACM.

[11] T. Kaldewey, G. Lohman, R. Mueller, and P. Volk. GPU Join Processing Revisited. In *Proc. of the 8th Int'l Workshop on Data Management on New Hardware*, pages 55–62. ACM, 2012.

[12] S. Kato, J. Aumiller, and S. Brandt. Zero-copy I/O processing for low-latency GPU computing. In *Proc. of the 4th Int'l Conf. on Cyber-Physical Systems*, pages 170–178. ACM/IEEE, 2013.

[13] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs. In *Proc. of the 2010 Int'l Conf. on Management of Data*, pages 339–350. ACM, 2010.

[14] S. Kim, S. Huh, X. Zhang, Y. Hu, A. Wated, E. Witchel, and M. Silberstein. GPUnet: Networking Abstractions for GPU Programs. In *Proc. of the 11th USENIX Conf. on Operating Systems Design and Implementation*, pages 201–216. USENIX, 2014.

[15] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka. Physis: An Implicitly Parallel Programming Model for Stencil Computations on Large-Scale GPU-Accelerated Supercomputers. In *Proc. of the 2011 Int'l Conf. for High Performance Computing, Networking, Storage and Analysis*, pages 11:1 –11:12. ACM, 2011.

[16] M. McNaughton, C. Urmson, J. M. Dolan, and J.-W. Lee. Motion Planning for Autonomous Driving with a Conformal Spatiotemporal Lattice. In *Proc. of the 2011 Int'l Conf. on Robotics and Automation*, pages 4889–4895. IEEE, 2011.

[17] NVIDIA. CUDA C Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, 2015.

[18] NVIDIA. GPU-Based Deep Learning Inference: A Performance and Power Analysis. http://developer.download.nvidia.com/embedded/jetson/TX1/docs/jetson_tx1_whitepaper.pdf, 2015.

[19] NVIDIA. NVIDIA Tesla P100 – The Most Advanced Datacenter Accelerator Ever Built Featuring Pascal GP100, the World's Fastest GPU. http://www.nvidia.com/object/pascal-architecture-whitepaper.html, 2016.

[20] N. Rath, J. Bialek, P. J. Byrne, B. DeBono, J. P. Levesque, B. Li, M. E. Mauel, D. A. Maurer, G. A. Navratil, and D. Shiraki. High-speed, multi-input, multi-output control using GPU processing in the HBT-EP tokamak. *Fusion Engineering and Design*, pages 1895–1899, 2012.

[21] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey. Fast Sort on CPUs and GPUs: A Case for Bandwidth Oblivious SIMD Sort. In *Proc. of the 2010 Int'l Conf. on Management of Data*, pages 351–362. ACM, 2010.

[22] T. Shimokawabe, T. Aoki, T. Takaki, T. Endo, A. Yamanaka, N. Maruyama, A. Nukada, and S. Matsuoka. Peta-scale Phase-Field Simulation for Dendritic Solidification on the TSUBAME 2.0 Supercomputer. In *Proc. of the 2011 Int'l Conf. for High Performance Computing, Networking, Storage and Analysis*, pages 3:1–3:11. ACM, 2011.

[23] M. Silberstein. OmniX: an accelerator-centric OS for omni-programmable systems. In *Proc. of the 16th Workshop on Hot Topics in Operating Systems*, pages 69–75. ACM, 2017.

[24] M. Silberstein, B. Ford, I. Keidar, and E. Witchel. GPUfs: Integrating a File System with GPUs. In *Proc. of the 18th Int'l Conf. on Architectural Support*

for Programming Languages and Operating Systems, pages 485–498. ACM, 2013.

[25] W. Sun, R. Ricci, and M. L. Curry. GPUstore. In *Proc. of the 5th Annual Int'l Systems and Storage Conf.*, pages 1–12. ACM, 2012.

[26] Y. Suzuki, H. Yamada, S. Kato, and K. Kono. GLoop: An Event-driven Runtime for Consolidating GPGPU Applications. In *Proc. of the 2017 Symp. on Cloud Computing*, pages 80–93. ACM, 2017.

[27] L. Zeno, A. Mendelson, and M. Silberstein. GPUpIO: The Case for I/O-Driven Preemption on GPUs. In *Proc. of the 9th Annual Workshop on General Purpose Processing using Graphics Processing Unit*, pages 63–71. ACM, 2016.