Turkish Journal of Electrical Engineering and Computer Sciences

Volume 30 | Number 5 Article 6

1-1-2022

Using a static naming approach to implement remote scope promotion

AYŞE YILMAZER

Turkish Journal of Electrical Engineering & Computer Sciences

http://journals.tubitak.gov.tr/elektrik/

Research Article

Turk J Elec Eng & Comp Sci
(2022) 30: 1758 – 1772

doi:10.55730/1300-0632.3903

Using a static naming approach to implement remote scope promotion

Ayşe YILMAZER-METİN*
Department of Computer Engineering, İstanbul Technical University, İstanbul, Turkey

Received: 27.09.2021 • **Accepted/Published Online:** 02.06.2022 • **Final Version:** 22.07.2022

Abstract: GPUs employ simple coherence mechanisms and require explicit use of costly synchronization operations for data integrity. *Local-scoped* synchronization can be utilized to lower the performance penalty of synchronization when sharing is within a subgroup of threads. Unfortunately, in asymmetric sharing (which is an important dynamic sharing pattern), it is necessary to use *global-scoped* synchronization due to possible accesses by remote sharers. *Remote Scope Promotion (RSP)* was introduced to take advantage of local-scoped synchronization at regular accesses while using scope promotion at occasional remote accesses. First implementation of RSP makes use of a simple approach that performs costly cache operations on **all** L1 data caches when implementing scope promotion, and therefore, it performs poorly on large scale GPU systems. We present *nRSP* which utilizes a static *naming* mechanism to identify regularly accessing agent in asymmetric sharing and avoids applying costly coherence actions on every L1 data cache when implementing scope promotion. We evaluate nRSP using timing detailed Gem5-APU simulator modeling a GPU system with 128 Compute Units and show that nRSP lowers remote synchronization overhead greatly and improves performance considerably. On average, nRSP provides around 28% speedup on a 128 Compute Unit GPU device.

Key words: Asymmetric synchronization, GPUs, remote scope promotion, work-stealing

1. Introduction

GPU devices lack sufficient support for efficient synchronization operations. They favor simple cache coherence protocols. Data integrity is maintained with heavyweight synchronization operations. As a result, many general-purpose GPU (GPGPU) applications with fine-grained synchronization suffer from significant performance overhead. *Scoped synchronization* [1] was proposed to alleviate the synchronization overhead. For example; synchronization overhead can be greatly reduced when the synchronization within a subgroup of agents using *local-scoped* synchronization. Unfortunately, scoped synchronization is only helpful when the participating agents are known statically.

An important dynamic sharing pattern exists in work-stealing [2] and cannot benefit from scoped synchronization. In work-stealing, each thread owns a task queue. When a thread is out of task, it attempts to steal task from another thread's task queue. Due to possible accesses from remote-agents (i.e. the stealing threads), all threads must synchronize specifying global-scope when accessing to any task queue (either local or a remote task queue). This dynamic sharing model is called *asymmetric sharing*.

[1]Despite the most accesses are local, due to possible rare accesses from remote-agents, lightweight local-scoped synchronization cannot be used in asymmetric sharing.

*Correspondence: yilmazerayse@itu.edu.tr
[1]In the perspective of a task queue, we will refer to the thread that owns the queue and performs regular accesses as *local-owner*. Other occasionally accessing threads (performing the stealing) will be referred as *remote-agents*.

Recently, Orr et al. [3] introduced *remote scope promotion* (RSP) to provide support for asymmetric sharing on GPUs. In RSP, remote-agents must synchronize using global-scope for their occasional accesses, while local-owner can utilize local-scoped synchronization. RSP performs *scope promotion* from *local* to *global* scope at the time of *remote synchronization*. This enables retaining the consistency of shared data. Unfortunately, the initial implementation of RSP [3] utilizes a *broadcast* approach that performs flushing and invalidating every L1 data cache to perform promotion of synchronizing local-scoped synchronization operations. Flushing and invalidating local caches excessively increases the synchronization overhead greatly, disrupts the locality in local caches, and diminishes the scalability.

We introduce a novel mechanism, *nRSP*, to implement RSP's remote synchronization operations. nRSP uses a static *naming* mechanism to locate the local-owner in asymmetric sharing, avoids flushing and invalidating every L1 data cache, and executes the costly cache operations only on the ***participating*** local caches when performing scope promotion. Use of a static naming approach reduces the complexity that is added to cache coherence protocol and avoids any hardware area overhead. We evaluate our design using timing detailed Gem5-APU[2] simulator modeling 64 and 128 Compute Unit GPU systems and show that nRSP can scale on big GPU devices.

2. Related work

There are several studies focusing on GPU coherence. TC-Weak [4] presented a cache coherence protocol which used physical time-stamp. Both RCC [5] and G-TSC [6] presented cache coherence protocols that were based on logical time-stamp. GPU DeNovo [7] was presented as a coherence protocol that was extended DeNovo [8] for GPUs. While it is feasible to provide support for asymmetric sharing with coherent local caches in GPU memory system, any of these studies do not benefit from scopes. Besides cache coherence on GPUs, there are studies focusing on GPU transactional memory (TM) [9–14] and GPU synchronization [15, 16]. Any of TM studies did not acknowledge scopes. While Hardware TMs bring increase in hardware cost, Software TMs suffer from high performance penalties. HQL introduced hierarchically managed hardware locks. [15] and [16] proposed software based locking mechanisms for efficient interthread communication. Scopes and asymmetric synchronization are not considered in any of these GPU synchronization work.

Simple and Fast Biased Locks [17] and Quickly Reacquirable Locks [18] are two studies focusing on asymmetric synchronization on CPUs. Unfortunately these two work are built around complex synchronization support that CPUs offer. GPU caches lack sufficient synchronization support. It is not possible to expand these work onto GPUs without great challenges.

There are three studies that are most related to our work. RSP [3] work presented RSP semantics and first implementation of them. First implementation of RSP semantics [3] has shown to not scale for large scale GPUs [19, 20]. Later, sRSP [20] presented a new implementation for RSP's remote synchronization operations with a dynamic naming approach that applies cache-flush and cache-invalidation operations selectively and avoids flushing and invalidating all local caches. With selective cache-flush and cache-invalidation operations, sRSP scales better. However, sRSP makes use of a hardware structure to track local release operations for dynamic naming implementation which increases area overhead and the complexity of cache coherence protocol. While providing around the same amount of performance benefits compared to sRSP, nRSP do not cause any area overhead with lower protocol complexity by making use of a static naming approach.

[2]AMD. (2015). The AMD gem5 APU simulator: modeling heterogeneous systems in gem5 [online]. Website http://www.m5sim.org/wiki/images/f/fd/AMD_gem5_APU_simulator_micro_2015_final.pptx [accessed 00 September 2021].

hLRC [19] does not implement scopes and RSP semantics but provides support for asymmetric synchronization. It requires registration of synchronization variables and dynamically tracks ownership of them. Similar to sRSP, hLRC's approach is dynamic. It has several shortcomings including increased protocol complexity and being prone to deadlocks and hot spots.

nRSP follows a static approach for identifying affinity of synchronization variables to provide an efficient and scalable implementation of RSP semantics.

3. GPU programming and execution model

In GPGPU programming, programmers divide an application into coarse-grained tasks to map onto CPU cores and GPU devices. Tasks that are mapped onto CPU cores are called *host threads* and the ones that are mapped onto GPU devices are called *kernels*. For each GPU kernel, fine-grained data parallelism must be extracted and expressed explicitly. Multiple instances of a kernel run in parallel as GPU *workitems*. GPGPU programming models provide an abstraction to hierarchically organize CPU and GPU threads. In this organization, all threads from CPU and GPU form a *system* group. GPU threads (i.e. workitems) executing on the same GPU make up a *component* group and are further batched into *workgroups*.[3,4]

A GPU device accommodates a set of Compute Units (CUs). A CU contains a set of processing elements forming a Single Instruction Multiple Data (SIMD) unit. GPU threads from the same workgroup must run on the same CU. During the execution of a kernel, GPU threads on a Compute Unit are batched into *wavefronts* to run in synchronization on SIMD units. Each CU is equipped with a private L1 data cache. A common L2 cache is shared by all CUs on a GPU device.

4. Memory consistency on GPUs

Weak memory models are preferred on GPUs. OpenCL defines a consistency model which is based on *acquire and release* memory ordering constraints.[5] In this model, threads synchronize with acquire-release orderings to preserve consistency of shared data. Typically, ordering constraint for a load operation is acquire and for a store operation it is release. OpenCL extends these synchronization operations with *scopes*.

Acquire and release ordering semantics require hardware and software support. Ordering of memory operations with respect to atomic operations with acquire-release orderings must be maintained.[6] When two threads synchronize using acquire and release orderings, hardware has to provide that most recent data (i.e. the data supplied prior to the synchronizing atomic operation with release ordering) has to be delivered for a read operation (i.e the read operation that is performed after the synchronizing atomic operation with acquire ordering).

In this model, synchronization overhead could become very substantial with thousands of threads. *Scoped* synchronization with acquire-release orderings are presented [1, 21] to allow narrowing the scope of synchronization to a subgroup of agents. OpenCL specifies following synchronization scopes: *workitem (wi)*,

[3]Our work makes use of OpenCL programming model and we utilize OpenCL terminology in this paper.

[4]The Khronos Group Inc. (2018). Khronos OpenCL registry [online]. Website https://www.khronos.org/registry/OpenCL/. [accessed 00 January 2021].

[5]International organization for standardization. (2017). Working draft, Standard for programming language [online]. Website http://www.open-std.org/jtc1/sc22/ wg21/docs/papers/2017/n4713.pdf [accessed 00 January 2021].

[6](1) Any memory operations following an atomic operation with *acquire* ordering must not execute prior to that atomic operation with acquire ordering. (2) Any memory operations before an atomic operation with *release* ordering must not execute after that atomic operation with release ordering.

wavefront (wv), *workgroup (wg)*, *component (cmp)*, and *system (sys)* scopes. Each of these scopes corresponds to a subgroup in hierarchical GPU execution model. In this paper, wg-scope is referred as *local*, cmp-scope is referred as *global*, and sys-scope is referred as *system level* scopes.[7] Besides limiting the extent of synchronization to a subgroup of agents, scopes also enable exploiting locality in a hierarchical memory design.

4.1. GPU memory system and support for scoped acquire-release synchronization

Memory bandwidth plays a critical role in GPU performance and unfortunately, any of the mechanisms that are applied on CPUs are not scalable on GPUs.

Therefore, GPUs favor simple cache coherence mechanisms. Typically, a simple caching mechanism with write-combining or write-through is preferred. Write-combining marks modified bytes for each cache line using a set of bit-flags. With write-combining, multiple writes could be combined before a cache line is evicted or flushed and only the modified bytes are written back.

While leveraging simple caching mechanisms, GPUs require explicit use of synchronization operations (i.e. atomic operations with scoped acquire-release orderings) for data consistency. It becomes quite expensive when the larger scopes are used. There are two main reasons for this: (1) a lot more threads are affected from the synchronization and (2) the *synchronization level* for a larger scope is at the lower levels of memory system and costly cache operations must be performed on the more levels of the memory system.

Synchronization level in memory system for each scope varies. Depending on the synchronization level, the implementation of a scoped synchronization differs. On a typical CPU+GPU heterogeneous system, the *synchronization levels* are L1 cache, L2 cache and L3 cache/system memory for local, global and system-level scopes, respectively. With a scoped atomic operation with release ordering, all locally performed modifications must be propagated down to the selected scope's synchronization level. For example, there is no need to propagate any local modifications with a local release ordering. On the other hand, all local updates must be propagated down to L2 cache from local caches for a global-scoped release ordering. Similarly, for a system-level release ordering, all local updates from local and L2 caches must be propagated down to L3 cache/system memory. Scoped atomic operations with acquire orderings are executed to guarantee that all shared data is obtained from the selected scope's synchronization level. It requires invalidating locally stored likely stale data. Naively, a cache-invalidation operation invalidates all cache lines in a cache. For workgroup scoped atomic operation with acquire ordering, there is no need for any invalidation. On the other hand, local caches must be invalidated for a global scoped atomic operation with acquire ordering. Correspondingly, all data from L1 data and L2 caches are expected to be invalidated for a system-level atomic operation with acquire ordering.[8]

Caches in GPU memory hierarchy are furnished with *cache-flush* operations for writing back all modified cache lines at once. A FIFO structure —called synchronization-FIFO (sFIFO) [22], can be used to track all modified cache lines on GPU caches. When updating a cache block, the block address is recorded into sFIFO. When flushing a cache block, block addresses are removed inorder from sFIFO and each associated cache block is flushed down to the next level in GPU memory hierarchy. To perform a scoped synchronization operation with acquire ordering, first, all local modifications are propagated down using sFIFO, and next, a one cycle cache-invalidation can be realized on the cache.

[7] Since our work targets memory system, we only focus on synchronization operations with local, global and system level scopes.
[8] Atomic operations themselves are performed at the selected scope's synchronization level.

5. Asymmetric sharing and remote-scope promotion

Use of *local-scope* limits the synchronization overhead since there is no need to flush or invalidate any local caches. Unfortunately, using local-scope with synchronizations is not viable for asymmetric sharing and costly global-scope must be used to contain all possible agents. *RSP [3]* allows guarding local-owner's frequent accesses to the shared data with local-scoped synchronization and puts off the costly synchronization operations to the rare remote accesses. During remote accesses, previous and subsequent local synchronizations are promoted to global-scope (typically with remote initiated cache-flush and cache-invalidation operations). Three new synchronization primitives are introduced to implement RSP semantics:

Remote-acquire: Triggers two suboperations for remote agent: (1) promotion of last local synchronization operation with release ordering to global-scope; (2) a synchronization with acquire ordering at global scope. Promotion of local synchronization with release ordering facilitates sending down all prior modifications to global scope's synchronization level. Acquire ordering on remote-agent's cache is performed for obtaining the most recent data from global scope's synchronization level.

Remote-release: Triggers two suboperations for remote agent: (1) a synchronization operation with release ordering at global scope; (2) promotion of subsequent local synchronization with acquire ordering to global scope. Synchronization operation with release ordering facilitates sending down all updates (remote-agent's local modifications) to global scope. Promoting the next local synchronization with acquire ordering guarantees that the local-owner gathers the most recent data from global scope's synchronization level.

Remote-acquire+release: Provides the remote-agent with two suboperations: (1) promotion of the most recent local synchronization with acquire and/or release ordering(s) (that was implemented by local-owner); (2) a global synchronization operation with acquire+release ordering.

First implementation of RSP extends basic GPU cache protocol to carry out RSP's remote synchronization operations. A ***broadcast*** mechanism is used to implement remote scope promotion. When performing remote scope promotion, expensive cache-flush and/or cache-invalidation operations are initiated by remote agent on ***all*** local data caches with a broadcast message. *Unfortunately, flushing and invalidating local caches heavily limits the scalability of RSP's first implementation [19, 20].*

6. nRSP: a static naming approach to implement RSP semantics

In this paper, we introduce a novel technique to realize RSP's remote synchronization operations. This technique is based on *statically* determining local-owner's cache affinity (i.e. *naming* the local-owner's cache). Using this information, we flush and/or invalidate only the local-owner's cache. Using *Named*-flush and *Named*-invalidation operations, we eliminate flushing and invalidating local caches unnecessarily and minimize remote synchronization overhead.

nRSP proposes modifications to programming model and run-time environment. The modifications in programming model and run-time environment facilitate naming the local-owner's cache using *workgroup-to-CU mapping*. It also modifies L1 data and L2 cache cache protocols to realize remote synchronization operations using *named*-flush and *named*-invalidation operations. nRSP provides support for RSP's all three remote synchronization operations. Below we explain our static naming mechanism, proposed programming approach, and changes to the baseline GPU cache protocol.

6.1. Proposed static naming mechanism

Our static naming approach makes use of following two inputs: (1) local-owner's workgroup-id, and (2) selected *workgroup scheduling model*. We propose to extend the GPU run-time to support and expose a predefined set of *workgroup scheduling* (workgroup-CU mapping/assignment)[9] models. Then, our work proposes to extend the programming environment to allow programmers to choose from one of the supported workgroup scheduling models at kernel launch time.

In Figure 1, we show the steps to get local-owner's cache-affinity and to use this information when performing remote-scope promotion.

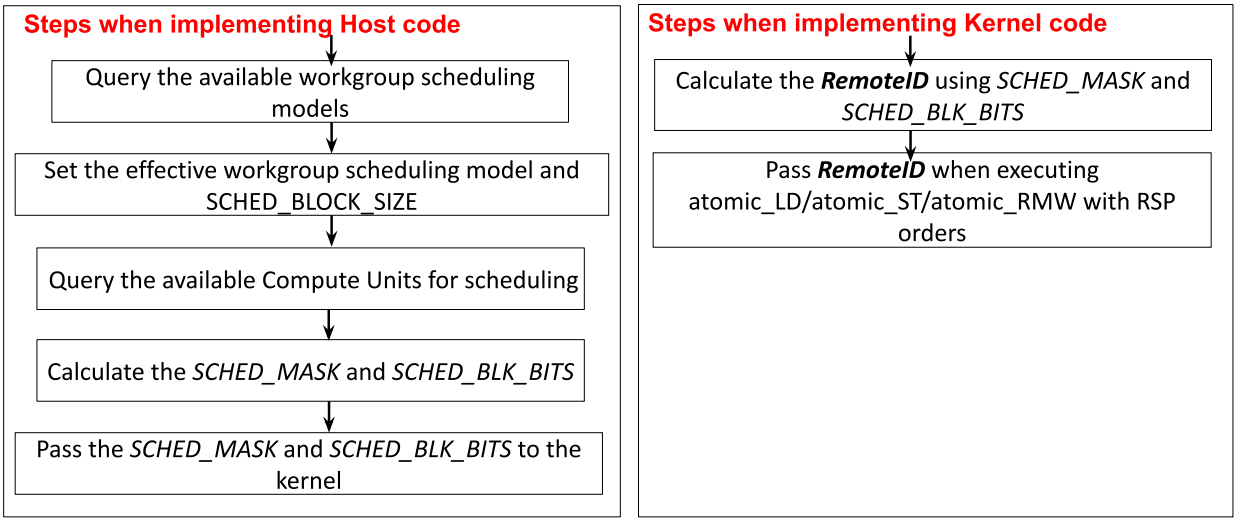

Figure 1. Steps for calculating local-owner's cache-affinity (left) and using the calculated affinity information when performing remote-scope promotion (right).

6.1.1. Basic workgroup scheduling models

In OpenCL programming model, workgroups are created as 1, 2, or 3 dimensional arrays. So, we base our workgroup scheduling models on array distribution schemes. We extend OpenCL runtime to support at least three basic scheduling models. Programmers are allowed to query the supported models and select one of them. We believe that this would be a very beneficial extension for locality management. These models are explained below.

`Block mapping:` In this model, first, the number of workgroups that could be assigned to a CU is calculated based on the amount of available resources. After that, all available workgroups are consecutively distributed into *scheduling-blocks*.[10] Next, each scheduling-block is sequentially mapped onto a CU. The number of workgroups in a *scheduling block* is referred as *SCHED_BLOCK_SIZE*. Scheduling blocks could be formed as 1, 2, or 3 dimensional. Since the workgroup scheduler first does a conversion to 1 dimension when the workgroups are created with 2 or 3 dimensions, our mechanism would work for 2 or 3 dimensional workgroups without requiring any change. This model will be exposed to programmers as `BLOCK_MAPPING` by programming environment. An example of this model is illustrated in Figure 2a.

[9] In this paper, *workgroup-CU mapping* and *workgroup scheduling* terms are interchangeably used.
[10] Scheduling-block is a group of workgroups that is assigned to a CU.

Block-cyclic mapping: Similar to Block mapping, this model first partitions workgroups into scheduling blocks. However, this time, the block size is chosen such that more than one block can be accommodated by a CU. So, the number of scheduling blocks will be larger than the number of CUs. Scheduling blocks will be assigned to the CUs in a round-robin manner. An example of this model is illustrated in Figure 2b. This model will be exposed to programmers as `BLOCK_CYCLIC_MAPPING` by programming environment.

Block-random mapping: Block-random mapping also starts with partitioning workgroups into scheduling blocks. Then, each scheduling block is *randomly* assigned to a Compute Unit. The mapping between scheduling blocks and CUs can be guided by the communication pattern between blocks. This model is illustrated in Figure 2c. This model will be exposed to programmers as `BLOCK_RANDOM_MAPPING` by programming environment.

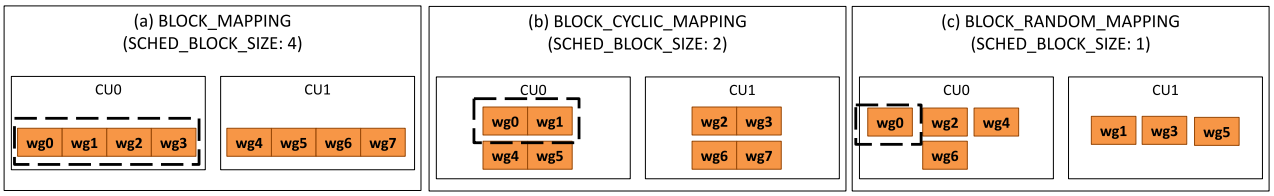

Figure 2. Three basic scheduling models: (a) Block mapping, (b) Block-cyclic mapping, and (c) Block-random mapping. The blocks within the dashed frame represents a *Scheduling-block*.

Persistent threading is the preferred approach when implementing work-stealing for dynamic load balancing. Therefore, all workgroups will be assigned to a CU at kernel launch time. For persistent threading, *Block* mapping or *Block-cyclic* mapping can be used. On the other hand, it is possible to have applications that do not use persistent threading and they might have too many workgroups to be scheduled at kernel launch time. In such a case, *Block-random* mapping can be used and run-time can utilize a table to hold workgroup-CU mappings. Later, this table can be used for naming. Block-random mapping offers a more flexible scheduling option.

Scheduling models can be exposed to programmers using an enumerated type shown in Figure 3. We propose extending OpenCL run-time to offer functions for querying and selecting scheduling capabilities/attributes of the GPU device. To support static naming, we propose programming environment to support functions shown in Figure 3. We also propose such a construct (i.e. SCHED_TBL_PTR) as shown in Figure 3 to be provided by run-time for supporting Block-random mapping model.

```
enum SCHED_MODELS {
    BLOCK_MAPPING=0,
    BLOCK_CYCLIC_MAPPING=1,
    BLOCK_RANDOM_MAPPING=2
};
typedef struct {
   ...
} SCHED_TBL_STRUCT, *SCHED_TBL_PTR;

int GET_NUM_COMPUTE_UNITS(); // returns the number of Compute Units
SCHED_MODELS* GET_SUPPORTED_MODELS(); // returns the supported scheduling models
int SET_SCHED_MODEL(SCHED_MODEL sched_model);
int GET_SCHED_MAX_BLOCK_SIZE(); // returns the number of work-groups in a scheduling block
int SET_SCHED_BLOCK_SIZE(int blk_sz); // set the block size for selected scheduling type
                                      // blk_sz cannot be greater than result of GET_SCHED_MAX_BLOCK_SIZE()
int CREATE_SCHED_TABLE(SCHED_TBL_PTR tbl_ptr, int num_sched_blocks); // create the scheduling lookup table
                                                                     // for BLOCK_RANDOM_MAPPING
```

Figure 3. Type and function definitions for workgroup scheduling models.

6.2. Programming approach for naming

We explain necessary code modifications for obtaining local-owner's cache-affinity with an example code fragment. Our example uses *Block-cyclic* mapping and the `SCHED_BLOCK_SIZE` is selected as 2. Our example assumes that there are 2 Compute Units on GPU device.

Figure 4 shows our sample code fragment. First, supported workgroup scheduling models are queried as shown in the figure. After checking if the Block-cyclic mapping is available, it is set as the effective workgroup scheduling model in our application's host code.[11] Next, possible maximum scheduling block size is queried and scheduling block size is set to 2. After that, *SCHED_MASK* and *SCHED_BLOCK_BITS* are calculated using number of CUs and *SCHED_BLOCK_SIZE*. These two parameters will be used later in kernel code for local-owner's affinity calculation. So, these two parameters are set as two *kernel* arguments.

Host code

```
int main(...) {
  int sched_blk_sz = 2;
  ...
  SCHEDULING_TYPE* sched_t;
  sched_t = GET_SUPPORTED_SCHEDULINGS();
  ...
  // make sure BLOCK_CYCLIC_SCHEDULING supported
  SET SCHEDULING_TYPE(BLOCK_CYCLIC_SCHEDULING);
  ...
  int max_sched_blk_sz = GET_SCHEDULING_MAX_BLOCK_SIZE();
  ...
  //make sure (sched_blk_sz <= max_sched_blk_sz)
  ...
  SET_SCHEDULING_BLOCK_SIZE(sched_blk_sz);
  // get number of CUs
  int num_cus = GET_NUM_COMPUTE_UNITS();
  // calcualte number of bits for SCHED_BLOCK_SIZE
  int sched_blk_bits = log2(sched_blk_sz);
  // calculate sched_mask
  int sched_mask = log2(num_cus) << sched_blk_bits;
  ...
  ...
  // set kernel arguments to pass SCHED_BLOCK_BITS and SCHED_MASK
  clSetKernelArg(my_kernel, 0, sizeof(int), (void*)&(sched_mask));
  clSetKernelArg(my_kernel, 1, sizeof(int), (void*)&(sched_blk_bits));
  // more param settings
  ...
  // launch kernel here
  ...
```

Kernel code

```
__kernel my_kernel(const int sched_mask,
                   const int sched_blk_bits,
                   // more params
                   )
{
   ...
   ...
   // calculate remote_id using
   // local modifiers id (remote_wg_id)
   int remote_id = (remote_wg_id & sched_mask) << sched_blk_bits;

   ...
   ... = atomic_compare_exchange_strong_explicit(
    ...,memory_order_remote_acquire, memory_scope_device, remote_id);

   ...
   ...

}
```

Figure 4. Example code fragment illustrating code modifications for obtaining and using local-owner's cache affinity.

In the kernel code, the SCHED_MASK, SCHED_BLOCK_BITS and local-owner's *Workgroup-Id* are used for calculating local-owner's cache affinity. Then the calculated affinity is passed as an argument to the remote-scope promotion operation. Our example in Figure 4 shows how to calculate local-modifier's affinity when *Block* mapping or *Block-cyclic* mapping is used. When *Block-random* mapping is used, the lookup table that holds the Workgroup-CU mappings will be used. In the host code, first, a table in *SCHED_TBL_STRUCT* type is created using *CREATE_SCHED_TABLE()* function. Next, this lookup table is passed as an argument to the kernel code. During the kernel launch time, this table is expected to be filled by the OpenCL runtime. Later, in the kernel code, the local-owner's affinity will be looked up using this lookup table and the local-owner's `Workgroup-id`.

6.3. Implementing RSP's remote synchronization operations with *Named*-flush and *Named*-invalidation operations

We modified L1 data and L2 cache protocols to carry out RSP's remote synchronization operations with *Named*-flush and *Named*-invalidation operations. A remote-identifier (identified name for local-owner's cache) must

[11] the application code that is running on cpu side is referred as *host code*.

be passed to cache controller when making a synchronization request with *remote-acquire*, *remote-release*, or *remote-acquire+release* orderings.

We explain our implementation of RSP's remote synchronization operations with a running example. This example depicts a critical section which is protected by a lock. The shared data in this critical section is owned by *wg0* (i.e. the local-owner). While this shared data is frequently accessed by the threads from wg0 and local to them, it is occasionally touched by remote agents (the threads from other workgroups, e.g. *wg1*). wg0 is scheduled to run on CU0. Threads from wg0 accesses to the shared data in this critical section using *local-scoped* synchronization operations with acquire and release orderings. Workgroup *wg1* is the remote-agent in our example and scheduled to run on CU1. Remote-agents use *global-scoped* synchronization operations with *remote-acquire* and *remote-release* orderings when accessing to the shared data owned by wg0. Figures 5 and 6 represent the accesses to this critical section by remote-agent using synchronization operations with remote-acquire and remote-release orderings.

6.3.1. Implementing local-scoped synchronizations operations in nRSP

For implementing a local-scoped synchronization with acquire and/or release ordering, nRSP does not require any changes in baseline GPU cache protocol. When the cache-controller receives an atomic operation with acquire, release, or acquire+release ordering, the atomic operation itself is implemented on local cache and the synchronization completes. There is no need to invalidate or flush the local cache for ordering constraints.

6.3.2. Implementing synchronization operations with remote-acquire ordering in nRSP

As illustrated in Figure 5, a GPU thread from wg1 executes `atomic_cas_rem_acq_cmp(L, 0 → 1, rem_id: 0)` instruction to enter the critical section. `$L1_1` sends a *Named*-flush request to local-owner's cache `$L1_0`. Upon receiving the *Named*-flush request, a cache-flush operation is started in `$L1_0` and all modified lines are flushed down using sFIFO. When the flushing `$L1_0` completes, then, line (`L`) is set as *Locked* and `$L1_1` is replied with a positive acknowledgment. Line(`L`) is set as Locked on local-owner's cache to prevent the local-owner from performing any local synchronization on this line. `$L1_1` is flushed and invalidated, while waiting acknowledgement from local-owner's cache (`$L1_0`). Next, `$L_1` forwards `atomic_cas` to L2 cache to finish the synchronization. When `atomic_cas` completes at `$L2` and the result is returned to `$L1_1`, a *Unlock-line(L)* message is sent to the `$L1_0`. In the end, the result of the `atomic_cas` is forwarded to the GPU thread executing this instruction.[12]

6.3.3. Implementing synchronization operations with remote-release ordering in nRSP

After successfully synchronizing using `atomic_cas` operation with remote-acquire ordering, the remote-agent from wg1 continues with critical section. After the critical section, the `atomic_st_rem_rel_cmp(L, 0)` instruction is run by the remote agent. This is illustrated in Figure 6. `Atomic_st` with remote-release ordering, first, triggers a cache-flush on `$L1_1`. After finishing the cache-flush on `$L1_1`, the `atomic_st` is performed on $L2. In the end, to enforce scope promotion for a subsequent local-scoped synchronization by local-owner, a *Named*-invalidation request is directed to the local-owner's cache `$L1_0`. Once receiving the *Named*-invalidation

[12]When implementing a synchronization operation with remote ordering, it is possible to have the local-owner and remote agent on the same Compute Unit. In such a case, scope promotion is not necessary. The cache controller in nRSP is modified to detect if the remote-agent and the local-owner shares the same L1 data cache and to avoid unnecessary remote scope promotion.

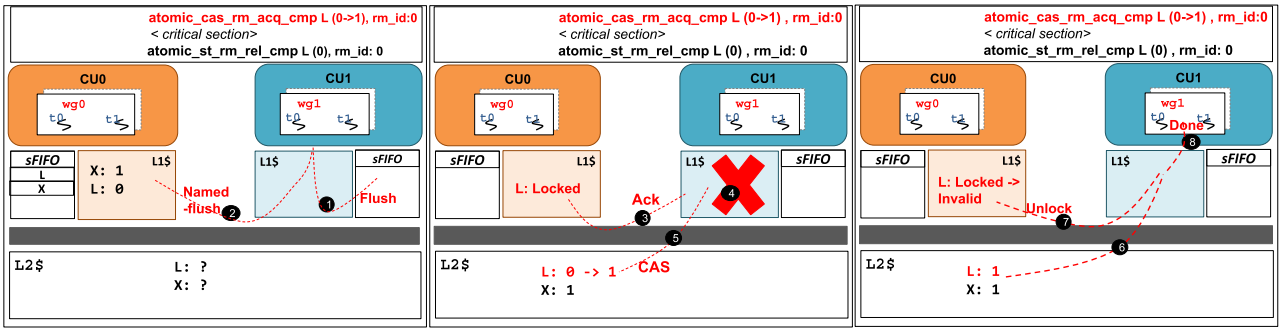

Figure 5. Illustrating steps in a global-scoped synchronization with remote-acquire ordering in nRSP. *wg1* on CU1 runs `atomic_cas_rem_acq_cmp(L, 0 → 1)` to touch to the data in critical section. Synchronizing *atomic_cas* operation with remote-acquire ordering is performed using remote-id *0*.

request with address (L), `$L1_0` flushes down all dirty lines and performs a single cycle invalidation with flash-invalidation.

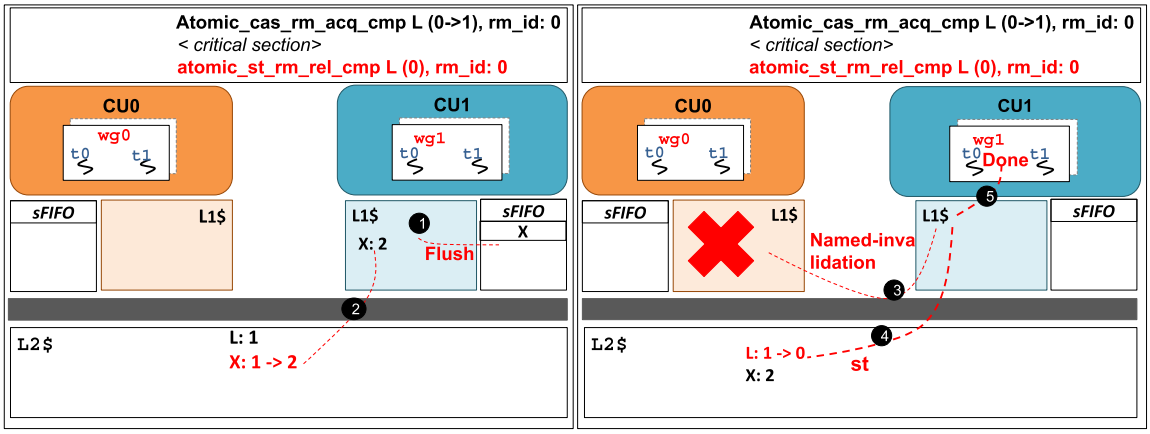

Figure 6. Illustrating steps in a global-scoped synchronization with remote-release ordering in nRSP. *wg1* runs `atomic_st_rem_rel_cmp(L, 0)` instruction following the critical section.

7. Evaluation

7.1. Methodology

For evaluation, we used Gem5 [23] APU simulator[13]. It supports emulation of OpenCL[14] run-time and execution of OpenCL kernels that are compiled into HSAIL[15] instructions. We simulate GPU devices that are configured using parameters listed in Table . On our simulated GPU devices, there exists a local L1 data cache for each CU. An L1 instruction cache is shared by 4 CUs. All CUs share a common L2 cache. At the lower level, L2 cache is connected to the System Memory. Depending on selected scope, a synchronization operation with

[13]AMD. (2015). The AMD gem5 APU simulator: modeling heterogeneous systems in gem5 [online]. Website http://www.m5sim.org/wiki/images/f/fd/AMD_gem5_APU_simulator_micro_2015_final.pptx [accessed 00 September 2021].

[14]The Khronos Group Inc. (2018). Khronos OpenCL registry [online]. https://www.khronos.org/registry/OpenCL/. [accessed 00 January 2021].

[15]HSA Foundation. (2018). HSA programmer's reference manual: HSAIL virtual ISA and programming model, compiler writer, and object format (BRIG) [online]. Website http://www.hsafoundation.com/standards/ [accessed 00 September 2021].

release ordering may trigger flushing local cache and acquire ordering may trigger 1 cycle cache-invalidation on local data and L2 caches. These caches are equipped with sFIFO structures to flush a cache efficiently. There are 16 sFIFO entries on every local cache and 24 sFIFO entries on shared L2 cache.

Table . Configuration parameters of simulated GPU devices.

`Number of Compute Units`	64/128
`SIMDs`	4 SIMDs per CU, 1GHz
`Wavefronts per Compute Unit`	40
`Data cache`	16kB (16-way) set-associative with 64B block size, 4 cycle access latency
`L2 Cache`	512kB (16-way) set-associative with 64B block size, 24 cycle access latency
`Instruction cache`	32kB (8-way) set-associative with 64B block size, 4 cycle access latency
`DRAM`	DDR3, 8 Channels, 500 MHz
`Protocol`	write-no-allocate with write-combining policy

For nRSP, we modified OpenCL runtime environment and local L1 data and L2 cache protocols to implement RSP's remote synchronization operations using *Named*-flush and *Named*-invalidation operation which is explained in Section 6.

7.1.1. Workloads

Our benchmarks are chosen from Pannotia Benchmark Suite [24]. We modified our selected graph benchmarks to benefit from dynamic load-balancing. We utilized a lock-free implementation of work-stealing that is described in [25]. In our implementation, each workgroup owns a task queue. Initially, the total number of graph nodes are distributed to task queues. Each workgroup starts working on its own local task queue. When a workgroup runs out of work, it tries to steal from other queues. Once all task queues are empty, kernel exits. Workgroups dequeue from their local queues' tail and steal from other task queue's head. Collision is possible when dequeing and stealing and synchronization must be used when accessing to the task queues. We run our benchmarks with various input graphs selected from 9th and 10th DIMACS Implementation Challenge[16] and from Sommer[17].

Our benchmarks and their inputs are as follows: **`MIS.1`** and **`MIS.2`** implement *Maximal Independent Set* algorithm and use `itdk0304_rlinks_undirected_MAX` and `caidaRouterLevel` as input, respectively. **`PRK.1`** and **`PRK.2`** implement *Page Rank* algorithm and use graphs `coAuthorsDBLP` and `smallWorld`, respectively. **`SSSP.1`** and **`SSP.2`** implement *Single Source Shortest Path (SSSP)* and use graphs `USA-road-BAY` and `USA-road-COL`, respectively.

7.1.2. Evaluation scenarios

We use five execution scenarios for evaluation. In **`Baseline`** scenario, dynamic load-balancing is not implemented, but accesses to task queues are guarded with synchronization using global-scope. Baseline scenario provides a baseline for us. **`Steal-only`** implements work-stealing using global-scoped synchronization. It only gains performance benefits from dynamic load-balancing. **`Scope-only`** does not implement dynamic load balancing. So it can utilize local-scoped synchronization and gains performance improvement coming from low-cost local-scoped synchronization. **`bRSP`** uses first implementation of RSP's remote synchronization operations

[16]DIMACS. (2017). DIMACS Implementation challenges [online]. Website http://archive.dimacs.rutgers.edu/Challenges/ [accessed 00 September 2021].

[17]Sommer C. (2010). Graphs [online]. Website http://www.sommer.jp/graphs/ [accessed 00 September 2021].

that make use of broadcast cache-flush and broadcast cache-invalidation operations. bRSP gains performance improvements from dynamic load-balancing and local-scoped synchronization with local-owner's frequent accesses. **nRSP** uses our new implementation of RSP's remote synchronization operations utilizing *Named*-flush and *Named*-invalidation operations. nRSP gains performance improvements from dynamic load-balancing, local-scoped synchronization with local-owner's frequent accesses, and *Named*-flush/invalidation operations.

7.2. Results

7.2.1. Speedup

Figure 7 shows a comparison of performance improvements of Steal-only, Scope-only, bRSP, and nRSP scenarios relative to Baseline scenario for 64 CU and 128 CU GPU devices. We observe similar performance trends for 64-CU GPU device and 128-CU GPU device.

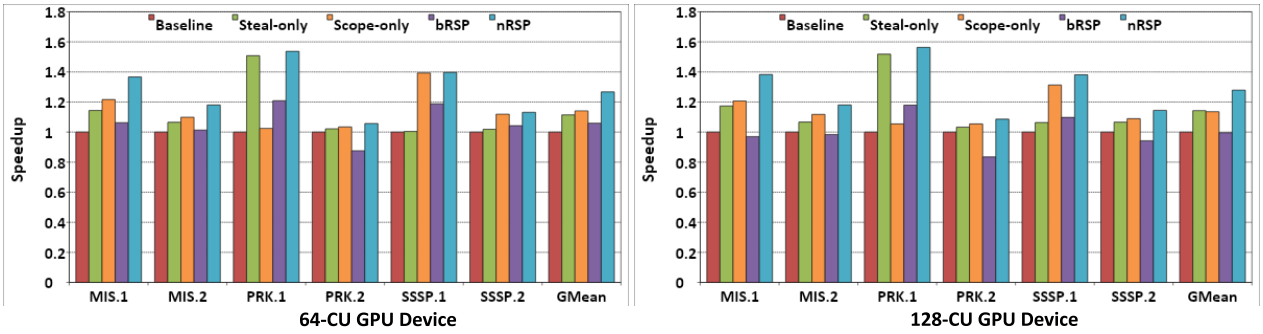

Figure 7. Speedups with our benchmarks (MIS.1, MIS.2, PRK.1, PRK.2, SSSP.1, and SSSP.2) using *Baseline*, *Steal-only*, *Scope-only*, *bRSP*, and *nRSP* scenarios on 64CU and 128CU GPU devices.

We observe good performance improvements with Scope-only scenario for all benchmarks both on 64 and 128 CU GPU devices. Use of lightweight local-scoped synchronization helps with synchronization overhead and retaining locality in local caches. Depending on amount of locality that exists in benchmarks, the performance benefits of using local-scoped synchronization vary. Scope-only scenario shows 14% and 13% performance improvements on the average on a 64 and 128 CU GPU devices, respectively. SSSP.1 gets the highest performance improvement with Scope-only scenario.

The performance improvement that we could get with Steal-only scenario depends on how much load imbalance that the application faces with its input. It could provide significant performance improvement for some benchmarks. We observe upto 50% and 52% performance improvements with PRK.1 benchmark running on 64 CU and 128 CU GPU devices, respectively. It provides 11% and 14% performance improvements on the average on 64 CU and 128 CU GPU devices, respectively.

Despite bRSP uses local-scoped synchronization, it cannot benefit as much from locality since local caches are flushed and invalidated heavily. Synchronization overhead becomes very significant with increased number of CUs. bRSP shows 5% performance improvement and 1% performance slowdown on the average on 64 CU and 128 CU GPU devices respectively. At the most, it exhibits 20% and 17% speedup with PRK.1 benchmark on 64 CU and 128 CU GPU devices, respectively. It shows up to 17% slowdown for PRK.2 benchmark.

While flushing and/or invalidating only participating agents' local caches, nRSP greatly reduces the remote synchronization overhead and manages to limit the effect of remote synchronization on locality. As a result, nRSP sustains the performance improvement on large scale GPUs. On the average, it demonstrates

27% and 28% speedups on the average on 64 CU and 128 CU GPU devices, respectively. It shows up to 56% performance improvement with PRK.1 benchmark on a 128 CU GPU device.

7.2.2. L2 cache accesses

In Figure 8, on the left, we present L2 accesses for five scenarios relative to Baseline while running our benchmarks on a 128 CU GPU device. Use of local-scoped synchronization leads to reduction in L2 accesses and that contributes to performance improvements. While we observe significant reduction in bandwidth use for Scope-only and nRSP scenarios, we observe mixed results for bRSP scenario. This is due to fact that excessive use of cache-invalidation operations harms the locality in L1 data caches. Scope-only and nRSP scenarios show up to 25% and 24% reduction in L2 accesses on a 128 GPU device, respectively.[18]

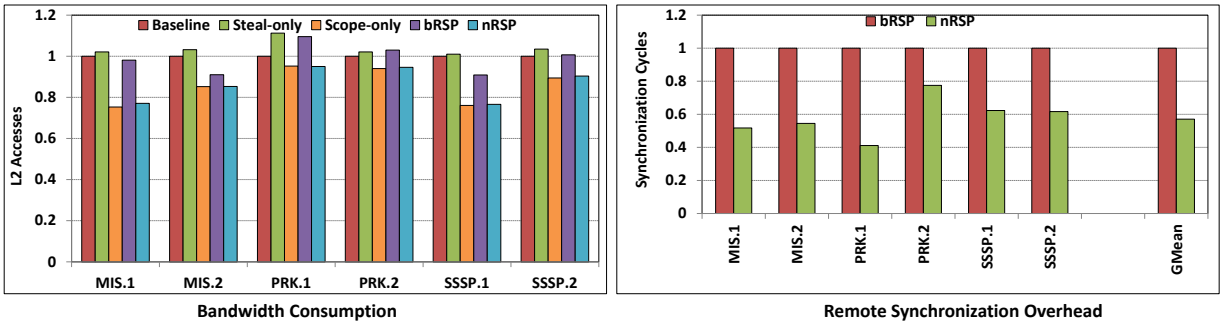

Figure 8. Illustrating *Bandwidth usage* (Left) and *remote synchronization overhead* (Right) using *Baseline*, *Steal-only*, *Scope-only*, *bRSP*, and *nRSP* scenarios running our benchmarks (*MIS.1*, *MIS.2*, *PRK.1*, *PRK.2*, *SSSP.1*, and *SSSP.2*) on a 128CU GPU device. While L2 accesses are represented relative to *Baseline* scenario, remote synchronization cycles are represented relative to bRSP scenario.

7.2.3. Overhead of remote synchronization operations

On the right of Figure 8, we demonstrate the overhead of remote synchronization operations for bRSP and nRSP scenarios on a 128 CU GPU device. nRSP scenario greatly lowers the overhead of remote synchronization operations. nRSP shows 43% reduction in overhead when compared to bRSP scenario. It shows up to 59% improvement when running PRK.1 benchmark.[19]

8. Hardware cost and limitations of nRSP

We modify the L1 data and L2 cache protocols to realize the remote synchronization operations with *Named*-flush and *Named*-invalidation operations. While nRSP's implementation increases the complexity of cache protocol, we believe it is much less compared to RSP's first implementation, sRSP and hLRC. Our implementation does not add any hardware. nRSP's static naming approach is based on the idea of owner computes model. nRSP cannot be used for any applications that is constructed with a more dynamic and irregular computing model. This limits the applicability of nRSP.

[18] While we illustrate and discuss results for 128 CU GPU device for brevity, we observe the similar trends on a 64 Compute Unit GPU.

[19] While we illustrate and discuss results for 128 CU GPU device for brevity, we observe the similar trends on a 64 Compute Unit GPU.

9. Conclusion

GPUs utilize synchronization for preserving data consistency. Unfortunately, synchronization becomes quite costly for large number of threads. High synchronization overhead can be avoided with use of local-scoped synchronization [1] within a subset of threads. Asymmetric sharing cannot benefit from scoped-synchronization due to its dynamic characteristic. RSP [3] introduces remote scope promotion to support use of local-scope in asymmetric sharing. Unfortunately, first realization of RSP semantics employs quite costly broadcast cache-flush and cache-invalidation operations that do not scale for large scale GPUs.

In this paper, we presented a novel mechanism to identify local-owner's cache affinity in asymmetric sharing and apply *Named*-invalidation and *Named*-flush operations when performing a remote synchronization. nRSP tries to minimize remote synchronization overhead, obtains significant speedup for applications that use asymmetric sharing and scales for big GPU devices. When running on 128 CU GPUs, with nRSP 28% speed up (on the average) is obtained.

Acknowledgment

This research is supported by the Scientific and Technological Research Council of Turkey (TÜBİTAK) under the Career Development Program (CAREER), grant number 117E053.

References

[1] Hower D, Hechtman B, Beckmann B, Gaster B, Hill M et al. Heterogeneous-race-free memory models. SIGPLAN Not. 2014; 49 (4): 427–440.

[2] Blumofe R, Leiserson C. Scheduling multithreaded computations by work stealing. J. ACM 1999; 46 (5): 720–748.

[3] Orr M, Che S, Yilmazer A, Beckmann B, Hill M et al. Synchronization Using Remote-Scope Promotion. SIGPLAN Not. 2015; 50 (4): 73–86.

[4] Singh I, Shriraman A, Fung W, O'Connor M, Aamodt T. Cache coherence for GPU architectures. In: IEEE 19th International Symposium on High Performance Computer Architecture (HPCA); 2013. pp. 578-590.

[5] Ren X, Lis M. Efficient Sequential Consistency in GPUs via Relativistic Cache Coherence. In: IEEE International Symposium on High Performance Computer Architecture (HPCA); 2017. pp. 625-636.

[6] Tabbakh A, Qian X, Annavaram M. G-TSC: Timestamp Based Coherence for GPUs. In: IEEE International Symposium on High Performance Computer Architecture (HPCA); 2018. pp. 403-415.

[7] Sinclair M, Alsop J, Adve S. Efficient GPU synchronization without scopes: saying no to complex consistency models. In: The 48th International Symposium on Microarchitecture (MICRO-48); Association for Computing Machinery, New York, NY, USA; 2015. pp. 647–659.

[8] Choi B, Komuravelli R, Sung H, Smolinski R, Honarmand N et al. DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism. In: International Conference on Parallel Architectures and Compilation Techniques; 2011. pp. 155-166.

[9] Fung W, Singh I, Brownsword A, Aamodt T. Hardware transactional memory for GPU architectures. In: The 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44); Association for Computing Machinery, New York, NY, USA; 2011. pp. 296–307.

[10] Xu Y, Wang R, Goswami N, Li T, Gao L et al. Software Transactional Memory for GPU Architectures. In: The Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '14); Association for Computing Machinery, New York, NY, USA; 2014. pp. 1–10.

[11] Cederman D, Tsigas P, Chaudhry M. Towards a software transactional memory for graphics processors. In: The 10th Eurographics conference on Parallel Graphics and Visualization (EG PGV'10); Eurographics Association, Goslar, DEU; 2010. pp. 121–129.

[12] Villegas A, Asenjo R, Navarro A, Plata O, Kaeli D. Lightweight Hardware Transactional Memory for GPU Scratchpad Memory. IEEE Transactions on Computers 2018; 67 (6): 816-829.

[13] Ren X, Lis M. High-Performance GPU Transactional Memory via Eager Conflict Detection. In: IEEE International Symposium on High Performance Computer Architecture (HPCA); 2018. pp. 235-246.

[14] Villegas A, Navarro A, Asenjo R, Plata O. Toward a software transactional memory for heterogeneous CPU–GPU processors. J Supercomput 2019; 75: 4177–4192.

[15] Li A, Braak G, Corporaal H, Kumar A. Fine-Grained Synchronizations and Dataflow Programming on GPUs. In: The 29th ACM on International Conference on Supercomputing (ICS '15); Association for Computing Machinery, New York, NY, USA; 2015. pp. 109–118.

[16] Wang K, Fussell D, Lin C. Fast Fine-Grained Global Synchronization on GPUs. In: The Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19); Association for Computing Machinery, New York, NY, USA; 2019. pp. 793–806.

[17] Vasudevan N, Namjoshi K, Edwards S. Simple and fast biased locks. In: The 19th international conference on Parallel architectures and compilation techniques (PACT '10); Association for Computing Machinery, New York, NY, USA; 2010. pp. 65–74.

[18] Dice D, Moir M, William S. Quickly Reacquirable Locks. 2010.

[19] Alsop J, Orr M, Beckmann B, Wood D. Lazy release consistency for GPUs. In: The 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO); 2016. pp. 1-14.

[20] Yilmazer-Metin A. sRSP: An efficient and scalable implementation of remote scope promotion. Concurrency Computat Pract Exper. 2022; 34 (9): e6483.

[21] Gaster B, Hower D, Howes L. HRF-Relaxed: Adapting HRF to the Complexities of Industrial Heterogeneous Memory Models. ACM Trans. Archit. Code Optim. 2015; 12 (1): 1–26.

[22] Hechtman B, Che S, Hower D, Tian Y, Beckmann B et al. QuickRelease: A throughput-oriented approach to release consistency on GPUs. In: IEEE 20th International Symposium on High Performance Computer Architecture (HPCA); 2014. pp. 189-200.

[23] Binkert N, Beckmann B, Black G, Reinhardt S, Saidi A et al. The gem5 simulator. SIGARCH Comput. Archit. News 2011; 39 (2): 1–7.

[24] Che S, Beckmann B, Reinhardt S, Skadron K. Pannotia: Understanding irregular GPGPU graph applications. In: IEEE International Symposium on Workload Characterization (IISWC); 2013. pp. 185-195.

[25] Cederman D, Tsigas P. Dynamic load balancing using work-stealing. Hwu Wen-mei W., GPU Computing Gems Jade Edition. In: Applications of GPU Computing Series Burlington, MA; Morgan Kaufmann; 2012. pp. 485-499.