A Simple Cache Coherence Scheme for Integrated CPU-GPU Systems

Ardhi Wiratama Baskara Yudha
University of Central Florida
Orlando, FL US
yudha@knights.ucf.edu

Reza Pulungan
Universitas Gadjah Mada
Yogyakarta, ID
pulungan@ugm.ac.id

Henry Hoffmann
University of Chicago
Chicago, IL US
hankhoffmann@cs.uchicago.edu

Yan Solihin
University of Central Florida
Orlando, FL US
Yan.Solihin@ucf.edu

***Abstract*—This paper presents a novel approach to accelerate applications running on integrated CPU-GPU systems. Many integrated CPU-GPU systems use cache-coherent shared memory to communicate. For example, after CPU produces data for GPU, the GPU may pull the data into its cache when it accesses the data. In such a *pull-based* approach, data resides in a shared cache until the GPU accesses it, resulting in long load latency on a first GPU access to a cache line. In this work, we propose a new, *push-based*, coherence mechanism that explicitly exploits the CPU and GPU producer-consumer relationship by automatically moving data from CPU to GPU last-level cache. The proposed mechanism results in a dramatic reduction of the GPU L2 cache miss rate in general, and a consequent increase in overall performance. Our experiments show that the proposed scheme can increase performance by up to 37%, with typical improvements in the 5–7% range. We find that even when tested applications do not benefit from the proposed approach, their performance does not decrease with our technique. While we demonstrate how the proposed scheme can co-exist with traditional cache coherence mechanisms, we argue that it could also be used as a simpler replacement for existing protocols.**

***Index Terms*—Cache coherence, GPU, Integrated CPU/GPU.**

I. Introduction

The abundance of transistors available on a die has led to new integrated processor designs combining CPUs and GPUs on a single chip [1]–[5]. This integration presents new opportunities for increased performance and ease of programming as the CPU and GPU can communicate through cache-coherent shared memory (CCSM) rather than through explicit data copying.

For most integrated CPU-GPU systems, the critical operation is moving data from CPU to GPU. The CPU-GPU relationship is different from peer cores in the traditional CPU shared-memory multiprocessor systems. In CPU-GPU systems, the CPU is a data producer and the GPU is a consumer. Adopting CCSM mechanisms from CPU multiprocessors is convenient but not efficient. GPU performance depends directly on how fast it can consume the CPU-produced data. Pulling data one block at a time upon a cache miss slows down data supply needed for high GPU performance. This clear producer-consumer relationship in CPU-GPU systems presents an opportunity for further performance gain by eliminating unnecessary cache coherence messages.

This paper proposes an alternative CPU-GPU cache coherence mechanism. Specifically, performance critical data is *homed* on the GPU. When the CPU writes to such data, the proposed protocol automatically forwards the data to the GPU memory hierarchy—instead of leaving it in the CPU memory. This proposal is an intuitive match for the CPU to GPU producer-consumer relationship, as it moves the data to the memory system where it will be consumed, reducing GPU miss rate, and improving overall performance.

While the basic idea of homed memory has appeared in many systems—from commercial products for distributed shared memory [6] and manycores [7] to research designs [8]—the integrated CPU-GPU system provides new challenges and opportunities. Specifically, prior approaches required that users explicitly declare the home core for different data ranges (e.g., in the TILE architecture [7]) or required added hardware complexity to automatically move data to an appropriate home (e.g., [8]). The unique nature of an integrated CPU-GPU system makes it possible to automatically translate existing programs to home memory and thus achieve the performance benefits of homing data on the GPU while requiring no extra work from users and keeping hardware complexity low.

Thus, in this paper we show how to support homed memory on an integrated CPU-GPU system. Our proposed approach requires no human intervention as it includes source to source translation that converts existing GPU programs to use our mechanism. Specifically, the translator changes existing memory allocation to allocate data that will be homed on the GPU in a specific data range. Data in that range cannot be cached on the CPU, only the GPU. Therefore, when the CPU attempts to write it (to produce data for the GPU consumption), there will be a write miss. Concurrently, the modified Translation Look-aside Buffer (TLB) will detect the address range and tell the CPU memory system to forward the write miss to the GPU memory system. The GPU then resolves the write miss and the produced data is stored in the GPU memory hierarchy. Then, when the GPU accesses it, the data is already in cache locally, reducing both compulsory misses and load latency on the first data access.

We test the proposed scheme by implementing it in the gem5-gpu simulator and compare it with recent proposals for the traditional CCSM between CPU and GPU. Across benchmarks from a range of suites, we find that our proposed approach can achieve performance improvements as high as 37%, with typical improvements in the range of 5–7%. While the proposed technique can co-exist with CCSM, we also believe it could be a simpler replacement for this approach.

II. Related Work

There is a rich set of papers in literature for *pull-based* data supply (e.g. prefetching), and to some extent *push-based* data supply (e.g. forwarding). Many prefetching techniques have been proposed both in hardware and software [9], including a technique that assumes that the integrated CPU-GPU system shares the last-level cache (LLC) [10], a dynamically adaptive prefetching strategy to reduce memory latency and improve better power efficiency [11], and a hybrid of software and hardware prefetching [12], which combines conventional stride prefetching and a hardware-implemented inter-thread technique, where each thread prefetches for threads located in other warps.

A GPU selective caching strategy that enables unified shared memory without hardware cache-coherence is proposed in [13]. This involves architectural improvements of selective caching by aggressive request coalescing, CPU-side coherent caching for GPU

uncacheable requests, and a CPU-GPU interconnect optimization to support variable-sized transfers. On the other hand, a heterogeneous system coherence for CPU-GPU system to address the effect of the huge number of GPU memory requests is proposed in [2]. This proposed mechanism replaces the standard directory with a region directory and inserts a region buffer into the L2 cache.

Push-based techniques could be found in the design of classical CPU multiprocessor systems, such as the DASH multiprocessor's *Deliver* instruction, KSR's poststore, and Multicube Notify, among others [14]. In general, it has been recognized that push-based techniques can sometimes outperform pull-based techniques if the sharing pattern is known at the producer more than at the consumer. Outside of CPU multiprocessors, direct cache access allows the network interface card (NIC) to push data it receives directly to the CPU caches for packet processing by the CPU [15]. While building upon past observations, this paper targets a different platform: CPU-GPU system where the CPU is the producer and GPU is the consumer. Therefore, the program transformation and architecture support in this paper are quite distinct from those in CPU multiprocessor and NIC cases.

While prefetching shares a goal of minimizing load latency, prefetching mechanisms are very different from, and complementary to, the mechanism proposed in this paper. The most related mechanism to ours is *remote store programming* (RSP) [7]. RSP is a memory model for multicores that allows data to be homed on particular cores, and data to be cached only on the home core. Our direct store shares similarity to RSP, but with several fundamental differences. First, RSP was designed for CPU multicores whereas direct store is designed for GPU. We argue that it is much better suited for GPU because the producer consumer relationships are well-defined and easily determined. Second, direct store works by simple modification to the cache coherence and can even replace traditional cache coherence between CPU and GPU, whereas RSP must rely on traditional CPU cache coherence. Third, thread migration is problematic in RSP, whereas GPU kernels will always execute in GPU and the lack of migration possibility between GPU and CPU simplifies the direct store coherence design.

III. Design and Implementation

In a conventional integrated CPU-GPU system, data is pulled to GPU on a load instruction. In contrast, we propose a scheme for pushing data directly to GPU cache to reduce load latency. To realize this scheme, several requirements arise: (1) A mechanism to identify data to be stored remotely, (2) separate memory allocation modes for such data, (3) mechanism in the memory system to handle such data correctly, and (4) a dedicated interconnection network to provide fast data delivery and to reduce the complexity of the cache coherence modifications. We will now describe our cache coherence scheme *direct store* in greater details.

A. Data Flow

The data flow of traditional cache coherence and of our proposed scheme are different. Figure 1 illustrates the difference in data movement between a traditional state-of-the-art CCSM used in integrated CPU-GPU and the proposed direct store. In the Hammer protocol, all CPU and GPU caches maintain cache coherence, except for the GPU L1 cache [16]. Software can still enforce coherence of GPU L1 caches by by writing through dirty data and flash invalidating at the time the kernel starts execution. Data movement in CCSM involves longer steps than in direct store (Figure 1 (left)). In CCSM, when a CPU core issues a store instruction *st x*, it also copies the data from the register file (RF) to its L1 cache. At a later time, there are three possibilities: the data remains in the CPU's L1 cache, it is moved to lower (L2) cache, or it is evicted to DRAM. On the other hand, with direct store, when the CPU issues a store *st x*, a copy is directly sent to the GPU L2 cache (Step 1 in Figure 1 (right)). If the GPU L2 cache is full, the system then writes data to DRAM.

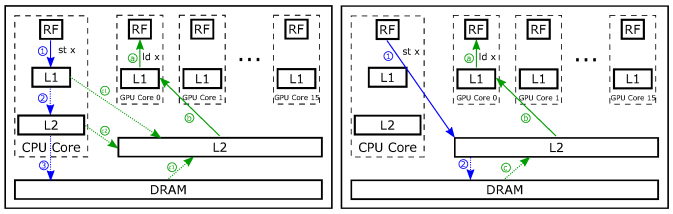

Fig. 1. Comparison of data movement between CPU and GPU under (left) CCSM and (right) the proposed direct store.

With CCSM, when GPU wants to consume data produced by CPU, it fetches it from the L1 cache. If L1 cache misses, GPU requests it from its L2 cache. If data is found in the L2 cache, it is fetched into the L1 cache (Step b in Figure 1 (left)). If the L2 cache misses, there are three possible places where data needs to be fetched from, as illustrated by Steps c1, c2, and c3 in Figure 1 (left). Thus CCSM's data movement is very different from what we propose in direct store. In our proposal, the GPU only needs to pull data to the L1 cache, since it is likely that it is already in the GPU L2 cache (Steps a and b in Figure 1 (right)). In summary, the main upside of direct store over CCSM is that the GPU does not need to pull data to its local cache, which reduces memory access time and reduces coherence traffic for providing the data to the GPU. We will now disucss the direct store mechanism in greater details.

B. Overall Design

There are five steps in the direct store scheme. The first is an automatic code translation, which modifies a program written for an integrated CPU/GPU to a program that can run on our proposed scheme. Secondly, when the program is running, it reserves and allocates a region of memory in the GPU L2 cache. The third step is address translation, which occurs when a CPU store targets the reserved memory. The address translation is accomplished by an added functionality of the TLB. The fourth step is a special cache coherence mechanism, which kicks in when a CPU store targets the reserved memory. The fifth is a new interconnection network that will bring data directly to the GPU L2 cache.

Intuitively, direct store increases the latency of CPU stores to memory regions designated for the GPU. In exchange, the data to be loaded by the GPU is moved into its memory hierarchy on the write. Thus, the protocol is designed to decrease GPU load latency (which is hard to hide) in exchange for increased CPU store latency (to which most programs are less sensitive).

C. Automatic Code Translation

We develop a code translator to implement our proposed scheme automatically from the baseline benchmarks that use no memory copy. The translator takes the source files of a program and identifies variables that will be accessed by the GPU. Specifically, all variable inferences in CUDA kernel invocations are scanned. The translator then searches for the pattern that matches the kernel call as `kernel_name<<<Dg, Db, Ns, S>>>`$(x_1, x_2, \ldots, x_n)$, where $x_1, x_2, \ldots, x_n$ are the variable names captured by the translator and stored in the temporary memory. Once all kernel invocations are

inspected, the translator scans each source code file to determine the amount of memory needed for each variable and the variable type. Finally, the translator searches the source code to find the memory declaration of the stored variables in the temporary memory.

The translator searches for memory declarations that use *malloc* and *cudamalloc* syntax. Whenever such a declaration is found, the translator modifies the memory declaration to use *mmap* and sets its memory size and the starting virtual address and saves this modification to the associated file. The translator increments the starting virtual address to the memory size needed by the variable, such that there is no overlapping starting virtual addresses for all variables. After all iterations are completed, the application is ready to be compiled in the standard way. By using this automatic code translator, there is no effort for the programmer to modify the source code to implement the direct store scheme.

D. Special Memory Allocation

The proposed method needs OS support for reserving exclusive memory regions. In the implementation, we use *mmap*, a Unix system call that maps files or devices to a virtual address space of the calling process. Our approach specifies the argument addr to high-order address bits and sets flags to MAP_FIXED. By, allocating high-order addresses for remotely stored data, we can signal to the TLB that this memory needs to be stored remotely to the GPU L2 cache.

E. Translation Look-aside Buffer

Figure 2 (left) illustrates the control flow of direct store targeting the GPU L2 cache. First, the CPU allocates data that will be read by the GPU with a special memory allocation. Using the mechanisms described above, we reserve bits of the virtual address space to represent data that is *homed* in the GPU's. This special data range can never be cached on the CPU side (so accesses from the CPU will always miss), but it can be cached on the GPU. When CPU stores target this memory, the memory management unit (MMU) translates the virtual address (VA) to a physical address (PA) by consulting the TLB. By definition, this memory will always result in a cache miss on the CPU. On detecting the special memory region's VA, the TLB signals the MMU to forward the store to the GPU (over a dedicated network), which resolves it. When a GPU wants to consume data stored by CPU, it simply issues a standard load.

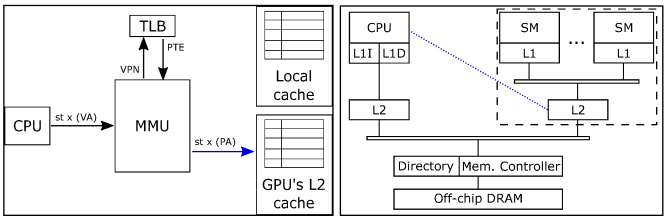

Fig. 2. Left: The control flow when the CPU issues a store instruction and the roles of the TLB; Right: the topology of the simulated system with a network connecting CPU and the GPU LLC (assumed to be L2 in this paper).

We modify the TLB by adding logic to detect high-order virtual addresses, which will use the memory range of the reserved system's virtual address space as described in the previous subsection. When a virtual address is referenced by a program, the TLB performs an address comparison to detect a high-order virtual address. When detected, the TLB sends a signal to the MMU indicating to the CPU's L1 cache controller to forward the store onto the GPU L2 cache.

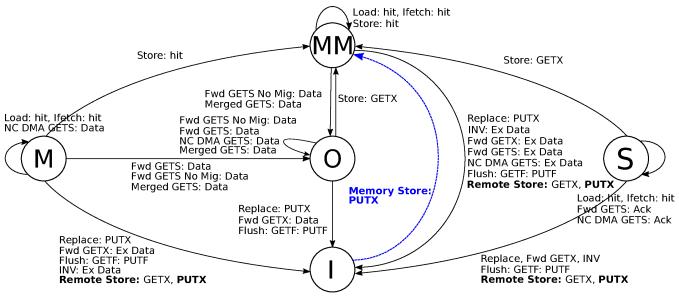

Fig. 3. The modified state-transition diagram of the Hammer protocol (based on [17]).

F. Cache Coherence Protocol

Since we add a network between the CPU and the GPU L2 cache, we need only a simple modification to the cache coherence protocol because data does not travel through the whole cache hierarchy and therefore does not affect the protocol much. However, for direct store to function correctly, we need to specify to the protocol the store state of the remotely stored data. Figure 3 depicts a modified state-transition diagram of AMD's Hammer cache coherence protocol [17], which consists of 5 stable states: MM, M, O, S, and I. State MM represents that the node has an exclusive hold on the cache block and the block is potentially locally modified (similar to conventional M state). State O indicates that the node owns the cache block, but has not modified the block, and no other node holds the block in exclusive mode, although sharers may exist. State M indicates that the cache block is held in exclusive mode, but has not been written to (similar to conventional E state), and no other node holds a copy of the block. Stores are not allowed in state M. State S indicates that the cache line holds the most recent, correct copy of the data, but other processors in the system may hold copies of the data in the shared state. The cache line can be read, but not written to in state S. State I indicates that the cache line is invalid and it does not hold a valid copy of the data.

We modify the protocol for handling a store instruction that targets remote location by adding new actions, depicted by bold text in the figure. The modification is minimal because CPU to GPU stores travel through the new network. When a remote store instruction arrives at the CPU's L1 cache targeting the GPU L2 cache, the protocol starts from state I and then data is forwarded directly from the CPU's L1 cache to GPU L2 cache without caching it into L1 cache through the newly added network that connects them. After data is sent through the network the protocol remains in state I.

To cover other possibilities, where the CPU might load data after remotely storing it, we add the ability to do a remote store from states S, M, and MM. All remote stores that begin from these states always go to state I. This is safe because when the protocol wants to flip from a beginning state to state I, it gets exclusive permission to the cache block needed. In the GPU L2 cache, after a store instruction is forwarded there, it begins in state I then always proceeds to state MM. This is safe because we make sure that the directly stored data will only be deposited in GPU L2 cache, not any other.

The GPU L2 cache has exactly the same protocol as the CPU. However, there is a single additional feature that makes it possible for the GPU to store data from a remote location. Every time a remote store arrives at the GPU L2 cache, it will transition from state I to MM as illustrated by the blue dashed-lined transition in Figure 3. This transition always starts from state I since before forwarding the

data, the CPU will issue GETX command. The store will be issued as PUTX action indicating the store is to the GPU L2 cache.

G. Interconnection Network

When a CPU store instruction targeting the GPU is detected by the TLB, it tells the MMU to direct the store to the GPU L2 cache. We therefore add a fast network directly connecting the CPU and the GPU L2 cache. Figure 2 (right) shows the newly added network, indicated by the dotted line. By connecting the CPU and GPU L2 cache directly, the data transfer will be faster because it bypasses much of the cache hierarchy and eliminates many coherence messages. Since we bypass many cache hierarchies and go straight to the GPU L2 cache, we need only introduce minor modifications on the cache coherence protocol, namely those related to CPU's L1 cache and GPU L2 cache. The newly added interconnection network will have exactly the same characteristics as the network used in many cache coherence systems to exchange the messages.

H. Implementing Direct Store without Traditional Cache Coherence

Although we have presented this scheme as a complement to existing coherence mechanisms (since we think that would be the most likely use case), it could be adopted as a simpler, stand-alone replacement for CPU to GPU communication. The proposed scheme is simple to implement because it requires fewer coherence messages than traditional protocols because the proposed scheme only allows memory shared between GPU and CPU to exist in the GPU cache. Thus, there will not be multiple sharers and so there are no transitions from (for example) shared states to exclusive states.

The proposed scheme could be easily attached to systems that use the state-of-the-art CCSM. This opens a new possibility for the programmer to use the new approach for only a particular type of data and other remaining data still uses the traditional approach. The programmer can set large variables to use this approach, since this will speed up the data movement and the remaining small-sized data to use CCSM system. Thus, the programmer benefits from using the fast CPU-GPU transfer of direct stores.

The proposed scheme could also replace the entire CCSM system and thus gains a simpler design with better performance. To do this we only need an interconnection network that directly connects the CPU to the GPU L2 cache and to remove the CCSM system's interconnection network. Finally, an existing program would just be run through our source-to-source translator, then compiled and run in the standard way. Thus, direct stores could completely replace more complicated CPU-GPU coherence schemes with no programmer burden and increased performance.

IV. Experimental Evaluation

Through experiments, we seek to answer two questions. First, does direct store improve performance over CCSM? We answer this question by measuring the execution time of both CCSM and our direct store for various applications, both for small inputs, which fit into the GPU LLC, and for large inputs, which exceed the LLC capacity. Second, does the proposed approach reduce cache misses overall? Here we measure the miss rate reductions for both CCSM and the proposed approach. In addition, we believe the proposed approach should specifically reduce compulsory misses, so we measure those for both approaches.

The results and analysis for all these experiments are included in subsequent sections. While omitted for space, we have also compared direct stores to prefetching and find that direct store's performance improvements there are even higher.

A. Simulation Infrastructure

Direct store is evaluated on gem5-gpu [18], a simulator that models tightly-coupled CPU-GPU systems. It merges two popular simulators: gem5 [19] and GPGPU-Sim [20]. Our baseline implementation in gem5-gpu is based on the configuration listed in Table I.

TABLE I
System configuration

CPU	
Cores	1
L1D cache	64KB, 2 ways
L1I cache	32KB, 2 ways
L2 cache	2MB, 8 ways
GPU	
SMs	16 - 32 lanes per SM @ 1.4Ghz
L1 cache	16KB + 48 KB shared memory, 4 ways
L2 cache	2MB, 16 ways, 4 slices
MEMORY	
Memory	2GB, 1 channel, 2 ranks, 8 banks @ 1GHz

All evaluations were performed using gem5-gpu in syscall-emulation mode. We use the Ruby memory system instead of the classic memory system. We simulate an integrated CPU-GPU system comprising of a CPU core and a GPU with 16 Fermi-like SMs. Each L1 data and instruction cache and L2 cache are private to each CPU core. For GPU, each L1 cache is private to each SM, while the L2 cache is shared by all SMs. Cache line size is 128 bytes across the whole system. The system has 2GB of total memory. The simulator provides full system coherence between CPU and GPU using the modified Hammer cache coherence protocol.

TABLE II
Benchmarks

Name	Small input	Big input	Suite	Shared
BP	1536	10000	Rodinia [21]	Yes
BF	4096	6000	Rodinia	No
GA	256x256	700x700	Rodinia	Yes
HT	64x64	512x512	Rodinia	Yes
KM	2000, 34 feat	5000, 34 feat.	Rodinia	Yes
LV	2	4	Rodinia	Yes
LU	256x256	512x512	Rodinia	Yes
NN	10691	42764	Rodinia	No
NW	160x160	320x320	Rodinia	Yes
PT	2500	5000	Rodinia	Yes
SR	256x256	512x512	Rodinia	Yes
ST	128x128x32	164x164x32	Parboil [22]	Yes
GC	power	delaunay-n15	Pannotia [23]	No
FW	256_16384	512_65536	Pannotia	No
MS	power	delaunay-n13	Pannotia	No
SP	power	delaunay-n13	Pannotia	No
BL	5000	10000	NVIDIA SDK	No
VA	50000	200000	NVIDIA SDK	No
BS	262144	524288	[24]	No
MM	256x256	900x900	[25]	No
MT	32x32	1600x1600	[25]	No
CH	150x150	600x600	[26]	No

B. Applications

To show a variety of applications running on direct store, we use 4 standard benchmark suites and 4 other individual benchmarks; as shown in Table II. The table lists each benchmark code name, the sizes of small and big inputs, benchmark suite name, and whether the benchmark uses the GPU's shared memory feature or not. Some of the benchmarks are provided in the gem5-gpu project [18], namely Rodinia [21], Parboil [22] and Pannotia [23]. In addition to the four suites, four other benchmarks are also used: bitonic sort from [24], matrix multiplication and transpose from [25], and Cholesky

decomposition from [26]. All benchmarks are used to compare the performance of direct store and CCSM.

Our code translator assumes that the input program performs no CUDA memory copy. For Rodinia and Parboil in [18], benchmark codes have specifically excluded memory copy from CPU to GPU and vice versa. For other benchmarks, this has to be done manually, by converting the code to eliminate all CUDA memory copy functions and deleting all device variables, because in integrated CPU-GPU systems the GPU can easily access CPU data without the need to explicitly copy it. After all source codes contain no CUDA memory copy, they are then processed by the automatic translator to produce modified source codes ready to be compiled in the usual way.

C. Speedup

We evaluate the effect of direct store by running all benchmarks with two different types of memory accesses: cache-coherent shared memory (CCSM) and the proposed direct store memory access. We report the speedup obtained by the direct store approach by normalizing its total ticks to the total ticks of CCSM.

Figure 4 shows the speedup produced by the proposed scheme for small (top) and big (bottom) inputs, ignoring those benchmarks with zero percent speedup for both small and big inputs, namely GA, KM, LV, PT, SR, ST, and MS. For these benchmarks the proposed scheme does not bring benefit but also does not hurt the application. For small inputs, of the 22 benchmarks, 5 show a speedup of over 10%, 7 show no speedup, while 10 show modest speedup. NN, BL, VA, MM, and MT are the benchmarks with a speedup above 10%. Almost all of these benchmarks also have large miss rate reduction as shown in Figure 5, except for MM and MT, which instead experience miss rate increase. Moreover, these benchmarks do not use GPU shared memory, and therefore, they extensively make use of the GPU's cache. BP, HT, and GC have large miss rate reduction but their speedup is not high, because they use the shared memory feature and therefore do not involve the GPU L2 cache much in their computation. The rest are benchmarks with small miss rate reduction followed by small speedup or zero speedup. Note that the proposed approach never decreases performance, and can often provide performance increases.

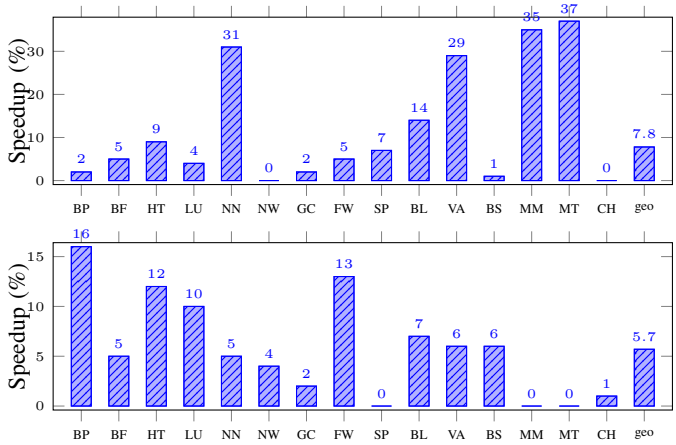

Fig. 4. Direct store speedup over CCSM for small (top) and big (bottom) inputs.

Figure 4 (bottom) shows that for BP, HT, LU, NW and FW, when input is big, although the benchmarks use shared memory, the massively parallel threads are unable to hide memory latency, and thus direct store impacts the performance significantly. For NN, BL, VA, MM, and TP, the speedup for big inputs is smaller compared to that for small inputs. This is because the input is larger than the size of the GPU L2 cache, and hence the miss rate reduction decreases followed by the speedup. The extreme condition happens for MM and MT, where the speedup drops to zero followed by a decrease in miss rate reduction. The rightmost bars on both figures represent the geometric means of all non-zero speedups, namely 7.8% speedup for small and 5.7% speedup for big inputs.

In summary, direct store does improve performance over CCSM. The mean performance improvements are moderate, but are achieved with no direct intervention from the programmer other than running our code translator on an existing program. Furthermore, while we might expect that the performance gains disappear for very large inputs, direct store programs still retain a performance advantage over CCSM in this case. Finally, we see that converting programs to use direct store never hurts performance compared to CCSM. Given these observations, we conclude that direct store can be a viable replacement for CCSM that requires little programming effort, never hurts performance, and often provides performance gains.

D. GPU L2 Cache Miss Rate Reduction

We now attempt to breakdown the performance gains of direct store (DS) over CCSM. Specifically, direct store 's performance gains should come from more favorable cache behavior compared to CCSM. We therefore begin by investigating the GPU L2 miss rate under both direct store and CCSM.

Figure 5 shows the GPU L2 cache miss rate for all benchmarks, ignoring those benchmarks with zero L2 cache miss rate both in CCSM and direct store, namely GA, LU, and BS. The red bar is the miss rate for CCSM, while the blue bar is for direct store.

Figure 5 (top) depicts the GPU L2 cache miss rate for small inputs. We can observe that in most cases the GPU L2 cache miss rate in CCSM is higher than in direct store. Benchmarks whose miss rate gets reduced are BP, BF, HT, KM, LU, NN, NW, SR, GC, FW, MS, SP, BL, VA, and CH. This is as we expect; since data is directed to the GPU L2 cache when the CPU stores, GPU does not need to fetch it again and this will reduce the miss rate.

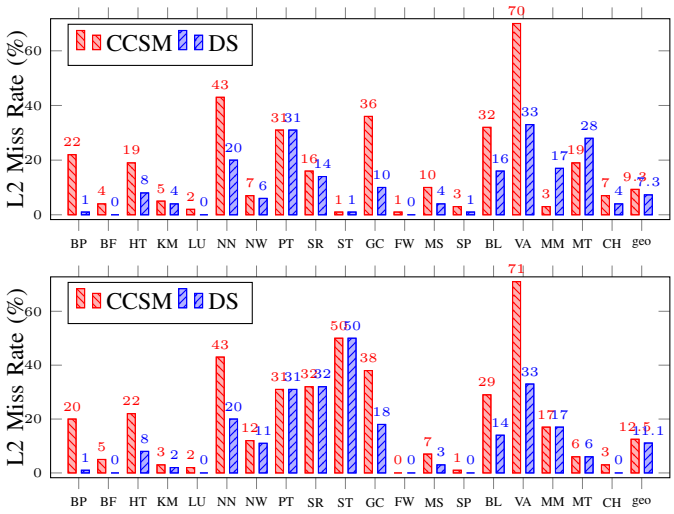

Fig. 5. GPU L2 miss rate for (top) small and (bottom) big inputs.

Nevertheless, some cases need further explanation. For MM and MT, the miss rate in direct store is higher than in CCSM. For both benchmarks, the total misses in CCSM and direct store actually

decrease. However, the GPU L2 accesses are also reduced and this reduction is even larger than the reduction in total misses. The rest of the benchmarks (GA, LV, PT, ST, and BS) produce no difference in miss rate between CCSM and direct store. For GA, LV, ST, and BS, the total misses decrease significantly; however, since the total cache accesses to the GPU L2 cache are extremely large compared to the total misses, the miss rate remains the same. For PT, there is no significant change in GPU L2 cache miss rate when direct store is implemented. The total misses and the total cache accesses to GPU L2 cache also do not change. This is due to the fact that in this benchmark the CPU does not store any data that will later be used by GPU.

Figure 5 (bottom) depicts the GPU L2 cache miss rate for big inputs. The miss rate reduction from CCSM to direct store is similar to the small input cases. The miss rate is reduced for BP, BF, HT, KM, LU, NN, NW, GC, MS, SP, BL, VA, and CH, while the rest shows no difference. For SR and FW, when input size is big, the reduction is canceled out due to the fact that cache accesses actually increase because the input is larger, while the total miss reduction is relatively small compared to the increase in cache accesses. Note again that the rightmost bars on both figures represent the geometric means of all L2 miss rate, namely 9.3% for CCSM compared to 7.3% for direct store for small inputs and 12.5% for CCSM compared to 11.1% for direct store for big inputs.

This data confirms that the proposed approach is having a positive impact on cache behavior. Specifically, under the direct store approach we hope that data resides in the GPU L2 cache on the first access, rather than being pulled to the GPU after a first miss.

E. Hardware Overhead

To apply direct store in integrated CPU-GPU systems, a small hardware overhead is incurred (all performance overhead is included in the above numbers). We add a network that directly connects the CPU's L1 cache and GPU L2 cache and a logic in the TLB to detect the incoming remotely stored data (Section III). The logic works by comparing store instructions' high-order addresses to the baseline address. This small overhead can be done by wiring to a logic gate.

V. Conclusion

We have proposed direct store, a new memory model for integrated CPU-GPU systems. In direct store, when a CPU stores data that will be consumed by the GPU, the memory system automatically deposits the data in the GPU LLC.

Our experimental results show that direct store reduces the GPU LLC miss rate and increases performance at the cost of a small hardware overhead for adding interconnection network and a new TLB logic. Additionally, direct store is much simpler than CCSM and needs less hardware support.

In summary, our proposed mechanisms can complement existing cache coherence mechanisms and provide a modest boost to performance in many cases, or they could be used as a simpler replacement for CPU to GPU communication.

Acknowledgments

Henry Hoffmann's effort is supported by the NSF (CCF-1439156, CNS-1526304, CCF-1823032, CNS-1764039) and a DOE Early Career award. Reza Pulungan is supported by hibah penelitian dosen DIKE Universitas Gadjah Mada dana BPPTNBH tahun 2019. Ardhi Yudha and Yan Solihin are supported in part by NSF grant 1914717 and 1908079, and by UCF.

References

[1] H. Wang, R. Singh, M. J. Schulte, and N. S. Kim, "Memory scheduling towards high-throughput cooperative heterogeneous computing," in *PACT*. ACM, 2014.

[2] 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 *MICRO*. ACM, 2013.

[3] H. Wang, V. Sathish, R. Singh, M. J. Schulte, and N. S. Kim, "Workload and power budget partitioning for single-chip heterogeneous processors," in *PACT*. ACM, 2012.

[4] J. Y. Jang, H. Wang, E. Kwon, J. W. Lee, and N. S. Kim, "Workload-aware optimal power allocation on single-chip heterogeneous processors," *TPDS*, 2016.

[5] E. S. Chung, P. A. Milder, J. C. Hoe, and K. Mai, "Single-chip heterogeneous computing: Does the future include custom logic, FPGAs, and GPGPUs?" in *MICRO*. IEEE, 2010.

[6] R. B. Gillett, "Memory channel network for pci," *IEEE Micro*, Feb 1996.

[7] H. Hoffmann, D. Wentzlaff, and A. Agarwal, "Remote store programming," in *HiPEAC*, Y. N. Patt, P. Foglia, E. Duesterwald, P. Faraboschi, and X. Martorell, Eds., 2010.

[8] G. Kurian, O. Khan, and S. Devadas, "The locality-aware adaptive cache coherence protocol," in *ISCA*, 2013.

[9] Y. Solihin, *Fundamentals of Parallel Multicore Architecture*, 1st ed. Chapman Hall/CRC, 2015.

[10] Y. Yang, P. Xiang, M. Mantor, and H. Zhou, "CPU-assisted GPGPU on fused CPU-GPU architectures," in *HPCA*, 2012.

[11] A. Sethia, G. Dasika, M. Samadi, and S. Mahlke, "APOGEE: Adaptive prefetching on GPUs for energy efficiency," in *PACT*, 2013.

[12] J. Lee, N. B. Lakshminarayana, H. Kim, and R. Vuduc, "Many-thread aware prefetching mechanisms for GPGPU applications," in *MICRO*, 2010.

[13] N. Agarwal, D. Nellans, E. Ebrahimi, T. F. Wenisch, J. Danskin, and S. W. Keckler, "Selective GPU caches to eliminate CPU-GPU HW cache coherence," in *HPCA*, 2016.

[14] D. Koufaty, X. Chen, D. K. Poulsen, and J. Torrellas, "Data forwarding in scalable shared-memory multiprocessors," in *Proceedings of the 9th international conference on Supercomputing, ICS 1995, Barcelona, Spain, July 3-7, 1995*, 1995.

[15] C.-L. Yang, A. R. Lebeck, H.-W. Tseng, and C.-H. Lee, "Tolerating memory latency through push prefetching for pointer-intensive applications," *TACO*, 2004.

[16] C. N. Keltcher, K. J. McGrath, A. Ahmed, and P. Conway, "The AMD Opteron processor for multiprocessor servers," *IEEE Micro*, 2003.

[17] "The gem5 simulator documentation: MOESI hammer," 2013. [Online]. Available: http://www.m5sim.org/MOESI_hammer

[18] J. Power, J. Hestness, M. S. Orr, M. D. Hill, and D. A. Wood, "gem5-gpu: A heterogeneous CPU-GPU simulator," *CAL*, 2015.

[19] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, 2011.

[20] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *ISPASS*, 2009.

[21] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC*, 2009.

[22] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W. mei W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," University of Illinois at Urbana-Champaign, Tech. Rep., 2012.

[23] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron, "Pannotia: Understanding irregular GPGPU graph applications," in *IISWC*, 2013.

[24] M. Endler, "Parallel bitonic sort using CUDA," 2011. [Online]. Available: https://gist.github.com/mre/1392067

[25] Z. Liu, "Matrix multiplication in CUDA," 2018. [Online]. Available: https://github.com/lzhengchun/matrix-cuda

[26] M. Bhatt, "Parallel implementation of Cholesky decomposition using CUDA APIs," 2017. [Online]. Available: https://github.com/bhattmansi/Implementation-of-Cholesky-Decomposition-in-GPU-using-CUDA