Efficient Multi-GPU Shared Memory via Automatic Optimization of Fine-Grained Transfers

Harini Muthukrishnan
University of Michigan
harinim@umich.edu

David Nellans
NVIDIA
dnellans@nvidia.com

Daniel Lustig
NVIDIA
dlustig@nvidia.com

Jeffrey A. Fessler
University of Michigan
fessler@umich.edu

Thomas F. Wenisch
University of Michigan
twenisch@umich.edu

***Abstract*—Despite continuing research into inter-GPU communication mechanisms, extracting performance from multi-GPU systems remains a significant challenge. Inter-GPU communication via bulk DMA-based transfers exposes data transfer latency on the GPU's critical execution path because these large transfers are logically interleaved between compute kernels. Conversely, fine-grained peer-to-peer memory accesses during kernel execution lead to memory stalls that can exceed the GPUs' ability to cover these operations via multi-threading. Worse yet, these sub-cacheline transfers are highly inefficient on current inter-GPU interconnects. To remedy these issues, we propose PROACT, a system enabling remote memory transfers with the programmability and pipeline advantages of peer-to-peer stores, while achieving interconnect efficiency that rivals bulk DMA transfers. Combining compile-time instrumentation with fine-grain tracking of data block readiness within each GPU, PROACT enables interconnect-friendly data transfers while hiding the transfer latency via pipelining during kernel execution. This work describes both hardware and software implementations of PROACT and demonstrates the effectiveness of a PROACT software prototype on three generations of GPU hardware and interconnects. Achieving near-ideal interconnect efficiency, PROACT realizes a mean speedup of 3.0× over single-GPU performance for 4-GPU systems, capturing 83% of available performance opportunity. On a 16-GPU NVIDIA DGX-2 system, we demonstrate an 11.0× average strong-scaling speedup over single-GPU performance, 5.3× better than a bulk DMA-based approach.**

***Index Terms*—GPGPU, multi-GPU, strong scaling, GPU memory management, data movement, heterogeneous systems**

I. Introduction

Despite advancements in GPU architecture and programming models, achieving peak performance on multi-GPU systems remains a challenge for GPU programmers [1]–[6]. To efficiently parallelize applications across a multi-GPU system, developers typically distribute an application's data structures across each GPU's physical memory. Programs are also designed to operate in logical phases, alternating between periods of heavy computation accessing mostly locally-available data and periods of data distribution and synchronization among the GPUs. Unfortunately, this typically results in inefficient use of resources: interconnects sit idle during computation phases, and compute units sit idle during communication phases.

To obtain high interconnect utilization and maximize performance on multi-GPU systems, developers are forced to dedicate substantial manual effort to performance-tuning and re-architecting their applications to ensure optimal data distribution, inter-GPU communication, and synchronization management. These steps often require detailed knowledge of internal GPU architecture, GPU interconnects, and the memory access patterns of applications. As a result, peak multi-GPU performance is often only achievable by "ninja" programmers.

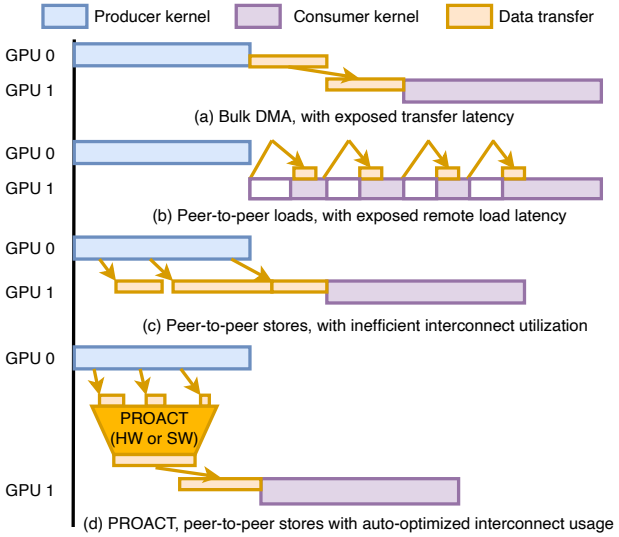


Fig. 1: **Different multi-GPU communication paradigms**

One major impediment to improving multi-GPU performance is the difficulty in efficiently overlapping GPU computation with communication phases, leading to system resource under-utilization. Historically, multi-GPU applications have used bulk DMA-based communication among GPUs because these DMA transfers provide high interconnect utilization during the transfer phases. Unfortunately, due to bulk synchronization, GPU computation cannot generally take place during these communication phases. Figure 1(a) depicts a typical DMA-based transfer between two GPUs. The DMA transfer occurs between the producer and consumer kernels on the GPUs, exposing the data transfer latency.

More recently, fine-grained peer-to-peer (P2P) transfers enable GPU compute units to issue loads and stores directly to the physical memory of remote GPUs. These peer-to-peer (P2P) transfers enable inherent overlap of compute and communication, but P2P transfers are inefficient on current GPU interconnects, leading to gross under-utilization of the

978-1-6654-3333-4/21/$31.00 ©2021 IEEE
DOI 10.1109/ISCA52012.2021.00020

inter-GPU interconnect. As depicted in Figure 1(b), P2P loads often result in the consumer kernel stalling while waiting for remote loads to complete, due to interconnect latency, negatively impacting performance. As shown in Figure 1(c), P2P stores between GPUs are often generated at sub-cacheline granularity with sporadic access patterns. Although some store latency is hidden, due to protocol packetization overheads, these stores do not use the GPU-interconnect efficiently.

This work proposes PROACT, a joint compile-time and runtime system that combines the flexibility of peer-to-peer stores with the interconnect efficiency of bulk transfers, as shown in Figure 1(d). PROACT exposes an easy-to-use P2P-store programming model to developers but leverages profiling and GPU-runtime support to track data generation, perform transfer coalescing, and dynamically issue transfers over the interconnect to achieve high utilization. PROACT provides efficient overlap of per-GPU computation and inter-GPU data transfer by performing the transfers proactively soon after data generation and can be supported with either dedicated hardware or software (at the expense of some GPU computation resources).

To evaluate the PROACT approach of balancing fine-grained transfers with intelligent interconnect utilization, we provide a comprehensive performance comparison to other common multi-GPU programming paradigms that use automatic GPU memory management, barrier-based GPU transfers, and fine-grained data accesses. To demonstrate the performance of PROACT, we implement a software prototype of our design on real hardware. We show that even while consuming GPU resources (an overhead which would be eliminated with future hardware support), PROACT is the best performing approach to multi-GPU programming, enabling strong scaling to an 11.0× mean performance improvement on a prototype 16-GPU system—5.3× better than the next-best alternative, a standard bulk-synchronous approach.

II. Background

A. CUDA Programming Model

CUDA [7] extends C++ by allowing the programmer to define functions, called *kernels*, that when called, are executed in parallel by many thousands of threads grouped into *blocks*. Sets of threads (32 in recent GPUs) within a block are grouped into *warps*, scheduled by hardware and executed in lock-step on a Streaming Multiprocessor (SM). GPU programming involves launching kernels with pre-configured thread counts and block dimensions on the GPU(s).

To parallelize an application across GPUs, kernels are launched on each GPU separately, with the execution spread across multiple GPUs, and the data structures localized on one GPU or replicated or partitioned across multiple GPUs. The GPU count can scale along with both the compute and memory bandwidth available to the application. Though application developers would prefer the work performed by each GPU to be fully independent, in practice, many/most algorithms inherently require some amount of communication among threads. In such applications, the relative portion of time spent on inter-GPU communication generally increases along with GPU count, and communication eventually becomes the performance bottleneck for strong scaling.

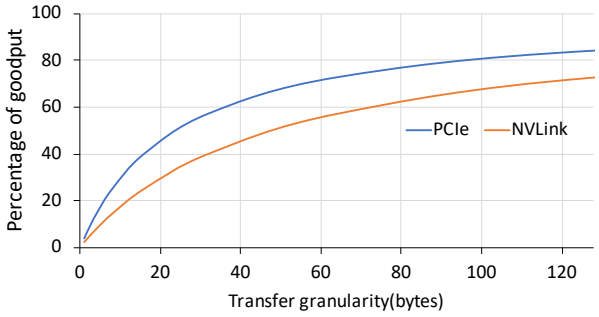


Fig. 2: **Interconnect goodput vs. write transfer granularity.**

B. Inter-GPU Communication Mechanisms

Regardless of the amount of remote data a multi-GPU program may access during its execution, GPU programmers must choose among several mechanisms to perform these accesses, and each mechanism has unique performance characteristics described below. We describe these mechanisms and their performance implications using NVIDIA's terminology to provide specific examples, though the concepts themselves are general. Pedagogically, we refer to a kernel that requires data produced by another GPU as a consumer kernel and the kernel that generates data for kernels on other GPU(s) as a producer kernel. In practice, a single kernel is often both the producer of some data and a consumer of other data simultaneously.

DMA-based bulk transfers: When using bulk DMA-based transfers between GPUs, data transfer is typically explicitly invoked from the host program executing on the CPU via `cudaMemcpy()`. The transfer is scheduled following the completion of the producer kernel, and before the invocation of the consumer kernel. The `cudaMemcpy()` call invokes the GPU's hardware DMA engine to transfer data directly between the GPUs' memories without the need for the transfer to reflect off of the CPU's memory system. This paradigm can saturate most GPU interconnects when transferring very large granularity data and ensures that subsequent accesses made by the consumer will be serviced at high bandwidth from within the consumer GPU's local physical memory. However, this method is not suitable for small (several cache lines) or medium (several KB) sized transfers due to the high initialization and synchronization overhead of returning to the host program and then programming the DMA engine. Each of these steps can consume several microseconds [8], which dominates the data transfer time itself. Though programmers can attempt to interleave computation and communication using DMA-based transfers, it requires substantial programmer expertise and effort.

Peer-to-peer (P2P) GPU accesses: In modern multi-GPU systems, individual GPUs are capable of directly reading and writing the physical memory of peer GPUs without GPU

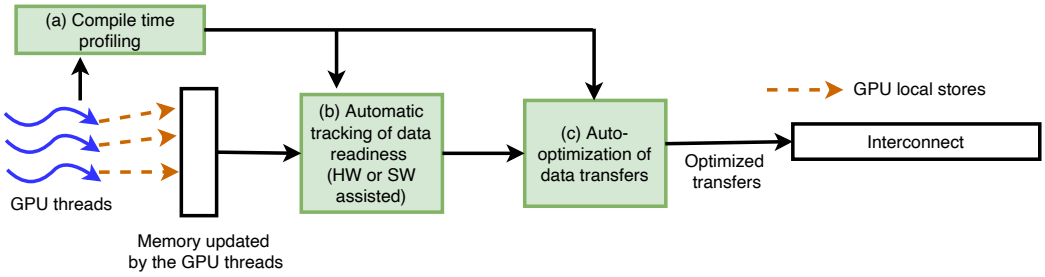


Fig. 3: **Overview of the major logical PROACT components.**

runtime intervention or host CPU synchronization. The advantages of performing direct remote accesses using GPU threads is that there is no initiation overhead (such as programming a DMA engine) and the accesses can be performed while unrelated computation continues in parallel, within a given thread and without stalling other threads. However, when loads are performed from remote memory, these P2P loads often stall thread execution beyond the GPUs' ability to hide the load latency with multi-threading due to the longer latency of the inter-GPU interconnect as compared to local memory. As these load stalls build up within the GPU's memory system, they eventually consume precious GPU resources that would otherwise have been used to make the execution progress.

Unlike P2P reads that eventually may stall each issuing thread in the GPU, P2P writes almost never stall the issuing thread because there is no implicit dependence within program execution. As such, they consume substantially fewer on-chip resources. We believe the non-blocking nature of P2P stores makes them fundamentally the most efficient way to perform multi-GPU communication. P2P stores are not a panacea, however. Straightforward use of P2P stores will typically result in numerous small writes (as small as one byte) issued over the inter-GPU interconnect, which results in poor interconnect utilization. PROACT adopts P2P stores as the basis of its programming model, and the major contribution of this work is designing a system that can overcome poor interconnect efficiency when performing small P2P writes.

Programmatic communication libraries: To ease the development burden, it is common for GPU programmers to use GPU-aware libraries such as MPI [9], NVIDIA's NCCL [10], or NVIDIA's NVSHMEM [11] to provide universal inter-GPU communication and synchronization patterns, irrespective of the interconnect technology and topologies on which the code runs. In turn, these libraries may selectively use bulk DMA transfers or P2P accesses under the hood, combined with the library-controlled memory management to optimize communication between GPUs.

Although these libraries continue to be optimized, there is little they can do to avoid DMA initiation overhead when performing large transfers. None of these libraries attempt to aggregate fine-grained transfers intelligently to improve interconnect efficiency to the best of our knowledge. We will show that brief compile-time profiling of a system and runtime optimization for interconnect utilization can provide a substantial performance improvement over the underlying mechanisms used by these libraries. Without loss of generality, the PROACT technique could be implemented as a new back end to many of these commonly used libraries. Section VI discusses several additional GPU communication libraries.

C. Inter-GPU Interconnect Efficiency

System architects and builders have a range of interconnect technology choices while designing multi-GPU systems. For over a decade and still today, PCIe [12] has been the dominant interconnect used to attach peripherals and accelerators to CPUs, and to each other. More recently, NVLink [13] was designed by NVIDIA as a dedicated GPU interconnect that targeted higher bandwidth and improved scalability over PCIe. Other high-performance interconnects such as Infiniband [9] and AMD's Infinity Fabric [14] exist and target large-scale networking, CPU-CPU, and CPU-GPU connections.

Despite both evolutionary (PCIe) and clean slate (NVLink) design progression where both protocols support direct peer-to-peer accesses between GPUs, both PCIe and NVLink have poor efficiency when performing small accesses. Figure 2 shows the percentage of goodput achieved on the interconnect, i.e., the percentage of useful data delivered over PCIe and NVLink interconnects for varying store granularities. Both interconnect technologies provide high efficiency for transfers with greater than 128 bytes (a common cacheline size) but drop off dramatically at smaller transfer sizes. Interconnect efficiency decreases at these smaller granularities because protocol packetization overheads dominate the effective goodput, and the transfer efficiency falls as low as 8% on NVLink and 14% on PCIe for 4-byte stores. Therefore, while P2P writes may fundamentally be the most efficient way to transfer data from a latency-hiding perspective, dramatic improvements in interconnect efficiency must be achieved to improve overall multi-GPU performance.

III. PROACT Design and Implementation

PROACT attempts to bridge the benefits of DMA-based bulk copies and peer-to-peer accesses in multi-GPU systems to provide the interconnect efficiency of large transfers with the non-blocking semantics of SM initiated peer-to-peer stores. PROACT improves the performance of multi-GPU systems by (1) balancing the overlap of data transfers with GPU

computation, (2) maximizing the opportunity for write coalescing, (3) smoothing interconnect utilization over time to ensure no bandwidth is wasted, and (4) increasing interconnect efficiency by ensuring communication occurs at sufficiently large granularities.

To provide these benefits for a range of applications with varying data access patterns, PROACT needs to make the following design choices depending on the application and system architecture.

- **Transfer mechanism**: GPUs provide different ways to transfer data among memories. The best approach overlaps compute with communication while providing high interconnect utilization and efficiency.
- **Transfer granularity**: Data transfer mechanisms vary in efficiency as a function of granularity. Performing transfers in coarser chunks can reduce initiation overhead but may result in a large *tail chunk* that extends past the end of the computation, delaying the next phase. On the other hand, transferring data as fine-grained chunks can lead to poor interconnect efficiency, as described in Section II-C.
- **Transfer resources**: Performing data transfers by employing all threads in the system typically leads to interconnect inefficiencies and compute stalls, but developing applications that use per-thread warp specialization so that only a subset of the threads are writing data to remote GPUs is both difficult to implement and error-prone. Thus, PROACT needs to automatically identify the appropriate amount of GPU resources to perform the transfers and hide this complexity from programmers as much as possible.
- **Tracking data generation**: When data transfer is performed by a different thread or engine than the thread that produced the data, PROACT must develop an appropriate mechanism to track data production and synchronize the producer with the transfer mechanism.

PROACT has three primary components that coordinate up and down the software and hardware stack to make these design choices and maximize throughput. Figure 3 provides an overview of these building blocks. They are (a) a compile-time profiler to determine the different PROACT parameters required to achieve optimized transfer efficiency, (b) a tracking unit to monitor data readiness, and (c) a data transfer mechanism that initiates optimized transfers. Each component is described in further detail below.

A. Optimizing Transfer Efficiency via Profiling

The first step in improving interconnect efficiency is identifying how to extract performance from the multi-GPU interconnect without hampering GPU compute throughput. If the granularity of P2P stores is too small, they offer poor interconnect efficiency, as described earlier in Section II-C. If a sufficient number of writes are not in flight on the interconnect at any given time, bandwidth will go underutilized.

Perhaps unsurprisingly, performance is a complicated function of multiple parameters. Figure 4 provides an example

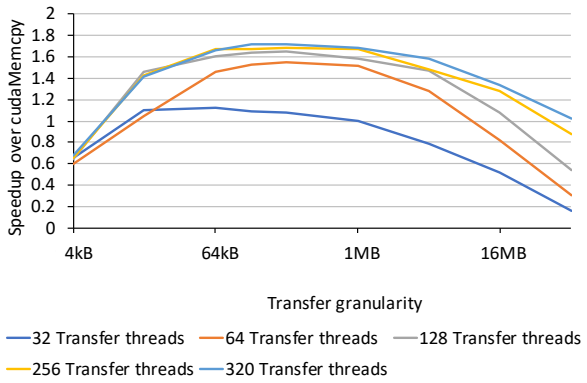


Fig. 4: **Automatic profiling showing workload throughput vs GPU transfer threads and aggregate transfer size.**

of the sensitivity to the number of transfer threads (see Section III-C) and aggregate transfer granularity on an NVIDIA Kepler-based system based on a microbenchmark study described in Section IV-C. The performance trends for this study are explored in greater detail in Section V-A. In this case, the best application throughput is obtained for transfer granularities between 64kB and 1MB when the number of threads used for the transfers is more than 128. For this study, we observed that 128 threads are sufficient to saturate the interconnect bandwidth, and including more threads resulted in no improvement in interconnect utilization. Each GPU generation and interconnect type has a specific number of GPU threads needed to saturate the interconnect bandwidth. By examining these profiles, PROACT can choose the number of transfer threads for a given system to maximize interconnect bandwidth while minimizing how many GPU execution lanes are used for data transfer.

To facilitate exploration of the complex design space, as shown in Figure 3(a), PROACT contains a compile-time software profiling suite that identifies the appropriate balance of the many competing factors that affect performance on a given GPU and interconnect. The PROACT profiler performs a parameter sweep and analyzes application performance across PROACT's different transfer mechanisms (described later in Section III-C), and varying the transfer chunk size and transfer thread count, to identify the configuration that achieves the best application runtime for a given application and system. It then configures the application compilation to emit code that is most efficient according to the profiling results.

We found that a brute-force search across the configuration space (explained in the following sections) was feasible and sufficient for the PROACT profiling pass. The best PROACT configuration depends on the GPU architecture, interconnect architecture, and compute and interconnect bandwidth requirements of the application, thus requiring the profiling on a per-application basis.

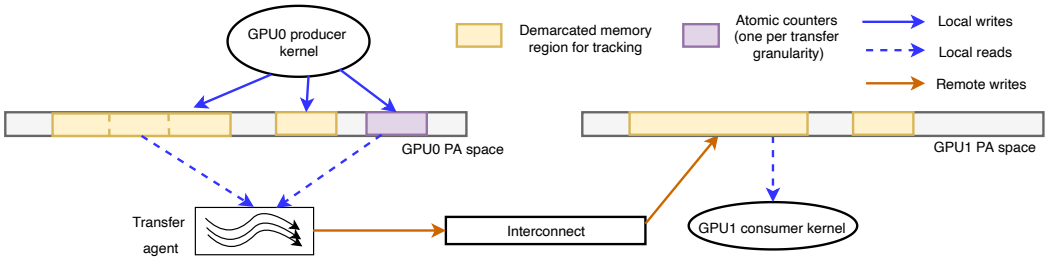


Fig. 5: **Decoupled local data generation with region tracking for decoupled transfers.**

B. Tracking Local Data Transfer Readiness

PROACT presents users with a programming model that encourages the use of P2P stores; however, transmitting data over the interconnect at fine granularity leads to poor interconnect efficiency. We propose that remote GPU transfers be decoupled from each thread's writes by aggregating these writes into local GPU memory before being transferred to remote GPUs. As shown in Figure 5, PROACT provides a staging mechanism whereby GPU threads individually write to local memory indicated by the programmer as a PROACT-enabled region. This locally written data is then propagated to remote GPUs by an asynchronously operating transfer agent. By decoupling the transfer of data over the interconnect from the generation of data locally, PROACT can optimize the transfer timing, granularity, and mechanisms it chooses to employ (based on profiling). Note that PROACT does not allocate and deploy explicit data transfer buffers, thus avoiding extra copy operations. Once a memory region is PROACT-enabled, a 1:1 correspondence is maintained between local and remote memory locations for that region. In other words, all the local writes to a PROACT-enabled region are sent to the remote GPUs.

The NVIDIA GPU memory model only requires weak writes and strong writes with `cta` and `gpu` scopes to be visible at remote GPUs no later than the succeeding global synchronization barrier. PROACT exploits this slack by aggregating data in local memory until all writes to a transfer chunk complete, only then pushing out data to keep the interconnect saturated with high-efficiency transfers. Strong writes with `sys`-scope are typically used as global memory synchronization instructions rather than for performing data transfers and hence do not fall under the purview of PROACT.

Atomic counters to track data generation: The existence of a decoupled transfer mechanism implies a need to track when a transfer is ready to begin. PROACT employs a set of atomic counters (one per chunk within the determined transfer granularity) to track data generation and begin the transfers. While ultra-fast hardware support for tracking data readiness may be beneficial, it is unnecessary because GPUs today already support atomic counters in memory (typically implemented in the GPU's L2 for performance reasons). Each atomic counter is initialized with a value equal to the number of Compute Thread Arrays (CTAs) that issue writes to its corresponding chunk, as determined by the compiler. During execution, the counter decrements with every CTA that writes to a chunk and triggers the transfer agent to begin copying data when the counter reaches zero. PROACT requires applications to issue a deterministic number of stores for it to identify the completion of data generation on a per chunk basis and perform the transfers.

Figure 5 shows a simplified example in which an application's memory is laid out linearly. As the producer kernel produces data, it also updates the atomic counters, maintained at a different location in memory. The transfer agent polls these counters, and when the expected value is observed, the transfer starts. Linear arrangements of data in memory are not required but writes to contiguous memory result in better performance and reduce the amount of storage overhead needed for the atomic counters. We note that because PROACT typically chooses transfer granularities of 4kB–16MB (shown later in Table II), the storage overhead of the counters themselves is not a significant concern.

C. Choosing the Decoupled Data Transfer Mechanism

By decoupling data transfer from local data generation, we have the flexibility to choose an inter-GPU transfer mechanism and transfer granularity that can maximize bandwidth utilization between GPUs. The initiation overhead of the GPU DMA engine is not appropriate for frequent data movement among GPUs, leaving us with three alternatives.

Polling: In this technique, a small configurable number of GPU warps are specialized to perform data transfers at a given granularity. Auto-generated and managed entirely by the PROACT runtime, this independently launched long-lived kernel polls the structured data-ready atomic counters that are updated by the application's producer kernels. When ready, the transfer threads read a chunk from the local staging array and perform remote writes to the target GPU(s). While PROACT's profiler tries to minimize the number of transfer threads required to maximize interconnect performance, ultimately, these threads and the polling of the counters (in a software implementation) compete with the compute kernel(s) for execution throughput. Memory consistency between the producer threads and the decoupled transfer threads is enforced using existing GPU memory ordering fences in our PROACT software prototype.

Dynamic Kernel Launches: CUDA Dynamic Parallelism (CDP) [15] presents another transfer agent mechanism by allowing a parent kernel to launch child kernels as needed. In contrast to using polling which consumes compute bandwidth in the polling loops that monitor for chunk transfer readiness, dynamic kernels are launched only when a chunk is ready for transfer and consume compute resources only during the transfer itself. The CUDA runtime guarantees that parent and child kernels have a fully consistent view of global memory when the child kernel starts and ends, so no additional synchronization is needed. PROACT implements CDP-based transfers as an alternative to polling to avoid the compute bandwidth cost of polling the transfer bitmaps. The trade-off, however, is that CDP kernel launch has a higher initiation latency than polling, though still lower than DMA-based transfers.

To use CDP-based transfers when the atomic data counter indicates that a data chunk is ready for transfer, PROACT's instrumentation in the producer kernel launches a dynamic kernel (with a pre-configured number of threads) to perform the transfer. This dynamic kernel transfers the data chunks to all destination GPUs using tightly packed SM store instructions.

Direct Inline Stores: To evaluate the entire design space of options, our PROACT software prototype also includes an *inline* version wherein native P2P stores issue remote writes to distribute data to remote GPUs as data are produced (i.e., without deferring to decoupled transfer threads), though we do not expect this to perform well in practice. Direct inline stores have the advantage that they spread remote writes over the course of the producer kernel without the tracking overhead of decoupled transfers, smoothing interconnect utilization as long as remote writes are evenly distributed within the kernel execution. Because GPU stores are usually non-blocking and can coalesce adjacent writes, remote transfer latency is hidden unless queuing resources are exhausted. Functionally, a memory ordering barrier present implicitly at the end of the producer kernel ensures that all transfers are complete and subsequent accesses by the consumer kernel can then be made locally. If data generated by the producer has poor spatial locality, write coalescing may fail, inflating the interconnect bandwidth required to transfer the application's data relative to the decoupled transfer mechanisms.

PROACT chooses from among the above three fine-grained transfer mechanisms. A `sys`-scoped release operation causes all PROACT buffers to be flushed.

D. Hardware Support for PROACT

In this study, we implement a software prototype of PROACT because it allows us to validate the concept across a wide range of GPU generations and interconnects, which would be impossible to simulate in a reasonable time. With additional hardware support, we envision an implementation where the data readiness counters are provisioned in a dedicated memory structure that is initialized by the PROACT runtime and associated with the base and bound of the PROACT-enabled region located in GPU memory. Local writes issued to a region tracked by PROACT automatically update the corresponding counter, replacing the explicit instructions added by PROACT instrumentation to maintain counters in our prototype. When a counter decrements to zero, hardware signals a transfer agent to initiate a transfer of the corresponding chunk. The transfer agent itself can be realized as a simplified DMA engine with descriptors for the geometry of each transfer prepared in advance in memory by the PROACT runtime. The salient aspect of a hardware design is that transfers are triggered automatically and without the need for interaction with GPU drivers running on the host CPU. Because PROACT can be prototyped entirely in software (with software overheads quantified later in Figure 8), we leave microarchitectural details of a PROACT hardware realization for future work and focus on demonstrating the efficacy and generality of the PROACT approach across numerous GPU and interconnect topologies using our software prototype.

System	4x Kepler	4x Pascal	4x Volta	16x Volta
GPU	Tesla K40m	Tesla P100	Tesla V100	Tesla V100
GPU Arch	Kepler	Pascal	Volta	Volta
#GPUs	4	4	4	16
Interconnect	PCIe3.0	NVLink	NVLink2	NVSwitch
Bidirectional BW per GPU	16GB/s aggregate	150GB/s aggregate	300GB/s aggregate	300GB/s aggregate
#Cores (SMs)	15	56	80	80
TFLOPS	1.43	5.3	7.8	7.8
BW (GB/sec)	288.4	720	920	920
Mem Cap (GB)	12	16	16	32

TABLE I: Key characteristics of the GPUs used in the experiments and their interconnect topologies in test systems.

IV. Experimental Methodology

To evaluate PROACT's software prototype, we perform our experiments across three 4-GPU platforms (4× Kepler, 4× Pascal, and 4× Volta) and one 16-GPU platform (16× Volta) with key characteristics described in Table I [12], [16]–[20].

A. PROACT Code Framework

The PROACT software prototype is demonstrated in Listing 1. The compiler automatically incorporates PROACT configuration parameters from the profiling run while inserting code to enable PROACT transfers. It generates both inline and decoupled versions of the code. PROACT also allocates meta-data structures to track data generation. The framework supports an arbitrary mapping between thread blocks and copy chunks. The block-to-address mapping is provided by proact_ds.mapping, which can point to the utility code PROACT provides for common mappings, such as one-to-one, stride, and stencil patterns or user-defined mappings.

In proact_init(), all atomic counters are initialized to the number of CTAs that write to the corresponding data chunk. During execution, PROACT then decrements the counter associated with a chunk. The last decrement triggers the copy of that chunk. Chunk size is selected automatically by compile-time profiling, empirically balancing the overhead of contention on atomic counters, copy initiation, and bandwidth utilization.

```
void proact_init(proact_ds* u_proact_ds) {
  u_proact_ds.xfer_mech = xfer_mech_from_profiler;
  u_proact_ds.counters = new counters[num_chunks];
  for chunk = 0 to num_chunks-1
      u_proact_ds.counters[chunk].value =
          chunk_val_from_profiler[chunk];

  if(u_proact_ds.xfer_mech == Polling) {
      u_proact_ds.bitmap = new bool[num_chunks];
      // copies data after bitmap values are set
      launch_polling_kernel<<<...>>>(u_proact_ds);
  }
}

__global__ void user_kernel_inline(proact_ds*
    u_proact_ds, ...) {
  // Generated from "ptr[tid] = computation(tid);"
  temp = computation(tid);
  u_proact_ds.region1[gpu0_id][tid] = temp;
  u_proact_ds.region1[gpu1_id][tid] = temp;
  u_proact_ds.region1[gpu2_id][tid] = temp;
  u_proact_ds.region1[gpu3_id][tid] = temp;
}

__global__ void user_kernel_decoupled(proact_ds*
    u_proact_ds, int* dest_arr, ...) {
  ptr = u_proact_ds.region1[devid]; //global memory
  ptr[tid] = computation(tid);

  // Code below added by compiler
  if(first_thread_of_cta) {
    if(u_proact_ds.xfer_mech == Dedicated_HW) {
        counter[chunk]--;
    } else if(u_proact_ds.xfer_mech == CDP) {
      atomicDec(counter[chunk]);
      if(counter[chunk] == 0) {
          src = map(ctaid, devid);
          dest0 = map(ctaid, peerGPU0);
          dest1 = map(ctaid, peerGPU1);
          dest2 = map(ctaid, peerGPU2);
          copy_kernel<<<nblockcdp, nthcdp>>>(src,
              ctaid, dest0, dest1, dest2);
      }
    } else if(u_proact_ds.xfer_mech == Polling) {
      atomicDec(counter[chunk]);
      if(counter[chunk] == 0)
          set bitmap[chunk];
      }
    }
  }
}

int main() {
  //Memory allocation by user for PROACT
  cudaMalloc4GPU(base_addr_0, base_addr_1,
      base_addr_2, base_addr_3);
  proact_ds u_proact_ds;
  u_proact_ds.ngpu = 4;
  //one-to-one mapping
  u_proact_ds.mapping = proact_contiguous;
  u_proact_ds.region1 = {base_addr_0, base_addr_1,
      base_addr_2, base_addr_3}
  // ... other initialization parameters
  proact_init(&u_proact_ds);

  if(XFR_METHOD == inline)
    user_kernel_inline<<<ncta, nthreads>>>(&
        u_proact_ds,...);
  else
    user_kernel_decoupled<<<ncta, nthreads>>>(&
        u_proact_ds,...);
}
```

Listing 1: PROACT sample code for 4 GPUs.

B. Evaluated Design Alternatives

To demonstrate the dynamic benefits of PROACT, we compare it to several different static multi-GPU programming approaches.

cudaMemcpy: In this paradigm, the computation kernel is followed by a `cudaMemcpy` call that duplicates data structures among all GPUs as needed. By the initiation of the following kernel, all data structures accessed by that kernel are resident in local GPU memory; there are no remote accesses. However, there is also no overlap between data transfers and compute.

Unified Memory (UM): We port our workload implementations to use UM by replacing conventional memory allocations with UM allocations, removing explicit data transfers, and adding the required inter-GPU synchronization. UM provides hint APIs to allow expert users to try to avoid the page faults and their large respective overheads. We hand-tested various hinting strategies, including data prefetching, read-replication, and pre-population of the GPU page tables to reduce fault overheads, making a best-effort attempt to optimize each application.

PROACT-inline: For PROACT-inline, remote stores are injected directly into the source kernel to push data to remote GPUs (similar to P2P stores), as shown in `user_kernel_inline` in Listing 1, rather than relying on the decoupled transfer agent.

PROACT-decoupled: PROACT-decoupled uses the full flexibility of the PROACT mechanisms, using the profiling tools to select the best data transfer method (Polling vs. CUDA Dynamic Parallelism) and other transfer parameters for each platform and application (number of transfer threads and transfer granularity).

Infinite Interconnect BW: Finally, we include a limit study that shows the performance when data transfers are instantaneous. It represents the maximum speedup that applications can attain from optimizing data movement. Here applications enjoy the benefit of fine-grained memory copies, but the data transfer time and the overhead of fine-grained tracking are neglected. Not attainable in practice, this study helps us understand how close each multi-GPU programming paradigm comes to a theoretical maximum performance. We compute this performance bound by using the bulk transfer implementation of each workload and then discounting its execution by the time spent performing data copies with `cudaMemcpy`.

C. Benchmarks

To evaluate PROACT, we implement multiple versions of workloads across various scientific domains and microbenchmarks to demonstrate the differences between the PROACT transfer mechanisms. All our benchmarks are compiled using CUDA 9.1 [21].

Microbenchmarks: Our microbenchmarks consist of a synthetic compute kernel running on a *source GPU* and generating data needed in its entirety by the *destination GPUs* for the next phase. Because the opportunity to overlap data transfer

and kernel execution is maximized when the duration of the compute kernel and transfer are equal, we tune the compute time of the synthetic kernel to match the transfer time under `cudaMemcpy()` transfers. Note that for this study, an ideal interconnect with zero latency and infinite bandwidth will be able to perform the data transfers instantaneously, resulting in a speedup of 2× over a real interconnect. To obtain a reasonable size of the compute kernel that can meet this criterion, we fix the total amount of data to be transferred at 256MB and tune the compute across various GPU generations. We then instrument the compute kernel on the source GPU to track its data production and initiate transfers via both PROACT's decoupled mechanisms.

MBIR X-ray CT: Model-Based Iterative Reconstruction (MBIR) is a computational technique that can produce high-quality images with low X-ray dose but at a high computational cost. We study an algorithm similar to that used in the FDA-approved GE Veo CT system, the only MBIR system approved for clinical use.

Page Rank: Page Rank assigns a 'Page-rank score' to web pages based on their importance. Our benchmark assigns page-rank scores to articles in the Wikipedia dataset [22].

Single Source Shortest Path (SSSP): Shortest Path algorithms are used to navigate between physical locations, such as in a road network [23]. In each iteration, every vertex computes its shortest distance from the source vertex using the Bellman-Ford algorithm [24]. We compute SSSP on the HV15R dataset [22].

Alternating Least Squares (ALS): ALS is widely used to perform matrix factorization in recommender systems. The algorithm is iterative. In each iteration, it fixes the user matrix and optimizes the item matrix and vice versa. Our study performs ALS using Stochastic Gradient Descent for vertices in the HV15R dataset [22].

Jacobi Solver: The Jacobi Solver iteratively solves a system of linear equations of the form $Ax = b$, where A is the coefficient matrix, b is a constant vector, and x is the required solution vector. The solver iteratively computes the solution vector using Jacobi's method [25]. We performed our study on the Jacobi solver for banded matrices, which arise widely in finite element analysis [26], [27].

V. Results

We first analyze PROACT's decoupled transfer mechanisms (Polling and CUDA Dynamic Parallelism) and compare them against `cudaMemcpy` in a microbenchmark study to understand their performance variation across different GPU architectures. Since PROACT's inline mechanism performs transfers at the granularity in which the data is generated, we do not include it in the microbenchmark study. However, all these transfer mechanisms are included in the results we present later. Also note that the PROACT framework picks the best out of the decoupled and inline variants as the transfer mechanism. We present them separately in the results for clarity.

A. Microbenchmarking Decoupled Transfer Mechanisms

Figure 6 shows the performance of the microbenchmark described in Section IV-C when varying the granularity for decoupled transfers from 4KB to 256MB, where each source thread block generates 4KB of data.

For the Kepler-based system and CDP case, the performance curve exhibits three regions. When the transfers happen at fine granularity (less than 16KB per chunk), performance is *initiation-bound*. Initiation overheads dominate, and there is a net slowdown relative to `cudaMemcpy`. The transfers are *bandwidth-bound* from 16KB to 1MB per chunk, and proactive transfers reach their peak speedup of 1.6×. As the granularity increases beyond 1MB, we enter a *tail–transfer–bound* region, where the left-over transfers, after the compute kernel completes (which we call the tail transfers) become significant, leading to a net slowdown. Polling substantially underperforms both `cudaMemcpy` and CDP on Kepler due to GPU resources wasted by numerous fruitless poll loops. The effect of utilizing a portion of GPU's compute and memory resources for polling loops is much more detrimental in the case of Kepler than Pascal or Volta because of lower compute and memory bandwidth availability.

For the Pascal-based system, CDP matches the trend of the Kepler-based system, offering a peak speedup of 1.8× in the bandwidth-bound region. However, polling performs even better, attaining up to 1.9× speedup once the transfer granularity is large enough so that the delay of iterating over the polling bitmap is amortized over the transfer operations.

CDP results in slowdowns at low granularities in the Volta-based system while achieving a 1.5× speedup at large granularities. The initiation overhead of dynamic kernels is higher on Volta than the other architectures. Polling offers higher speedups at nearly all granularities.

On current GPUs, launching dynamic kernels requires intervention from the host driver. As such, the degree to which dynamic kernel launches impact the parent's performance depends on the GPU hardware and the driver. We have observed this cost to vary substantially across GPU generations. Furthermore, the degree the data transfer kernel interferes with the parent computation depends on the number of warps in the data transfer kernel and again varies across GPU platforms. Thus, per GPU platform, the highest-performing data transfer mechanism varies across transfer granularity, transfer mechanism (DMA vs. fine-grained), and transfer agent type (polling vs. CUDA dynamic parallelism)

B. End-to-End Performance on Full Applications

Figure 7 shows the speedups achievable on a 4-GPU system over a single GPU across three different GPU generations, and Table II depicts the corresponding PROACT configurations.

With programmer-provided `cudaMemAdvise` hints, UM can substantially improve performance over a single GPU for some applications. However, since the hardware support for page faulting and migration was added only in the Pascal architecture, Kepler, an earlier architecture, uses a more

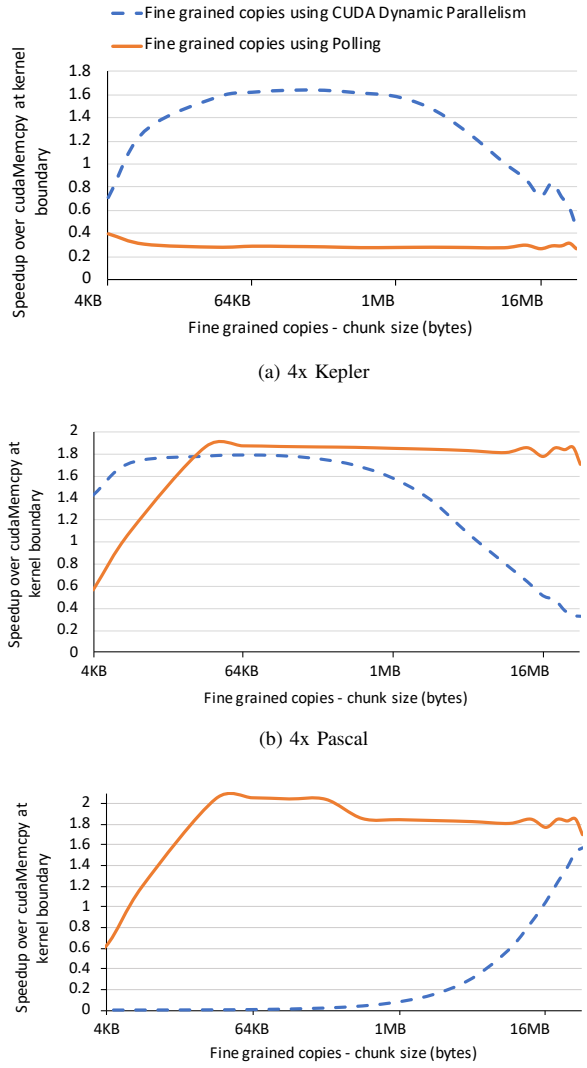


Fig. 6: **Performance of the PROACT microbenchmarks showing the effect of decoupled transfer paradigm and transfer granularity.**

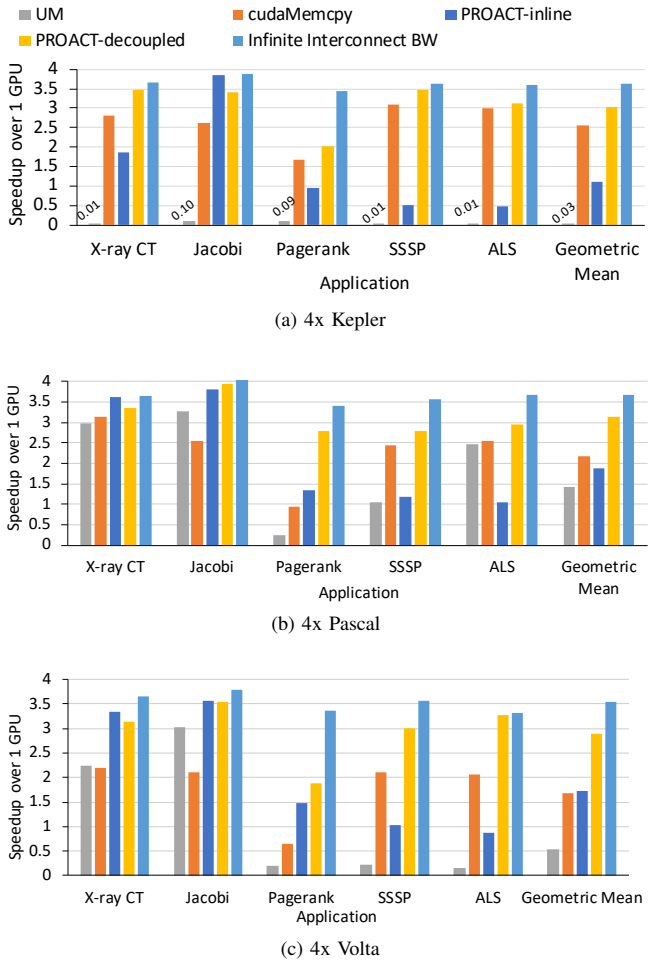


Fig. 7: **4-GPU speedup under each data transfer method for different hardware configurations.**

primitive version of Unified Memory, resulting in lower performance. For Jacobi, UM even outperforms `cudaMemcpy` duplication. The large number of sporadic accesses results in expensive page faulting and migration, causing UM to perform poorly for the Pagerank application in Pascal and Volta generations.

Performing explicit duplication via `cudaMemcpy` on these 4-GPU systems outperforms a single GPU for all applications except Pagerank, where it underperforms a single GPU on our Volta- and Pascal-based systems.

PROACT-inline outperforms decoupled transfers in just four cases where the applications naturally exhibit spatial locality. In Jacobi and X Ray-CT, GPU threads are scheduled so that data are produced densely in increasing address order, leading to excellent write coalescing within the SM. In these cases, the increase in interconnect efficiency achieved by decoupling data transfers is insufficient to overcome the overheads of the software implementation consuming SM resources. This marginal difference in performance is an artifact of prototyping PROACT-decoupled in software, and we expect a hardware implementation to outperform the inline variant in all cases.

In the remaining applications, where coalescing is poor due to more random memory update ordering, PROACT-decoupled performs best, as its larger-grain transfers always coalesce. For example, in ALS on 4x Volta, there are 26× more store transactions over the interconnect under PROACT-inline than PROACT-decoupled due to poor coalescing. Depending on the nature of the application and system architecture, PROACT picks the most suitable mechanism to perform the data transfers.

The abundant parallelism available in these applications is apparent in the near-linear scaling achieved under the

Application	4x Kepler	4x Pascal	4x Volta
X-ray CT	D 16kB 256 CDP	I	I
Jacobi	I	I	D 128kB 2048 Poll
Pagerank	D 16kB 256 CDP	D 1MB 4096 Poll	D 128kB 2048 Poll
SSSP	D 16kB 256 CDP	D 1MB 4096 Poll	D 128kB 2048 Poll
ALS	D 16kB 256 CDP	D 1MB 4096 Poll	D 128kB 2048 Poll

TABLE II: Best performing configuration as determined by the PROACT profiler. Each configuration is represented as transfer scheme (I: PROACT-inline and D: PROACT-decoupled), transfer granularity (range studied: 4kB to 16MB), transfer thread count (range studied: 32 to 8192), transfer mechanism (Poll: Polling and CDP: CUDA Dynamic Parallelism).

theoretical upper bound with infinite interconnect bandwidth. Regardless of GPU generation, the geometric mean theoretical opportunity is a 3.6× speedup over a single GPU for this set of applications. PROACT enables 3× speedup across all generations, capturing 83% of the possible opportunity. The story is more complicated for `cudaMemcpy`, where there is significantly more variation across GPU generations but with an average of 2.1× speedup over a single GPU. UM displays the most variable results, achieving good speedups for some applications, on some platforms, but on average underperforming even a single-GPU configuration. We conclude that, even with programmer-directed hints, traditional fault-based UM can not be relied upon to achieve peak performance. Thus, compared with different communication paradigms, the benefits of PROACT (both inline and decoupled) over cudaMemcpy come from its ability to overlap compute and communication, and the benefits over UM come from avoiding expensive page faulting and migration. PROACT-decoupled performs better than PROACT-inline when the higher interconnect efficiency achieved by decoupling outweighs the overheads on the compute kernel and vice-versa.

C. Decomposing PROACT's Performance

To further understand the performance, we analyze the overheads incurred by PROACT. PROACT-inline incurs no overhead on computation since there is no tracking instrumentation involved. For PROACT-decoupled, the largest performance overhead stems from prototyping the data tracking interface entirely in software. We determine the overhead by comparing runtime with tracking instrumentation but without the data transfers to the runtime of the theoretical infinite interconnect BW case. We report this overhead in Figure 8.

The figure reveals that the overhead averages between 10% and 15%, depending on the GPU platform. The variation across applications is significant, ranging from negligible overhead to as much as 40% for Pagerank. Note that the overhead is included in all our results (both in Figure 7 and Figure 10), which shows that PROACT-decoupled achieves substantial performance improvements despite the software overhead. A hardware implementation will alleviate the overheads on the compute kernel and can further improve performance, motivating its inclusion in future GPU architectures.

To understand how efficient PROACT is at overlapping data transfer and computation, we investigate the fraction of data

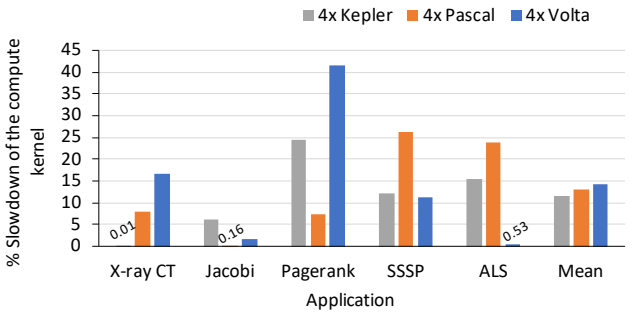


Fig. 8: **Compute slowdown due to PROACT for decoupled transfers (included in all our results).**

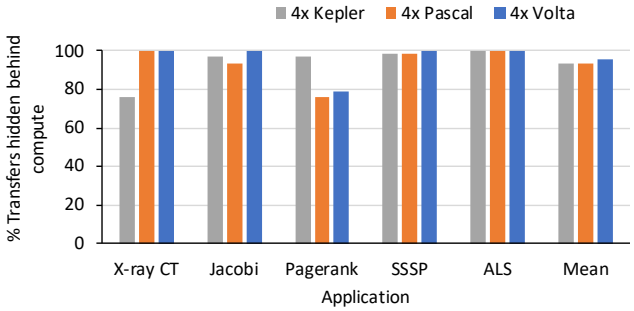


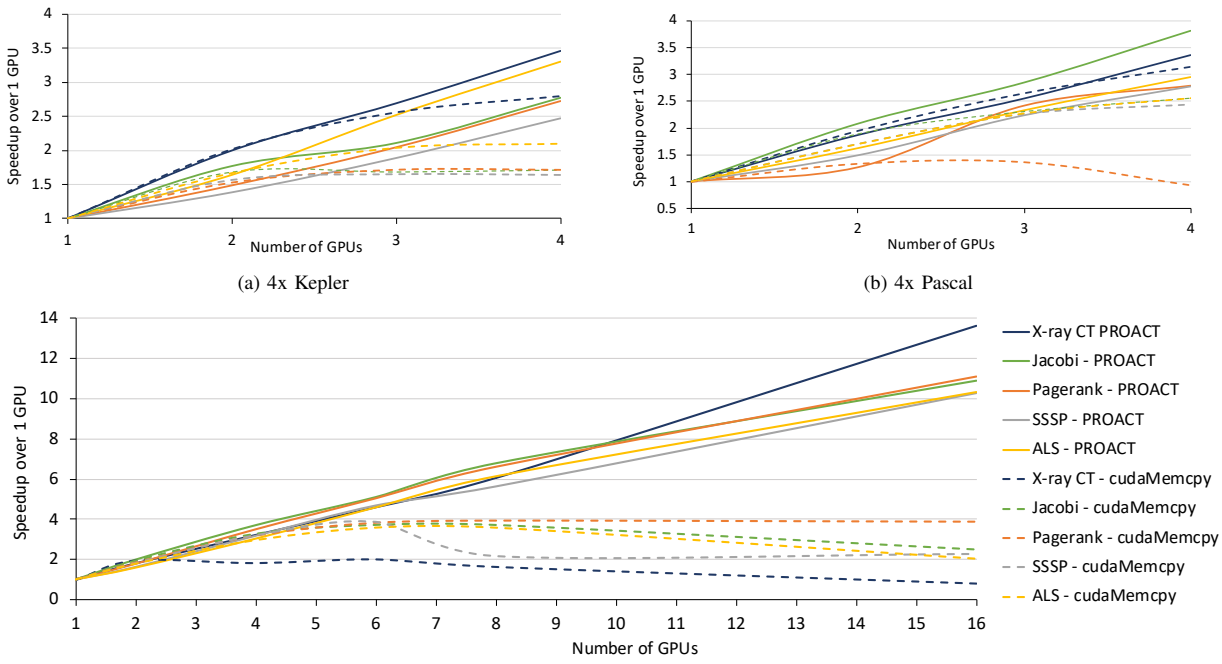
Fig. 9: **Transfer overlap achieved by PROACT.**

transfer overhead overlapped with computation on each of the 4-GPU systems. We obtain this result by executing each application with the instrumentation and initiation overheads of PROACT but eliding the stores that actually perform the data transfers to remote GPU memory. The difference between the runtime with and without the data transfer operations reveals the portion of transfer time that is not overlapped. We then determine the fraction of overlap by comparing the non-overlapped transfer time to the baseline duplication time with `cudaMemcpy`, reported in Figure 9.

Although there are variations among applications and platforms, Figure 9 reveals that PROACT always hides at least 75% of transfer time. In many cases, it can overlap nearly 100% of the communication, which will enable great scalability at high GPU counts relative to `cudaMemcpy` duplication. (which achieves no transfer overlap). Though UM with hints can help achieve overlap of compute and communication in theory, we note that this requires substantial programmer effort to track data generation at small granularities, schedule prefetch operations, and synchronize with computation.

D. Strong Scaling with PROACT

To this point, we have shown studies of PROACT's (inline and decoupled) performance impact for 4-GPU systems with different generations of GPUs and interconnects. We next consider the scalability of PROACT with multi-GPU system size. Figure 10 shows the absolute speedup that PROACT



(a) 4x Kepler

(b) 4x Pascal

(c) 16x Volta

Fig. 10: **Scalability achieved on different hardware configurations.**

achieves on Kepler, Pascal, and Volta for systems with up to 4, 4, and 16 GPUs, respectively.

In Figure 10, we omit unified memory results, which do not scale well (on average) and instead focus on the performance improvements with PROACT and `cudaMemcpy` duplication compared to the single-GPU performance achieved on the respective system. We see that when employing just two GPUs, performance is insensitive to the transfer method across all three platforms. This result is not surprising, as the total fraction of time spent in data transfers is insignificant with only two GPUs. However, as the number of GPUs increases to four (on the Kepler- and Pascal-based systems), and five-six on the Volta-based system, performance with `cudaMemcpy` flattens and even begins decreasing.

This effect manifests beyond just two GPUs on the Kepler system, which we hypothesize is because our PCIe-based Kepler system has the lowest inter-GPU bandwidth; thus, transfer overheads start affecting performance more quickly than in the other systems. We see that, on the Pascal system, performance with `cudaMemcpy` is generally competitive up to three GPUs before leveling off. When examining many GPU scaling on the Volta-based prototype system, `cudaMemcpy` scales to five GPUs before leveling off. PROACT, however, exhibits nearly linear multi-GPU scaling, indicating that it is doing a good job using the interconnect bandwidth effectively and overlapping GPU communication with computation. Overall, on our 16-GPU Volta system, PROACT achieves a mean speedup of 1.2×, 2.2×, and 5.3× over `cudaMemcpy` duplication at 4-, 8-, and 16-GPU configurations, respectively, while coming within 90%, 87%, and 77% of the theoretical application performance limit.

E. Discussion

The performance benefits of PROACT come from its ability to perform fine-grained data transfers, thus ensuring an overlap of compute and communication. Thus, applications satisfying the following conditions will benefit from PROACT: (1) strong scaling performance is limited by inter-GPU communication, (2) the write access pattern of the application is deterministic in nature, (3) programmers use structured programming, i.e., they can annotate the shared data structures that PROACT needs for tracking writes. For compute-bound applications, the compute overheads of PROACT software prototype may render it less suitable; however, such applications will still benefit from a PROACT hardware implementation. Also, for applications that perform sporadic accesses at small granularities, the overheads of initiating decoupled transfers will hurt the performance of PROACT-decoupled, and the profiler will pick PROACT-inline instead.

VI. Related Work

Accelerating scientific applications using GPUs [28]–[34] and optimizing GPU communication [9], [35], [36] has been widely studied, but in what are now legacy contexts, given the introduction of direct switch-connected LD/ST-based multi-GPU systems.

Communication in GPU systems: To our knowledge, ours is the first work to explore proactive direct stores to optimize

multi-GPU communication. Prior works range from optimizing communication between GPU thread blocks to methods that improve MPI communication among GPUs spread across multiple nodes. Xiao et al. [37] optimize inter-block GPU communication via barrier synchronization. CudaDMA [38] overlaps computation and data transfer between GPU global and shared memory. MVAPICH2-GPU [9] integrates CUDA data movement transparently with MPI. Potluri et al. provide intra-node MPI communication on multi-GPU nodes in [35]. CGCM [39] is an automated system for managing and optimizing CPU-GPU communication. Chen et al. [40] explore weak execution ordering to reduce host synchronization overhead.

Groute [41] provides an asynchronous multi-GPU model for irregular applications but does not consider optimizing fine-grained communications between GPUs. Even in MCM-GPU [42], where a package-level integration of multiple GPU modules is done, PROACT could help improve performance by effecting efficient copies to local DRAM partitions. Prior work [43]–[46] has explored various hardware and software mechanisms to improve multi-GPU performance. Many other works [47]–[51] perform NUMA-aware optimizations to improve GPU performance and perform hardware-based peer caching [52]–[56].

Performance of heterogeneous systems: Staged Memory Scheduling [57] decouples the primary tasks of a memory controller to improve the performance of CPU-GPU systems. Sutherland et al. [58] improve performance by using texture cache approximation on GPUs. While Schaa et al. [59] allow developers to accurately predict execution time of GPU scaling, Yao et al. [60] provide a theoretical analysis of multicore scalability. Unicorn [61] provides a parallel programming model for hybrid CPU-GPU clusters, while [62], [63] provide frameworks for automatic multi-GPU parallelization. CAWS [64] provides a scheduling policy, and Bhattacharjee et al. propose thread criticality predictors for parallel applications [65]. Dymaxion [66] attempts to overlap CPU-GPU PCIe transfer with data layout transformations on the GPU, while Lustig et al. [52] offer techniques for more effective GPU offloading. These topics are orthogonal and complementary to optimizing fine-grained multi-GPU communication, the subject of this work.

Auto-tuning and code generation: General literature in auto-tuners include automating the construction of compiler heuristics using machine learning [67], automatic tuning of heuristics for code inlining [68], and a genetic algorithm approach for compiler optimizations [69]. Additionally, obtaining the heuristic scheduling algorithm automatically [70] and finding the shortest program to compute a function [71] have been studied previously. In the GPU realm, multiple works have explored application-specific auto-tuning for GPUs [72]–[78], but this work provides a new methodology for scaling multi-GPU performance. Spafford et al. attempt to improve load balancing and bus utilization [79], yet they do not attempt to overlap compute with communication. CLTune [80] provides a generic auto-tuner for OpenCL Kernels but does not consider inter-device communication. While different works have explored code generation for specific GPU applications [81]–[85], we attempt to create a generic mechanism allowing all applications to scale.

VII. Conclusion

In this work, we proposed PROACT, a hardware and software system that improves multi-GPU performance by overcoming the limitations of existing inter-GPU communication mechanisms. PROACT combines the flexibility of peer-to-peer transfers with the efficiency of bulk transfers to enable interconnect friendly data transfers while hiding transfer latencies. Demonstrated across three different 4-GPU system architectures, PROACT provides an average speedup of 3.0× over a single-GPU implementation, capturing 83% of the theoretical limit. We also show how PROACT provides dramatic scalability improvements on next-generation systems with large GPU counts, achieving an 11.0× average speedup over a single-GPU implementation while capturing 77% of the available opportunity. Scalable multi-GPU programming is no easy task; to maximize programmer productivity, runtime systems like PROACT will be necessary to enable rapid development cycles while leveraging next-generation architectural improvements in future GPUs.

VIII. Acknowledgments

The authors thank the anonymous reviewers for their feedback. This work was supported in part by the Center for Applications Driving Architectures (ADA), one of six centers of JUMP, a Semiconductor Research Corporation program cosponsored by DARPA. It was also supported by the NIH Grant U01 EB018753.

References

[1] S. Breß, M. Heimel, N. Siegmund, L. Bellatreche, and G. Saake, "GPU-accelerated Database Systems: Survey and Open Challenges," in *Transactions on Large-Scale Data-and Knowledge-Centered Systems (TLDKS)*, 2014.

[2] A. Eklund, M. Andersson, and H. Knutsson, "fMRI Analysis on the GPU—Possibilities and Challenges," in *Computer Methods and Programs in Biomedicine*, vol. 105, pp. 145–161, 2012.

[3] G. M. Striemer and A. Akoglu, "Sequence Alignment with GPU: Performance and Design Challenges," in *Intl. Parallel & Distributed Processing Symposium (IPDPS)*, 2009.

[4] K. E. Niemeyer and C.-J. Sung, "Recent Progress and Challenges in Exploiting Graphics Processors in Computational Fluid Dynamics," in *The Journal of Supercomputing*, vol. 67, pp. 528–564, 2014.

[5] X. Ren, D. Lustig, E. Bolotin, A. Jaleel, O. Villa, and D. Nellans, "HMG: Extending Cache Coherence Protocols Across Modern Hierarchical Multi-GPU Systems," in *Intl. Symposium on High Performance Computer Architecture (HPCA)*, 2020.

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

[7] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable Parallel Programming with CUDA. Is CUDA the Parallel Programming Model That Application Developers Have Been Waiting For?," *Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[8] B. Goglin, "Improving Message Passing over Ethernet with I/OAT Copy Offload in Open-MX," in *Intl. Conference on Cluster Computing (CLUSTER)*, 2008.

[9] H. Wang, S. Potluri, M. Luo, A. K. Singh, S. Sur, and D. K. Panda, "MVAPICH2-GPU: Optimized GPU to GPU Communication for InfiniBand Clusters," *Computer Science-Research and Development*, vol. 26, no. 3-4, p. 257, 2011.

[10] S. Jeaugey, "NCCL 2.0," *GPU Technology Conference (GTC)*, 2017.

[11] S. Potluri, N. Luehr, and N. Sakharnykh, "Simplifying Multi-GPU Communication with NVSHMEM," in *GPU Technology Conference (GTC)*, 2016.

[12] PCISIG, "PCI Express Base Specification Revision 3.1a Specifications," 2010. pcisig.com/specifications, last accessed on 04/23/2021.

[13] NVIDIA, "NVLINK Fabric for Advanced Multi-GPU Processing," 2018. nvidia.com/en-us/data-center/nvlink/, last accessed on 04/23/2021.

[14] AMD, "Infinity Fabric (IF)," 2019. amd.com/en/technologies/infinity-architecture, last accessed on 11/23/2020.

[15] S. Jones, "Introduction to Dynamic Parallelism," in *GPU Technology Conference (GTC)*, 2012.

[16] NVIDIA, "NVIDIA DGX-1 Essential Instrument of AI Research." nvidia.com/en-us/data-center/dgx-1/, last accessed on 04/23/2021.

[17] G. Dearth and V. Venkatraman, "Inside DGX-2." on-demand.gputechconf.com/gtc/2018/presentation/s8688-extending-the-connectivity-and-reach-of-the-gpu.pdf, last accessed on 04/23/2021.

[18] NVIDIA, "NVIDIA Tesla GPU Accelerators." nvidia.com/content/tesla/pdf/NVIDIA-Tesla-Kepler-Family-Datasheet.pdf, last accessed on 04/23/2021.

[19] NVIDIA, "NVIDIA Tesla P100 Whitepaper." images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf, last accessed on 04/23/2021.

[20] NVIDIA, "NVIDIA Tesla V100 Datasheet." nvidia.com/content/PDF/Volta-Datasheet.pdf, last accessed on 04/23/2021.

[21] NVIDIA, "NVIDIA CUDA Compiler Driver NVCC," 2018. docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html, last accessed on 04/23/2021.

[22] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *Transactions on Mathematical Software*, vol. 38, no. 1, pp. 1–25, 2011.

[23] H. Hu, D. L. Lee, and V. Lee, "Distance Indexing on Road Networks," in *Intl. Conference on Very Large Data Bases (VLDB)*, 2006.

[24] J. Kepner and J. Gilbert, *Graph Algorithms in the Language of Linear Algebra*. SIAM, 2011.

[25] H. Rutishauser, "The Jacobi Method for Real Symmetric Matrices," *Numerische Mathematik*, vol. 9, no. 1, pp. 1–10, 1966.

[26] A. D. Mirlin, Y. V. Fyodorov, F.-M. Dittes, J. Quezada, and T. H. Seligman, "Transition from Localized to Extended Eigen States in the Ensemble of Power-Law Random Banded Matrices," *Physical Review E*, vol. 54, no. 4, p. 3221, 1996.

[27] J. Krüger and R. Westermann, "Linear Algebra Operators for GPU Implementation of Numerical Algorithms," in *ACM SIGGRAPH 2005 Courses*, p. 908–916, 2005.

[28] D. Goddeke, H. Wobker, R. Strzodka, J. Mohd-Yusof, P. McCormick, and S. Turek, "Co-Processor Acceleration of an Unmodified Parallel Solid Mechanics Code with FEASTGPU," *Intl. Journal of Computational Science and Engineering (IJCSE)*, vol. 4, no. 4, p. 254–269, 2009.

[29] Q. Xiong, B. Li, F. Chen, J. Ma, W. Ge, and J. Li, "Direct Numerical Simulation of Sub-Grid Structures in Gas-Solid Flow—GPU Implementation of Macro-Scale Pseudo-Particle Modeling," *Chemical Engineering Science*, vol. 65, no. 19, pp. 5356–5365, 2010.

[30] D. Göddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, S. H. Buijssen, M. Grajewski, and S. Turek, "Exploring Weak Scalability for FEM Calculations on a GPU-enhanced Cluster," *Parallel Computing*, vol. 33, no. 10, pp. 685–699, 2007.

[31] J. Bolz, I. Farmer, E. Grinspun, and P. Schröoder, "Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid," *ACM Transactions on Graphics (TOG)*, 2003.

[32] I. S. Ufimtsev and T. J. Martinez, "Quantum Chemistry on Graphical Processing Units. Strategies for Two-Electron Integral Evaluation," *Journal of Chemical Theory and Computation*, vol. 4, no. 2, pp. 222–231, 2008.

[33] A. Corrigan, F. F. Camelli, R. Löhner, and J. Wallin, "Running Unstructured Grid-based CFD Solvers on Modern Graphics Hardware," *Intl. Journal for Numerical Methods in Fluids*, vol. 66, no. 2, pp. 221–229, 2011.

[34] R. Salomon-Ferrer, A. W. Gotz, D. Poole, S. Le Grand, and R. C. Walker, "Routine Microsecond Molecular Dynamics Simulations with AMBER on GPUs," *Journal of Chemical Theory and Computation*, vol. 9, no. 9, pp. 3878–3888, 2013.

[35] S. Potluri, H. Wang, D. Bureddy, A. K. Singh, C. Rosales, and D. K. Panda, "Optimizing MPI Communication on Multi-GPU Systems using CUDA Inter-Process Communication," in *Intl. Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2012.

[36] S. Potluri, K. Hamidouche, A. Venkatesh, D. Bureddy, and D. K. Panda, "Efficient Inter-Node MPI Communication using GPUDirect RDMA for InfiniBand Clusters with NVIDIA GPUs," in *Intl. Conference on Parallel Processing (ICPP)*, 2013.

[37] S. Xiao and W.-c. Feng, "Inter-Block GPU Communication via Fast Barrier Synchronization," in *Intl. Parallel & Distributed Processing Symposium (IPDPS)*, 2010.

[38] M. Bauer, H. Cook, and B. Khailany, "CudaDMA: Optimizing GPU Memory Bandwidth via Warp Specialization," in *Intl. Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.

[39] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August, "Automatic CPU-GPU Communication Management and Optimization," in *Conference on Programming Language Design and Implementation (PLDI)*, 2011.

[40] J. Chen, Z. Huang, F. Su, J.-K. Peir, J. Ho, and L. Peng, "Weak Execution Ordering-Exploiting Iterative Methods on Many-Core GPUs," in *Intl. Symposium on Performance Analysis of Systems & Software (ISPASS)*, 2010.

[41] T. Ben-Nun, M. Sutton, S. Pai, and K. Pingali, "Groute: An Asynchronous Multi-GPU Programming Model for Irregular Computations," in *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2017.

[42] 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 *Intl. Symposium on Computer Architecture (ISCA)*, 2017.

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

[44] 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 *Intl. Symposium on Microarchitecture (MICRO)*, 2018.

[45] T. Baruah, Y. Sun, A. Dinçer, M. S. A. Mojumder, J. L. Abellán, Y. Ukidave, A. Joshi, N. Rubin, J. Kim, and D. Kaeli, "Griffin: Hardware-Software Support for Efficient Page Migration in Multi-GPU Systems," in *Intl. Symposium on High Performance Computer Architecture (HPCA)*, 2020.

[46] H. Kim, J. Sim, P. Gera, R. Hadidi, and H. Kim, "Batch-Aware Unified Memory Management in GPUs for Irregular Workloads," in *Intl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[47] K. Zhang, R. Chen, and H. Chen, "NUMA-Aware Graph-Structured Analytics," in *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2015.

[48] N. Agarwal, D. Nellans, M. Stephenson, M. O'Connor, and S. W. Keckler, "Page Placement Strategies for GPUs Within Heterogeneous Memory Systems," in *Intl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[49] B. Lepers, V. Quéma, and A. Fedorova, "Thread and Memory Placement on NUMA Systems: Asymmetry Matters," in *USENIX Annual Technical Conference (USENIX ATC)*, 2015.

[50] I. Psaroudakis, T. Scheuer, N. May, A. Sellami, and A. Ailamaki, "Scaling Up Concurrent Main-Memory Column-Store Scans: Towards Adaptive NUMA-Aware Data and Task Placement," *Proceedings of the VLDB Endowment*, 2015.

[51] A. Drebes, K. Heydemann, N. Drach, A. Pop, and A. Cohen, "Topology-Aware and Dependence-Aware Scheduling and Memory Allocation for Task-Parallel Languages," *Transactions on Architecture and Code Optimization (TACO)*, 2014.

[52] D. Lustig and M. Martonosi, "Reducing GPU Offload Latency via Fine-Grained CPU-GPU Synchronization," in *Intl. Symposium on High Performance Computer Architecture (HPCA)*, 2013.

[53] J. Power, A. Basu, J. Gu, S. Puthoor, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood, "Heterogeneous System Coherence for Integrated CPU-GPU Systems," in *Intl. Symposium on Microarchitecture (MICRO)*, 2013.

[54] M. Dashti and A. Fedorova, "Analyzing Memory Management Methods on Integrated CPU-GPU Systems," in *Intl. Symposium on Memory Management (ISMM)*, 2017.

[55] I. Singh, A. Shriraman, W. W. Fung, M. O'Connor, and T. M. Aamodt, "Cache Coherence for GPU Architectures," in *Intl. Symposium on High Performance Computer Architecture (HPCA)*, 2013.

[56] A. Basu, S. Puthoor, S. Che, and B. M. Beckmann, "Software Assisted Hardware Cache Coherence for Heterogeneous Processors," in *Intl. Symposium on Memory Systems (MEMSYS)*, 2016.

[57] R. Ausavarungnirun, K. K.-W. Chang, L. Subramanian, G. H. Loh, and O. Mutlu, "Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems," in *International Symposium on Computer Architecture (ISCA)*, 2012.

[58] M. Sutherland, J. San Miguel, and N. E. Jerger, "Texture Cache Approximation on GPUs," in *Workshop on Approximate Computing Across the Stack (WAX)*, 2015.

[59] D. Schaa and D. Kaeli, "Exploring the Multiple-GPU Design Space," in *Intl. Parallel & Distributed Processing Symposium (IPDPS)*, 2009.

[60] E. Yao, Y. Bao, G. Tan, and M. Chen, "Extending Amdahl's Law in the Multicore Era," *SIGMETRICS Performance Evaluation Review*, vol. 37, no. 2, pp. 24–26, 2009.

[61] T. Beri, S. Bansal, and S. Kumar, "The Unicorn Runtime: Efficient Distributed Shared Memory Programming for Hybrid CPU-GPU Clusters," *Transaction on Parallel and Distributed Systems*, vol. 28, no. 5, pp. 1518–1534, 2017.

[62] J. Cabezas, L. Vilanova, I. Gelado, T. B. Jablin, N. Navarro, and W.-m. W. Hwu, "Automatic Parallelization of Kernels in Shared-Memory Multi-GPU Nodes," in *Intl. Conference on Supercomputing (SC)*, 2015.

[63] T. Ben-Nun, E. Levy, A. Barak, and E. Rubin, "Memory Access Patterns: The Missing Piece of the Multi-GPU Puzzle," in *Intl. Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2015.

[64] S.-Y. Lee and C.-J. Wu, "CAWS: Criticality-Aware Warp Scheduling for GPGPU workloads," in *Intl. Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2014.

[65] A. Bhattacharjee and M. Martonosi, "Thread Criticality Predictors for Dynamic Performance, Power, and Resource Management in Chip Multiprocessors," in *Intl. Symposium on Computer Architecture (ISCA)*, 2009.

[66] S. Che, J. W. Sheaffer, and K. Skadron, "Dymaxion: Optimizing Memory Access Patterns for Heterogeneous Systems," in *Intl. Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.

[67] M. W. Stephenson, *Automating the Construction of Complier Heuristics Using Machine Learning*. PhD thesis, Massachusetts Institute of Technology, 2006.

[68] J. Cavazos and M. F. O'Boyle, "Automatic Tuning of Inlining Heuristics," in *Supercomputing Conference (SC)*, 2005.

[69] S. R. Ladd, "Acovea: Analysis of Compiler Options Via Evolutionary Algorithm," *Describing the Evolutionary Algorithm*, 2007. https://github.com/Acovea/ last accessed on 04/23/2021.

[70] J. E. B. Moss, P. E. Utgoff, J. Cavazos, D. Precup, D. Stefanovic, C. E. Brodley, and D. Scheeff, "Learning to Schedule Straight-line Code," in *Advances in Neural Information Processing Systems*, 1998.

[71] H. Massalin, "Superoptimizer: A Look at the Smallest Program," in *Intl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1987.

[72] A. Nukada and S. Matsuoka, "Auto-Tuning 3-D FFT Library for CUDA GPUs," in *Conference on High Performance Computing Networking, Storage and Analysis (SC)*, 2009.

[73] Y. Li, J. Dongarra, and S. Tomov, "A Note on Auto-Tuning GEMM for GPUs," in *Intl. Conference on Computational Science (ICCS)*, 2009.

[74] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams, "An Auto-Tuning Framework for Parallel Multicore Stencil Computations," in *Intl. Parallel & Distributed Processing Symposium (IPDPS)*, 2010.

[75] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil Computation Optimization and Auto-Tuning on State-of-the-art Multicore Architectures," in *Supercomputing Conference (SC)*, 2008.

[76] A. Davidson and J. Owens, "Toward Techniques for Auto-Tuning GPU Algorithms," in *Intl. Workshop on Applied Parallel Computing (PARA)*, 2010.

[77] J. Li, G. Tan, M. Chen, and N. Sun, "SMAT: an Input Adaptive Auto-tuner for Sparse Matrix-vector Multiplication," in *Conference on Programming Language Design and Implementation (PLDI)*, 2013.

[78] P. Guo and L. Wang, "Auto-Tuning CUDA Parameters for Sparse Matrix-vector Multiplication on GPUs," in *Intl. Conference on Computational and Information Sciences (ICCS)*, 2010.

[79] M. Steuwer, T. Remmelg, and C. Dubach, "Matrix Multiplication Beyond Auto-Tuning: Rewrite-based GPU Code Generation," in *Intl. Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, 2016.

[80] C. Nugteren and V. Codreanu, "CLTune: A Generic Auto-tuner for OpenCL Kernels," in *Intl. Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, 2015.

[81] J. Holewinski, L.-N. Pouchet, and P. Sadayappan, "High-Performance Code Generation for Stencil Computations on GPU Architectures," in *Intl. Conference on Supercomputing (SC)*, 2012.

[82] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gomez, C. Tenllado, and F. Catthoor, "Polyhedral Parallel Code Generation for CUDA," *ACM Transactions on Architecture and Code Optimization (TACO)*, 2013.

[83] M. Christen, O. Schenk, and H. Burkhart, "Patus: A Code Generation and Auto-Tuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures," in *Intl. Parallel & Distributed Processing Symposium (IPDPS)*, 2011.

[84] E. Alerstam, W. C. Y. Lo, T. D. Han, J. Rose, S. Andersson-Engels, and L. Lilge, "Next-Generation Acceleration and Code Optimization for Light Transport in Turbid Media using GPUs," *Biomedical Optics Express*, vol. 1, no. 2, pp. 658–675, 2010.

[85] Y. Zhang and F. Mueller, "Auto-Generation and Auto-Tuning of 3D Stencil Codes on GPU Clusters," in *Intl. Symposium on Code Generation and Optimization (CGO)*, 2012.