Talk to My Neighbors Transport: Decentralized Data Transfer and Scheduling Among Accelerators

Amogh Akshintala*, Vance Miller†, Donald E. Porter*, and Christopher J. Rossbach ‡

*The University of North Carolina at Chapel Hill,
†The University of Texas at Austin,
‡ The University of Texas at Austin and VMware Research Group

1. Introduction

The demise of Dennard scaling has ushered in an era of unprecedented and ever-increasing heterogeneity, in pursuit of increasing performance via specialization. While CMOS scaling is believed to be approaching its end, continued increases in the number of transistors available on a chip have made specialized hardware an attractive alternative to increasing core counts or cache sizes. GPUs are commonplace in many computing domains [7, 14, 16, 17, 19, 24, 48], FPGAs are arriving in the cloud [15, 18, 34, 40]; smart storage [20, 43] and networking hardware [8, 32] are commercially available.

This proliferating menagerie of diverse computing resources is both a boon and burden for the application programmer. When specifying a computation, an application programmer must now specify not just the algorithm, but must account for device characteristics and orchestrate data movement to the device(s).

Work on this problem, to date, has focused on improving the *programmability* of these systems, in particular by eliminating the need for the application programmer to perform the complex interactions required to move data between disjoint physical memory spaces. Abstractions and techniques for data movement are well-explored in hardware [4–6], and software [12, 23, 28, 33, 41, 45, 46]. A common model is separating the *control plane*, responsible for specifying what data movement is allowed (e.g., by configuring DMA engines and configuring protections), from a *data plane*, which actually moves or manipulates the actual data as the computation proceeds. Traditionally, the control plane is the realm of the OS.

The fundamental issue is that many accelerators, such as GPUs, cannot run an OS, and control plane activities are delegated to run on a host CPU. The CPU then becomes a choke point on the critical path for data movement in the system. This paper provides data to show that this is the case.

Even if one pushes some control plane functionality onto the accelerators [28, 45, 46], current proposals are not sufficient to ensure efficient data movement. Consider a data transfer between GPU memory and CPU system memory. If the control plane is run on the CPU, a feature-rich, out-of-order core with a great deal of silicon devoted to ILP- and MLP-enhancing mechanisms is being squandered on simple data movement tasks, such as configuring DMAs. Alternatively, one could run the control plane on the GPU, but the massive internal parallelism is frittered away on copying data over the bus. In both cases, the hardware brought to bear on the problem would be better used on computation for which it is well-tuned, and simple data movement tasks preempt other useful work on that hardware.

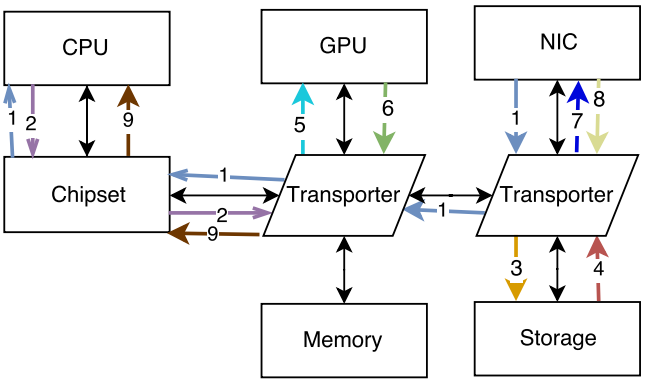
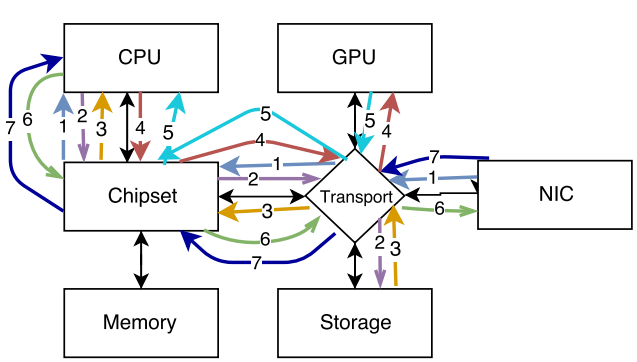


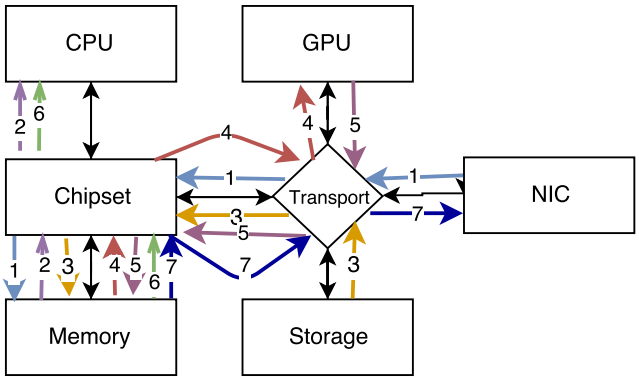
Figure 1: TMNT Control Flow.
Note: Transporters shown as 2 blocks as a visual convenience
1. NIC sends interrupt to CPU about incoming request 2. CPU figures out kernels that each accelerator needs to run, the data dependencies between them, etc., and then offloads the plan to TRANSPORTER 3. TRANSPORTER tells Storage to send required data to GPU. 4. Storage acknowledges completion 5. TRANSPORTER schedules processing of data on GPU 6. GPU acknowledges completion 7. TRANSPORTER triggers reply on NIC 8. NIC acknowledges completion 9. TRANSPORTER notifies completion to CPU

This paper argues for separating *transport* — the actual physical management of data, from the rest of the control plane by adding simple hardware specialized purely for this task, called TRANSPORTERS. TRANSPORTERS facilitate offloading accelerator scheduling, data movement, and inter-accelerator communication and co-ordination, through a management protocol called TALK TO MY NEIGHBORS TRANSPORT (*TMNT*).

TMNT abstracts away the management of *transport*, (e.g., memory/buffer management, data copying, and cache-coherence) allowing the application programmer and OS developer to express their computation in higher level abstractions, such as a Data Flow Graph (DFG). The TRANSPORTER hardware then assumes the scheduling of tasks on accelerators, pipelining of accelerators, and enforcing access control.



(a) Traditional Control Flow.
1. NIC sends interrupt to CPU about incoming request 2. CPU issues DMA request to read data from storage 3. Storage sends interrupt to CPU 4. CPU initiates GPU processing 5. GPU interrupts CPU about finished task 6. CPU initiates network transfer on NIC 7. NIC interrupts CPU



(b) Traditional Data Flow.
1. Move request packets from NIC to Memory 2. CPU loads request packets 3. CPU requests data from Storage 4. Move data to GPU for processing 5. Move results from GPU to memory 6. Read results to CPU 7. Send reply packets to NIC

Figure 2: Control and Data Flow on a Traditional System

TRANSPORTERS may be implemented on the bus hub (e.g., Gen-Z's hub) or as a cache-coherent side-core [29, 31] on the CPU die. Figure 1 shows the control flow for serving a request on an image-resize server that uses a GPU, a smart storage device, a smart NIC, and TRANSPORTERS.

In the remainder of this paper, we motivate the need for such a system by enumerating the challenges that designing a multi-accelerator system poses, with particular attention to performance and programming. We quantify the potential performance gains left on the table by current systems, and conclude with a design sketch of TRANSPORTERS and *TMNT*.

2. Motivation

We posit that in the near future, systems that leverage multiple accelerators will be fairly common. As an example, consider a system that is made up of several different accelerators: a Storage NXU, a Network NXU and a GPU — all of which are commercially available today [8, 20, 32, 43]. NXUs [46] are accelerators that operate near some I/O resource; *near* indicating that these compute units are closer than the CPU — they may be loosely-coupled with the I/O device, e.g., implemented as an FPGA that is on the same device, or in a more tightly integrated fashion, e.g., CPUs on the same die as the Flash/NVM like in Willow [43].

These devices operate as "Slaves" to the host CPU in the archetypal Master-Slave model of managing I/O devices, i.e., all data movement and control flow is controlled by the "Master". Data that is effectively exchanged between two accelerators must first be copied to the system's main memory before it can be copied into another accelerator. Control flow is, similarly, carried out in a synchronous manner by the software running on the CPU.

To illustrate the drawbacks of the Master-Slave model on multi-accelerator systems, we present two motivating examples that use GPU offload in the cloud, possibly orchestrated with other accelerators. First, consider a GPU-based image resizing workload at a web-service, such as Flickr [42, 46], which responds to an image request by reading a base image from storage and producing a resized version using a GPU. Second, consider image similarity search, in which requests perform image classification on the GPU, and the results are compared against a database in storage. These scenarios feature network and storage I/O as well as acceleration on specialized hardware.

In both of these examples, the sequence of events on a traditional system looks something like the data and control flow illustrated in Figures 2a and 2b, respectively. A request arrives over the network, packets are copied to the system's main memory, and an interrupt is delivered to the CPU (or the CPU polls the NIC). The CPU loads the request packets, performs some computation to determine the data it needs from storage, and issues the read request. The data is then DMA'd from storage into system memory. The CPU then launches a GPU kernel to perform computation on the data loaded from storage, and orchestrates movement of the data from system memory to the GPU's memory. Once the GPU finishes running the kernel, it sends an interrupts to the CPU, which then proceeds to copy the results from GPU memory to system memory. At this point the requested service is complete; the CPU moves the result to the NIC and the reply packets are sent out.

Consider the same workload running on a system with storage and network NXUs. Data that should effectively be flowing directly between the NXUs must first bounce through main memory, and control flow must be mediated on the CPU, leading to synchronous inter-device stalls to evaluate relatively simple and static policies.

When programming a system with multiple accelerators to perform a specific task, as in the examples outlined above, a

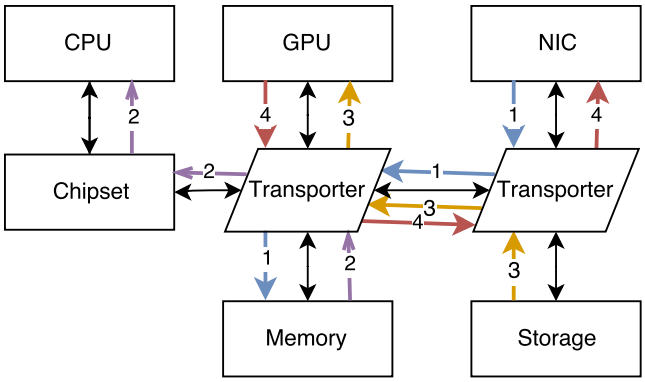


Figure 3: TMNT Data Flow.
Note: Transporters shown as 2 blocks as a visual convenience
1. Move request packets from NIC to Memory 2. CPU loads request packets
3. Move data from Storage to GPU for processing
4. Move results from GPU to NIC to be sent out

Figure 4: Overheads of UVM
Ordering of bars: *memcpy*, *UVM*, *UVM with prefetch*

dataflow programming model is a natural fit. Each accelerator can be modeled as a node in a Data Flow Graph (DFG) that consumes input from another node and/or produces data that is then input for another node.

Figure 3 shows the improved data flow for the examples described above on a multi-accelerator system. For example, steps 3 and 4 in Figure 2b can be replaced by a single step where data from the Storage NXU is directly moved to the GPU, as shown in Step 3 of Figure 3. Similarly, Steps 5, 6 and 7 in Figure 2b are replaced by Step 4 in Figure 3. In this case, the processed data from the GPU is moved directly to the Network NXU, which has instructions on replying to the user.

A key challenge to implementing a dataflow programming model in current systems is the lack of autonomy for the nodes. Rather, control flow in current systems is centralized and synchronous, which results in the well studied problem of the control plane being intertwined with the data plane. Prior work [13, 44, 49, 50, 52] has shown how to move data between accelerators with the least number of hops on current systems, even adaptively transferring to system memory as an optimization when the accelerator's memory is full, or the bus is busy, but decentralizing control flow remains an open problem.

3. Challenges

Any system designed to simplify accelerator programming must overcome several key challenges, 3 of which are described below. For each challenge, we observe how it can be addressed by separating out the transport layer.

Challenge #1: Abstracting low-level data placement and movement details without sacrificing performance.

Current models for programming accelerators (primarily GPGPUs) demand too much of the application programmer. In addition to tuning the computation to fit the model supported by the accelerator, the developers must also worry about the physical placement and movement of data. If the workload operates on a large dataset, the programmer must partition the data into chunks that will fit into the accelerator. In order to extract the maximal throughput from the system, the programmer must also manually schedule data movement such that it overlaps with computation on the GPGPU.

Current solutions to this challenge, such as NVIDIA's Unified Memory [35] and AMD's Heterogeneous System Architecture [33], address programmability of memory movement, but not performance. In Unified Memory, the GPU kernel operates in the same virtual address space as the CUDA application on the CPU. For data that should be shared between the GPU and CPU, the application uses a custom allocator, `cudaMallocManaged`. The CUDA framework assumes control of the physical location of these pages, migrating them to and from the GPU, driven by demand paging. Unified memory also frees the application programmer from partitioning their dataset to fit into the device's memory; GPU kernels can access larger data sets (up to the size of main memory) just as they would in a CPU program.

Unfortunately, Unified Memory typically results in worse performance compared to hand-written *memcpy*; Unified Memory also causes spikes in CPU utilization. GPUs are massively parallel processors that work in lock-step through a program [30]. For example, the Nvidia P100 device, that we use as an experimentation platform has 3584 CUDA cores [1]. Relying on demand-paging means that, at any given moment, some of these 3584 cores may be stalled while the offending page is being faulted in. When the the GPGPU kernel takes a page-fault, an interrupt is raised on the CPU to request the migration of the faulting frame. This interrupt is delivered to a handler registered by the CUDA framework, the *UVM-Handler*. The *UVM-Handler* services this request and initiates the transfer of the faulting frame to the GPU. We measured the average time to fault-in a page on the P100 GPU to be ~170 µs, whereas the average time to transfer a 4KiB page on the same x16 PCIe 3.0 bus is ~0.128 µs.

In order to understand the performance impact of using unified memory, we measured total execution time, and CPU utilization when running a micro-benchmark which al-

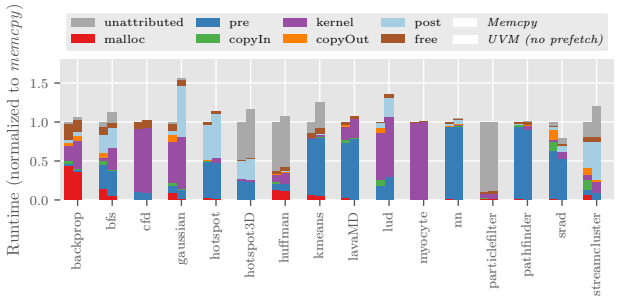


Figure 5: Execution time for rodinia benchmarks on the NVIDIA P100.

locates a configurable number of pages of memory, writes 1024 4-byte floating point numbers to each page in a *pre-processing* step, runs a *kernel* on the GPU that computes *round(pow(log(x), 2))* on each of the numbers — a simple, if somewhat nonsensical, computation that is easily parallelizable, and then verifies that each of the numbers is the correct expected value in a *post-processing* step on the CPU.

All experiments were carried out on a machine with 2 24-core Intel Xeon CPU E5-2650 v4 CPUs, 128 GiB of 2400MHz DDR4 DRAM, and 4 NVIDIA P100 GPUs connected via x16 PCIe 3.0 (although only one was used for our experiments). The P100 is a Pascal generation GPU with 16GiB of HBM2 memory. The system ran Ubuntu 16.04.3 LTS with a stock GNU/Linux 4.4.0-112-generic x86_64 kernel, and used the CUDA 9.1 framework. We ran the micro-benchmark in 3 different memory management configurations: *MemCpy*: where memory is explicitly moved to and from the accelerator using *cudaMemcpy()*, *UVM*: memory movement is managed by the Unified Memory handler, with no pre-fetch hints, and *UVM with pre-fetch*: memory is managed by the unified memory handler, but pre-fetch hints are provided where appropriate.

Figure 4 shows the end-to-end run time of the micro-benchmark, when run on differing sizes of input data (40 MB, 400 MB, and 4000 MB). The run time data presented is averaged across 100 iterations. For each data size, the run time is normalized to that of the *memcpy* configuration. We observe that unified memory performs worse than *memcpy* in all cases. While there is some speedup in the memory allocation phase from using the new *cudaMallocManaged* allocator, and from eliminating the copy-in and copy-out phases when using unified memory, these gains are significantly dwarfed by the extra time spent paging-in the data accessed during each processing stage. We observe that the *UVM-Handler* attempted to speculatively move data around (with and without pre-fetch hints), but it was not enough.

Figure 5 shows the run time for applications in the Rodinia [16] GPU benchmark suite when run using explicit memcpy and unified memory, normalized to memcpy. For a majority of the applications, unified memory negatively impacts their run time.

While schemes like unified memory are a step forward in simplifying the programming of accelerators, especially when the workload needs to access a larger data set than will fit in the accelerator's memory, this simplicity comes at a cost in performance. Given the monetary cost of the accelerators, and the computationally-intensive nature of the workloads that rely on such accelerators, application developers are reluctant to adopt Unified Memory [3]. Any system attempting to solve the programmability issue must also ensure that the performance impact of their scheme is minimal, or their solution will see poor adoption.

Observation: Manual memcpy is fast because programmers understand when data is going to be needed better than a framework like CUDA can infer. Frameworks that abstract away memory management should leverage this knowledge by providing programmers the means to express said relationships at a higher level of abstraction.

Challenge #2: Performance isolation of accelerators.

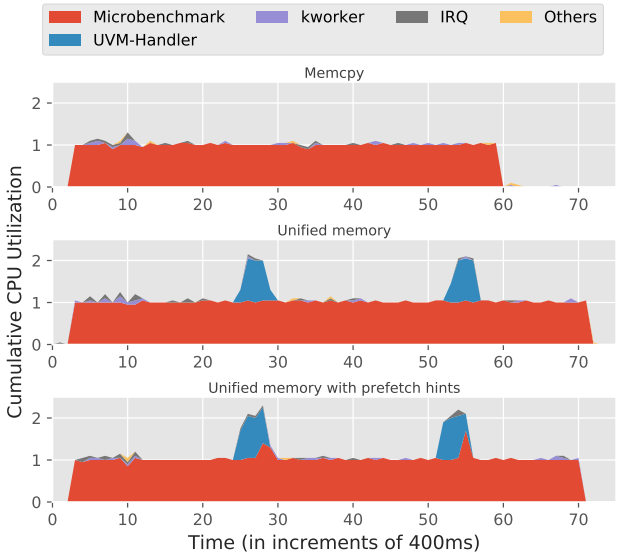


Figure 6: CPU Utilization when running a micro-benchmark that accesses 4000 MiB of data, touching each page 1024 times on the CPU in pre- and post- processing steps, and on the GPU.

Unified memory relies on CPU code to move data, which leads to spikes in CPU utilization. Figure 6 shows the CPU utilization while running one iteration of our micro-benchmark on 4000 MiB of data. We see that the *UVM-Handler* consumes 100% of cycles on 1 CPU core whenever data needs to be moved (~15% of the total run time of the micro-benchmark). Although we have little insight into what the *UVM-Handler* does during the time it occupies the CPU, our educated guess, both from reading the CUDA spec [36] and observing externally visible effects, is that there is some

bookkeeping about the physical location of frames, as well as, data movement and heuristic-based pre-fetch.

Worse, funneling data movement through the CPU can cause one accelerator to delay otherwise unrelated data movement for a different accelerator. Because the software on the CPU manages data movement for accelerators synchronously, the CPU becomes a bottleneck for data movement. This problem has been observed independently in systems that incorporate high-throughput I/O devices [37]. Attempting to build a dataflow-style pipeline composed of several high-throughput accelerators will likely exacerbate this situation even further. System software must ensure that there is performance isolation between accelerators.

Observation: Performance isolation becomes an issue when the control planes of otherwise independent devices are all bound together on a shared resource. Decentralization of control (autonomy) is key to performance isolation.

Challenge #3: Memory protection.

Hardware support for memory protection among accelerators remains primitive despite significant research effort [10, 11, 22, 25, 26, 38, 39, 51]. It remains to be seen if the situation will improve in the future.

IOMMUs have typically been used to enforce memory protection on traditional I/O devices, by mapping only those parts of memory that an I/O device should have access to for the given task. This model imposes high overheads for NXUs as it requires expensive round trips to the OS running on the CPU for access control.

Prior work [46] has made the case for taking advantage of the coherent-shared memory between the host CPU and their accelerators [33]. The idea is to decompose the application into services running on accelerators in a single address space. Applications can communicate by independently invoking services on other accelerators via shared memory message passing. This addresses the programmability challenge by abstracting away the details of access control, communication, and co-ordination (implemented by a combination of hardware and Library OSes [27]). The downside of this model is that it foregoes fault isolation.

In order to be able to pipeline accelerators in an efficient manner, the system needs to enable accelerators to enforce policies, but not make policy decisions.

Observation: Control is made up of two parts: Policy and Enforcement. Policy must remain centralized; Decentralizing enforcement grants enough autonomy to remove the CPU bottleneck, while preserving memory protection.

4. Design

In order to achieve programmability and performance in a multi-accelerator system, we envision a new system design wherein application programmers represent their program as a sequence of tasks that can be delegated to various accelerators, explicitly outlining the relationships between these tasks — this is known as a *plan*. New on-fabric hardware, called TRANSPORTERS, "execute" this *plan*, managing the various accelerators on the system, as necessary (e.g., data movement, communication, scheduling, and access control). In order to provide the same security guarantees as current systems, the *plan* is verified by the OS, and any necessary access privileges are pre-authorized.

Figure 1 shows the above changes in action for the image-resize example. In terms of the control/data plane framework, the data plane entirely resides on the TRANSPORTERS; the control plane is spread across both the CPU and the TRANSPORTERS. Policy decisions are made by the application program and the OS, but are enforced on the TRANSPORTER.

A natural place to implement the TRANSPORTER is on the switches of the bus that connects the CPU to the remaining devices in the system, giving the TRANSPORTER a great vantage point to observe and control the movement of data between devices. In the simplest mode of operation (when the attached devices do not support the *TMNT* protocol), TRANSPORTERS build on the idea of a Copy Engine [9, 21, 47, 53] to relieve the duties of managing data movement from the CPU. When *TMNT*-capable devices are present, TRANSPORTERS turn the otherwise passive bus into an active bus that can be programmed to provide services, such as pre-fetch, connection negotiation, buffer management, etc., to the attached devices.

We envision TALK TO MY NEIGHBORS TRANSPORT (*TMNT*) to be a management-layer protocol that enables the TRANSPORTER to orchestrate accelerators into a pipeline by managing task scheduling, access control, movement of data, and inter-accelerator co-ordination. *TMNT* enables the TRANSPORTER to effect finalized *plans*, offloaded to it by the OS, using the attached accelerators. The TRANSPORTER should be able to control devices using the legacy interfaces available today (command queues, ring buffers, etc.). Modifying accelerators to include support for the *TMNT* protocol may provide more performance opportunity.

Concretely, we envision *TMNT* as an extension to GenZ [5], a newly proposed inter-connect standard that introduces the notion of memory-centric communication, i.e., all devices perform loads and stores on memory locations that may be physically distributed on various devices, including main memory, GPU memory, and self-controlled NVM. This standard was primarily developed to allow memory resources to control themselves independent of processor-specific idiosyncrasies, thereby eliminating a common choke-point on current systems — the CPU chipset. GenZ inter-connects are packet-switched, passive, and backwards compatible with PCIe [2], the most commonly used inter-connect today. Layering *TMNT* on top of the GenZ protocol enables the system to take advantage of the data-path independence that GenZ provides.

References

[1] NVIDIA Tesla P100: The Most Advanced Data Center GPU Ever Built. http://www.nvidia.com/object/tesla-p100.html. Accessed: 2018-2-11.

[2] Specifications — PCI-SIG. https://pcisig.com/specifications/pciexpress/. Accessed: 2018-2-10.

[3] Tensorflow Github Issue #3678: New Feature: Pascal, Cuda 8, Unified memory. https://github.com/tensorflow/tensorflow/issues/3678. Accessed: 2018-1-31.

[4] Cache coherent interconnect for accelerators (ccix). https://www.ccixconsortium.com/, 2018.

[5] Gen-z. https://www.genzconsortium.org/, 2018.

[6] Opencapi. https://opencapi.org, 2018.

[7] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[8] Agilio. *SmartNIC Overview – Netronome*, 2018.

[9] Sandeep R Agrawal, Sam Idicula, Arun Raghavan, Evangelos Vlachos, Venkatraman Govindaraju, Venkatanathan Varadarajan, Cagri Balkesen, Georgios Giannikis, Charlie Roth, Nipun Agarwal, and Eric Sedlar. A many-core architecture for in-memory data processing. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 245–258. ACM, October 2017.

[10] R Ausavarungnirun, V Miller, J Landgraf, S Ghose, J Gandhi, A Jog, C J Rossbach, and O Mutlu. MASK: Redesigning the GPU Memory Hierarchy to Support Multi-Application Concurrency. 2018.

[11] Rachata Ausavarungnirun, Joshua Landgraf, Vance Miller, Saugata Ghose, Jayneel Gandhi, Christopher J Rossbach, and Onur Mutlu. Mosaic: a GPU memory manager with application-transparent support for multiple page sizes. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 136–150. ACM, October 2017.

[12] Antonio Barbalace, Anthony Iliopoulos, Holm Rauchfuss, and Goetz Brasche. It's time to think about an operating system for near data processing architectures. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, pages 56–61, New York, NY, USA, 2017. ACM.

[13] Stephen Bates. Project Donard: Peer-to-Peer Communication with NVM Express Devices, Part 1 - PMC Blog. http://blog.pmcs.com/project-donard-peer-to-peer-communication-with-nvm-express-devices-part-1/, September 2014. Accessed: 2016-5-12.

[14] M. Burtscher, R. Nasre, and K. Pingali. A Quantitative Study of Irregular Programs on GPUs. In *IISWC*, 2012.

[15] Adrian Caulfield, Eric Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. A cloud-scale acceleration architecture. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, October 2016.

[16] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54. Ieee, 2009.

[17] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 44–54, Washington, DC, USA, 2009. IEEE Computer Society.

[18] Amazon EC2. Amazon ec2 f1 instances, 2017.

[19] M. Flynn. Very High-Speed Computing Systems. *Proc. of the IEEE*, 54(2), 1966.

[20] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, Jaeheon Jeong, and Duckhyun Chang. Biscuit: A framework for near-data processing of big data workloads. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pages 153–165, Piscataway, NJ, USA, 2016. IEEE Press.

[21] T J Ham, J L Aragón, and M Martonosi. DeSC: Decoupled supply-compute communication management for heterogeneous architectures. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 191–203, December 2015.

[22] Y Hao, Z Fang, G Reinman, and J Cong. Supporting Address Translation for Accelerator-Centric Architectures. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 37–48, February 2017.

[23] Mark Harris and View All Posts by. Unified Memory for CUDA Beginners — NVIDIA Developer Blog. https://devblogs.nvidia.com/unified-memory-cuda-beginners/, June 2017. Accessed: 2018-1-19.

[24] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. Mars: A MapReduce Framework on Graphics Processors. In *PACT*, 2008.

[25] Adwait Jog, Onur Kayiran, Tuba Kesten, Ashutosh Pattnaik, Evgeny Bolotin, Niladrish Chatterjee, Stephen W Keckler, Mahmut T Kandemir, and Chita R Das. Anatomy of GPU Memory System for Multi-Application Execution. In *Proceedings of the 2015 International Symposium on Memory Systems*, MEMSYS '15, pages 223–234, New York, NY, USA, 2015. ACM.

[26] Adwait Jog, Onur Kayiran, Asit K Mishra, Mahmut T Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R Das. Orchestrated Scheduling and Prefetching for GPGPUs. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 332–343, New York, NY, USA, 2013. ACM.

[27] M Frans Kaashoek, Dawson R Engler, Gregory R Ganger, Hector M Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Macken-

zie. Application Performance and Flexibility on Exokernel Systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 52–65, New York, NY, USA, 1997. ACM.

[28] Sangman Kim, Seonggu Huh, Yige Hu, Xinya Zhang, Emmett Witchel, Amir Wated, and Mark Silberstein. Gpunet: Networking abstractions for gpu programs. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 201–216, Berkeley, CA, USA, 2014. USENIX Association.

[29] Sanjay Kumar, Himanshu Raj, Karsten Schwan, and Ivan Ganev. Re-architecting vmms for multicore systems: The sidecore approach. In *Workshop on Interaction between Operating Systems & Computer Architecture (WIOSCA)*, 2007.

[30] E Lindholm, J Nickolls, S Oberman, and J Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 28(2):39–55, March 2008.

[31] Jiuxing Liu and Bulent Abali. Virtualization polling engine (vpe): Using dedicated cpu cores to accelerate i/o virtualization. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 225–234, New York, NY, USA, 2009. ACM.

[32] Mellanox. *Mellanox Innova 2*, 2018.

[33] mfrickie. HSA Foundation Members Preview Plans for Heterogeneous Platforms - HSA Foundation. `http://www.hsafoundation.com/hsa-foundation-members-preview-plans-heterogeneous-platforms/`, October 2015. Accessed: 2015-12-18.

[34] Microsoft. Microsoft azure goes back to rack servers with project olympus, 2017.

[35] NVidia. *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, 2007.

[36] NVIDIA. *CUDA C PROGRAMMING GUIDE version 9.1*, January 2018.

[37] Simon Peter, Jialin Li, Irene Zhang, Dan R K Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System Is the Control Plane. *ACM Transactions on Computer Systems*, 33(4):11:1–11:30, November 2015.

[38] Bharath Pichai, Lisa Hsu, and Abhishek Bhattacharjee. Architectural Support for Address Translation on GPUs: Designing Memory Management Units for CPU/GPUs with Unified Address Spaces. *SIGARCH Comput. Archit. News*, 42(1):743–758, February 2014.

[39] J Power, M D Hill, and D A Wood. Supporting x86-64 address translation for 100s of GPU lanes. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 568–578, February 2014.

[40] Andrew Putnam, Adrian Caulfield, Eric Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, Eric Peterson, Aaron Smith, Jason Thong, Phillip Yi Xiao, Doug Burger, Jim Larus, Gopi Prashanth Gopal, and Simon Pope. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA)*, pages 13–24. IEEE Press, June 2014.

[41] Christopher J Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. PTask: operating system abstractions to manage GPUs as compute devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 233–248. ACM, 2011.

[42] Archie Russell. Real-time Resizing of Flickr Images Using GPUs — code.flickr.com. `http://code.flickr.net/2015/06/25/real-time-resizing-of-flickr-images-using-gpus/`. Accessed: 2018-2-2.

[43] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. Willow: A user-programmable ssd. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*, pages 67–80, 2014.

[44] Mustafa Shihab, Karl Taht, and Myoungsoo Jung. GPUdrive: Reconsidering Storage Accesses for GPU Acceleration. In *Workshop on Architectures and Systems for Big Data*, 2014.

[45] M. Silberstein, B. Ford, I. Keidar, and E. Witchel. Gpufs: Integrating a file system with gpus. 2013.

[46] Mark Silberstein. Omnix: An accelerator-centric os for omni-programmable systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, pages 69–75, New York, NY, USA, 2017. ACM.

[47] Alan Jay Smith. The task of the referee. *Computer*, 23(4):65–71, 1990.

[48] J. A. Stratton, C. Rodrigues, I. J. Sung, N. Obeid, L. W. Chang, N. Anssari, G. D. Liu, and W. W. Hwu. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. Technical Report IMPACT-12-01, Univ. of Illinois at Urbana-Champaign, March 2012.

[49] H W Tseng, Q Zhao, Y Zhou, M Gahagan, and S Swanson. Morpheus: Creating Application Objects Efficiently for Heterogeneous Computing. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 53–65, June 2016.

[50] Hung-Wei Tseng, Yang Liu, Mark Gahagan, Jing Li, Yanqin Jin, and Steven Swanson. Gullfoss: Accelerating and Simplifying Data Movement among Heterogeneous Computing and Storage Resources. Technical report, Tech. Rep. CS2015-1015, Department of Computer Science and Engineering, University of California, San Diego technical report, 2015.

[51] J Vesely, A Basu, M Oskin, G H Loh, and A Bhattacharjee. Observations and opportunities in architecting shared virtual memory for heterogeneous systems. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 161–171, April 2016.

[52] J Zhang, D Donofrio, J Shalf, M T Kandemir, and M Jung. NVMMU: A Non-volatile Memory Management Unit for Heterogeneous GPU-SSD Architectures. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 13–24, October 2015.

[53] L Zhao, L N Bhuyan, R Iyer, S Makineni, and D Newell. Hardware Support for Accelerating Data Movement in Server Plat-

form. *IEEE transactions on computers. Institute of Electrical and Electronics Engineers*, 56(6):740–753, June 2007.