GPU Concurrency and Consistency

Dennis Sebastian Rieber
Technische Informatik
Universitaet Heidelberg
Email: rieber@stud.uni-heidelberg.de

***Abstract*—Modern graphics processing units (GPUs) are gaining popularity and become more popular for general purpose computing. However, Alglave et al. showed in a recent publication that the memory behavior of software and hardware does not always behave as specified in the documentation. This report takes a look current consistency models for GPGPU computing and the advantages and disadvantages of recent models.**

After that, it takes a look at emerging technologies that introduce new concepts for software and hardware to create an easier user-experience while maintaing or improving performance.

I. Introduction

While GPGPU computing is gaining popularity, the programming model and memory behaviour isn't as well understood as traditional CPU computing. Using a set of Litmus test Alglave et. al demonstrated weak memory behaviour and for both NVidia and AMD hardware. While weak memory itself isn't a problem, they could demonstrate that documentation, instructions and examples by the vendors were flawed and produced unintended results. For example, the message passing litmus in figure 1 could not be implemented reliably on certain architectures. The goal was to isolate the memory-accesses 11 and 21 from 13 and 23 through memory fences. However, on NVidia's Tesla C2075 the fences were not working as advertised when targeting L1 cache, resulting in the inability to reliably communicate between to CTA. And on AMD's GCN 1.0 Architecture fence instructions were removed by the compiler, without any warning. The research also showed that increased stress on the hardware, more running threads and memory accesses, inflates the probability of unintended results. However, developers need to be sure that they are writing correct software, so the issues lies with the possibility and not the probability of a failure. Lacking documentation is identified as a source of these problems, especially in the area of memory consistency. [1]

This report will talk about how the field of GPU memory models looks like today and what is planned for the future. Section II looks at consistency models in general, where they become important, how they affect the users and why things aren't as simple on GPUs. Section III looks what current programming models, represented by OpenCL, offer in terms to memory ordering and where possible flaws are. Section IV introduces the reader to the concept of DeNovo coherence and how changes this also influences the consistency.

`Init: x=y=0; inter-CTA`	`Exp. Result: r1=0, r2=1`

`T1`
`11: x = 1`
`12: fence`
`13: y = 1`
	`T2`
	`21: r1 = y`
	`22: fence`
	`23: r2 = x`

Fig. 1. Message-Passing Litmus

II. Memory Consistency

Before we take a look at GPU Consistency the term of consistency in general is introduced, along with some more popular consistency concepts for CPUs.

One definition for consistency is: "Consistency dictates the value that a read on a shared variable in a multithreaded program is allowed to return" [2]. While this definition may sound simple at first, it bears some implications that are crucial to the understanding of consistency.

- Does hardware has to guarantee that a write is visible to all threads at the same time? This is where coherence has a strong influence on consistency.
- Is reordering of loads and stores allowed?
- How and where has the user to intervene to maintain an intended ordering?
- Is stale data allowed in the system?

Every component that alters code needs to stick to the contract that is defined by the consistency model, from the user over compilers to hardware. If one component, for example the compiler, falls out of line a correct execution can not be guaranteed anymore. As an example, lets take a look at a basic producer-consumer pattern. In this pattern the particular order of loads and stores is important to maintain correctnes. This means that the order of instructions given by the user needs to be maintained, otherwise correct exection is not guaranteed. If the consistency model does not give a guarantee for this by default, means to enforce the ordering need to provided to the user. The user himself has to aware of these cirumstances and use the tools provided by the languange. Unlike coherence, which is usually invisible to user, consistency is visible to the user and different models of consistency have different effects on how a user has to write software in order to maintain correctness. But for all models, the user has to know how the model is going to behave, especially for concurrent programming.

In a strictly single-threaded environment consistency is a lesser problem because reader and writer are always the same

atomic<int> A,B; int Dat=0		Exp. Result: R1 = 1, R2 = 1
// T1 11:Dat = 1; 12:A.store(1,seq);	// T2 21:while(!A.load(seq)); 22:R1 = Dat; 23:B.store(1,seq);	// T3 31:while(!B.load(seq)); 32:R2 = Dat;

Fig. 2. Transitive message-passing example.

entity. Concurrency however can create conflicts in concurrent code, that isn't visible immediately. In this context a conflict represents the situation of two accesses to a shared variable from different threads, where at least one access is write. Because the instructions of multiple threads interleave in a non-controllable faishon during the parallel execution, the consistency model has to provide rules on what has to be read, and how this has to be enforced.

In the following this reports discusses a selection of consistency models that are relevant for this report and gives a short summary how it influences users, compilers and hardware.

A. Sequential Consistency

Sequential consistency was defined 1979 by Lamport and describes a concurrent consistency model, that most intuitive for the user. "...the results of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program." [6] This means that all memory operations, especially write operations, are perceived in the same order by all processors, and at the same time. It does not matter, if this is the order the operations were actually issued in.

While this concept is easy to implement on systems with a bus interconnect, it becomes inefficient on modern systems like Hyper-Transport because every time a memory operation is issued, no other processor can do so, because it has to wait for the preceding one to complete.

B. Processor Consistency

This model is the one used by many modern processors, including the x86 processor family. Each processor sees its own write operations in order, using a store buffer, but the write operations of other processors may not appear in order. This is easy to realize with modern system interconnects and saves the processor from stalling on unfinished write operations and coherence operations. To maintain the processors intrinsic ordering, a store buffer is used that additionally allows own reads to bypass outstanding writes. Each read makes a look-up in the store-buffer to get the latest value, if present. Every other ordering (W2W, R2R, R2W) is maintained.

The user however, suffers from exposure to hardware structures, namely the store buffer. In order to guarantee visibility at certain points memory fences have to issued to flush the store buffers.

C. Weak Consistency

Weak consistency realeases every ordering and gives virtually no guarantees in which order memory operations are happening. Synchronization access falls back to sequential consistency, where (1) all memory operations have to be finished before the synchronization, (2) no memory operations are allowed during a synchronization access and (3) normal data operations resume after synchronization has finished.

This allows the compilers and hardware to freely rearrange access in order to maximize performance. In hardware, this allows coalescing and out-of-order execution.

However, this also means the user has to be carefully synchronize each critical point where the actual order of execution matters to maintain correctness. Also, it introduces potential for errors or over-synchronization which in the end nullifies the performance gained by the optimization freedom.

D. Data-Race-Free

While the preceeding models have been mainly hardware oriented, Data-Race-Free(DRF) is a more software oriented model that appears in many specifications of programming languages, such as Java [2], C++11 [2] or OpenCL [7] and has become a de-facto standard for modern concurrent languages. It was observed, that a well written program is either data race free or correctly synchronized. This observation, together with the ability to distinguish between data and synchronization access on all levels is leveraged to create the appearance of sequential consistency for the user, but allows relaxation of the memory ordering for compiler and hardware. DRF-0 defines a happens-before order $\xrightarrow{po}$ for all memory operations in a program and $\xrightarrow{so}$ for all synchronization operations. This order is bound to the program order and the overall happens-before order $\xrightarrow{hb}$ is defined as the irreflexive, transitive closure of $\xrightarrow{po}$ and $\xrightarrow{so}$.

$$\xrightarrow{hb} = (\xrightarrow{po} \cup \xrightarrow{so})^+$$

In an multi-threaded environment, the $\xrightarrow{hb}$ of each thread T interleaves to form a global order $< T$.[1]

E. C11 Example

In C11, DRF-0 is used to specify the concurrency memory behaviour. Other than Java, no guarantees are given in case of a data race. Every event of a race is handled by the programmer, using sequentially consistent atomics to resolve the race. C11 further allows the specification of other memory orderings, especially more relaxed models. However, this comes with the

price of much higher complexity and unwanted side-effects. For example, it is possible that the relaxed consistency "leaks" out of or into a library breaking code that is expected to be sequentially consistent, if not used properly.

Figure 2 shows an example for passing a value *Dat* from T1 to thread T3 over a middle-man T2. The conflicting accesses of *A* in T1 and T2 are resolved using stores and loads with single-copy atomicity which count as synchronization operations. This means that the load from *Dat* can not be reordered with the store to *A* without violating the $\xrightarrow{hb}$ T1. The same applies to the handshake between T2 and T3, $A.ld \xrightarrow{hb} Dat.ld \xrightarrow{hb} B.st$ in T2 and $B.ld \xrightarrow{hb} Dat.ld$ in T3. This chain of orderings creates a transitive relation between T1 and T3 and ensures the correct transmission of *Dat*.

The C11 standard offers the option to relax the memory ordering for performance gains, at the risk of introducing unwanted race conditions, and even leaking of weak memory behaviour out of their intended scopes.

III. GPU Consistency - Tradiational Approach

When developing memory ordering models for GPUs, the unique hardware architecture creates new problems that need to be addressed by the model. CPUs generally live in a flat world with a single memory address space and no grouping of threads. Or at least they are built to appear in such a fashion because the memory hierarchy that is created by the caches in not accessible for the user.

The basic concepts of GPUs requires an adjusted memory ordering model. Both, threading model and memory are hierarchical and thus ordering needs to be managed on multiple levels. Threads are grouped into three-dimensional cooperative thread arrays (CTAs), which themselves are then again group in three-dimensional structures (Grid in CUDA, NDRange in OpenCL). Each thread and each CTA has an ID to exclusively identify a thread. All threads have access to the global memory of the device, additionally all threads inside a CTA share an exclusive local memory (scratch-pad in CUDA) that is private to the executing compute unit. In this section we will present two concepts that dive into this complexity. First OpenCL 2.0 will represent the state of the art programming frameworks for GPUs.

A. OpenCL 2.0

Based on the C11 standard, OpenCL 2.0 is extended to cater the needs to GPGPU computing. The most significant change is taking the separated address spaces of local and global memory into account by creating multiple, hierarchical happens-before orders. These so called scopes determine the visibility of atomic operations and are designed to reduce the cost of synchronization operations. For example, if the operations only happens inside a specific CTA, only the threads belonging to it need to see the modifications. This way other CUs are not affected by the resulting flushing of caches and buffers, which reduces the overhead for these operations. The resulting order of modifications to an atomic object is independent between local and global memory.

Just like C11, OpenCL 2.0 uses relaxed memory ordering that appears to be sequentially consistent, if sequentially consistent atomics are used. Again, the memory ordering can be relaxed by the user.

This concept aligns with hierarchical thread and memory model of GPUs, but also need the user to always find the smallest possible scope to utilize the potential performance. Figure 3 shows how the message-passing over a middle-man is realized in OpenCL. It is the same example as Figure 2, but now the atomics are bound to the scopes happens before order. The used scope needs to include all executing threads, otherwise the ordering is not guaranteed any more. If all threads are in the same CTA, the scope "work_group" would suffice to ensure ordering. The synchronization would only affect the executing CU and be relatively cheap, since it can be performed in local memory, if A and B live in this address space. If one of the threads lives in another CTA, the situation is more complex.

For this example we assume T3 lives in another CTA than T1 and T2.

1) The user selects the scope "global" for all atomics and synchronization works as expected because all the CUs on the device, even those not directly involved in the communication, are forced to participate, what makes this an expensive operation. All CUs need to stop working and flush the buffers and caches to the global memory. If the internal interconnect is unordered, it also needs to wait for an acknowledgement that all CUs are finished and normal memory operations are allowed to resume.
2) The handshake on A between T1 and T2 is performed in the "work_group" scope and the handshake with B is performed in "global" scope. Because the action to A is not global it is not included into the global happens before order. B however is in the global scope which makes local atomic A appear like a normal memory operation that is allowed to be reordered. Due to this, there are no guarantees that the program order will be maintained by T2. This is called a heterogeneous race. The result is displayed in Figure 4, where the two happens-before orders run independently.

This example shows that while possibly increasing the performance, also new pitfalls are created. Since there is no transitive scope relation, not all problems actually benefit from the different scopes because they need to fall back the expensive global scope to prevent heterogeneous races. [7]

B. Heterogenous Race Free (HRF)

HRF proposes a more formalized model of OpenCL 2.0, although it was released before OpenCL 2.0 was officially specified. The authors propose two related models, (1) HRF-Direct which roughly corresponds to OpenCL 2.0. (2) HRF-Indirect is an extension that allows transitivity. HRF-Direct addresses the missing transitivity and redefines how synchronization-order $\xrightarrow{so_S}$ and memory operation-order $\xrightarrow{po}$ are connected. Note the new subscript "S" in $\xrightarrow{so_S}$ which represents the scope an

atomic<int> A,B; int Dat=0	Scope = device || local	Exp. Result: R1 = 1, R2 = 1
// T1 11:Dat = 1; 12:A.store(1,seq,scope);	// T2 21:while(!A.load(seq,scope)); 22:R1 = Dat; 23:B.store(1,seq,scope)	// T3 31:while(!B.load(seq,scope)); 32:R2 = Dat;

Fig. 3. Transitive message-passing in OpenCL 2.0

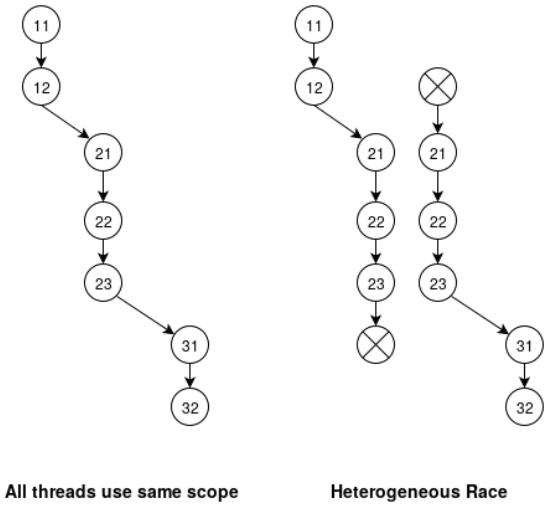

Fig. 4. Race caused by disjoint synchronization orders. It might fail to run correctly, depending on the chosen scopes

order belongs to. HRF-Direct formalizes the unification of all scopes as the global happens-before order $\xrightarrow{ghb.d}$

$$\xrightarrow{ghb.d} = \bigcup_{\forall S \in K} (\xrightarrow{po} \cup \xrightarrow{so_S})^+$$

where K represents all scopes existing in the executing kernel. This leaves all the separate scopes disjoint, creating the transitivity problem. Note this is also an approximate formalization of OpenCL 2.0.

To resolve this, HRF-Indirect relates all $\xrightarrow{so_S}$ by moving the unification of all scopes into the individual happens-before orders, creating the new global happens-before order $\xrightarrow{ghb.i}$

$$\xrightarrow{ghb.i} = (\xrightarrow{po} \cup \bigcup_{\forall S \in K} \xrightarrow{so_S})^+$$

While removing the possibility of heterogeneous races, this change in the formalization has a big impact on how efficient the synchronization can be realized in hardware. Where $\xrightarrow{ghb.d}$ only has to flush addresses hat are directly involved in the synchronization to the visibility plane, $\xrightarrow{ghb.i}$ needs to flush the whole visibility plane in order to address possible transitive relations.

Consider the example in Figure 3 again, in the scenario of T1 and T2 living in another CTA than T3. Now the execution where T1 and T2 are synchronizing locally and T2 and T3 globally is valid and also performing better than the option to always synchronize on the global plane. However, now all local synchronizations perform worse. [4]

C. The road so far

OpenCL and HRF as presented in the preceding section share the basic idea of using explicit scopes to order memory access on the smallest required level to increase the performance. This places the burden of memory ordering optimization on the user and relies on his knowledge about the consistency model and the underlying hardware architecture. By staying as close as possible to the existing frameworks, complexity has increased compared to CPUs, which lays more responsibility onto the user. But with this increased complexity, also new sources for errors have been introduced that are not intuitive, especially for novice users.

While not mentioned in the sections above, NVidias CUDA is basically using the same principles like OpenCL, but the documentation is heavily lacking detailed information on memory consistency, what is the reason OpenCL was used as a representation of today's models.

Another issue is the decreased performance portability that scopes cause. If future hardware architectures change, so might the required scopes and legacy applications could lose performance due to sub-optimal scoping. In conclusion, while the situation is better documented than it was when Alglave et. al published their research, the complexity for user remains the same. In the following section a new approach to this problem is introduced.

IV. DeNovo Coherence

While coherence and consistency are orthogonal problems, they are not disjoint. Consistency is giving the user a guideline which value to expect upon reading a shared variable, and it is the responsibility of coherence to actually deliver this value to the reader. It has to keep track where the latest value of this address is to be found.

The previously presented models in this report are all based on current GPU programming frameworks and hardware architectures and try to do best they can while maintaining these established technologies. The research group that developed DeNovo Coherence took a radical step and redesigned the way a user is interacting with concurrency and coherence in a parallel environment. The MESI protocol and its derivatives are carrying the responsibility to always deliver a correct value alone and are invisible to the user. MESI are write-invalidate protocols where the ownership upon an address is claimed by the writer and each write-miss in the cache triggers a

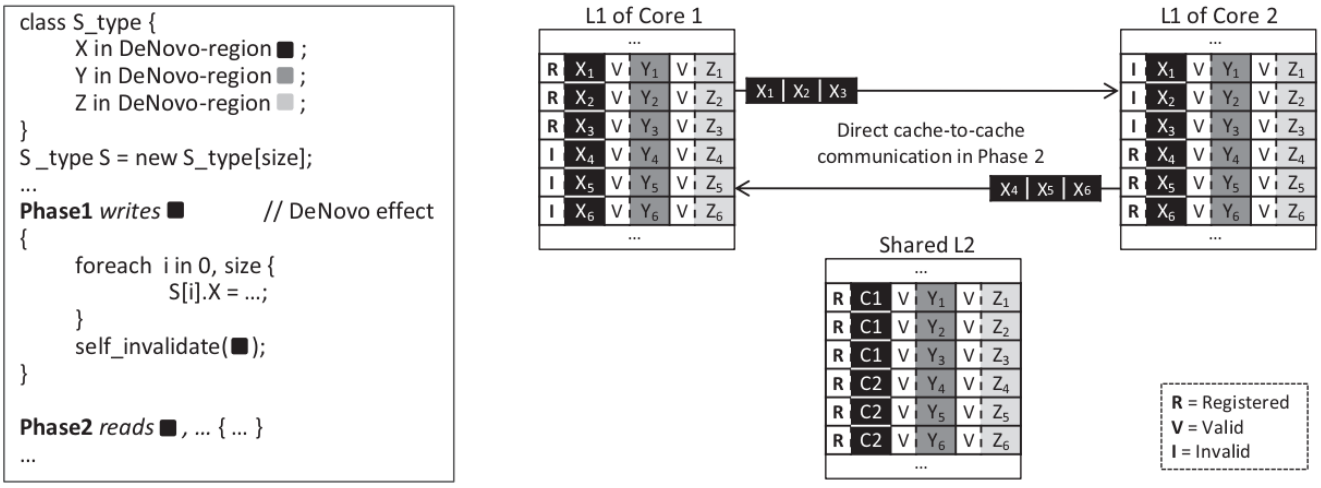

Fig. 5. DeNovo Coherence example.

lengthy chain of events where the hardware makes sure no other cache has ownership of this cache-line. During this process, (1) directoy lookups are performed, (2) an invalidate needs to be sent to each sharer and (3) all acknowledgements need to be received to authorize the coherence-state transition to ”Modified”. The cache line in the core that issued the original write operation lingers in a transient state until the coherence protocol has handled all the necessary steps. During this time another core could try to claim ownership on a different address inside the same cache line. After that a core that is reading this cache line then triggers the process that restores the ”Shared” state to the cache line, which also requires the propagation of new value. The reason for this expensive process are write operations which can appear at any point of execution to every possible address in the shared memory.

The DeNovo research team claims that modern parallel software frameworks like Deterministic-Parallel-Java (DPJ) already contain information to guarantee data-race-freedom in parallel phases through software and that it is usable to create a simpler HW/SW-Hybrid coherence protocols.

A. Deterministic Parallelism

Discipled shared-memory programming models like DPJ or Cilk-Plus allow a redesign of current systems towards models that are more efficient in terms of complexity, power and performance. These languages have parallel phases, like a ”parallel_for” or ”fork-join” blocks that are automatically parallelized by the compiler/tooling. The following conditions need to apply for a framework to be compatible to DeNovo:

1) Guarante of data-race-freedem and deterministic semantics
2) Concepts for structured parallel control
3) The information about effects on shared-memory accesses. In other words, which addresses are read and or written by which thread during a parallel phase.

This allows the implementation of the following principles. First, the knowledge of when and where which address is accessed by whom allows caches to self-invalidate stale data at the end of a parallel phase. This removes the need for hardware directories keeping track of sharers. Second, the guarantee of data-race-freedom through software eliminates the need for writer-invalidation and thus transient states. Third, by removing the need for sharers, cache-to-chache data transfer latency is reduced, since no directory needs to be consulted. Fourth, by guarantee of data-race-freedom in software enables easier implementation of strong memory ordering in hardware.

Figure 5 shows an example of both, the hardware and the software structure of DeNovo. The code example on the left hand side shows how deterministic code could look like. Each Phase is annotated with the region it manipulates and the effect’s nature (read or write). Out of this phase a data-race-free parallel section is create. The compiler then adds information about the self-invalidation for each thread at the end of the phase. When a phase is closed, the all caches are in a legal state. [3]

B. Hardware Architecture

The DeNovo team proposes a cache hierarchy with a private L1 cache and a shared LLC. While Figure 5 shows two cache levels, more can be implemented to deepen the hierarchy. The LLC is responsible to keep cached data locatable by storing it’s value (state ”Valid”), fetching it from memory (state ”Invalid”) or know which core has it currently registered (state ”Registered”). In hardware, DeNovo implements a true three-state protocol consisting of ”Valid”, ”Invalid” and ”Registered” states. Each cache in DeNovo is organized in cache lines, but the states are managed on a ”word” base. While ”Invalid” is self-explanatory, ”Valid” corresponds to the ”Shared” state in ”MSI” and ”Registered” is closest to the ”Modified” state, however there are differences in how the states are transitioned. The guarantee of data-race freedom allows an immediate transition from Valid or Invalid to Registered, even if value

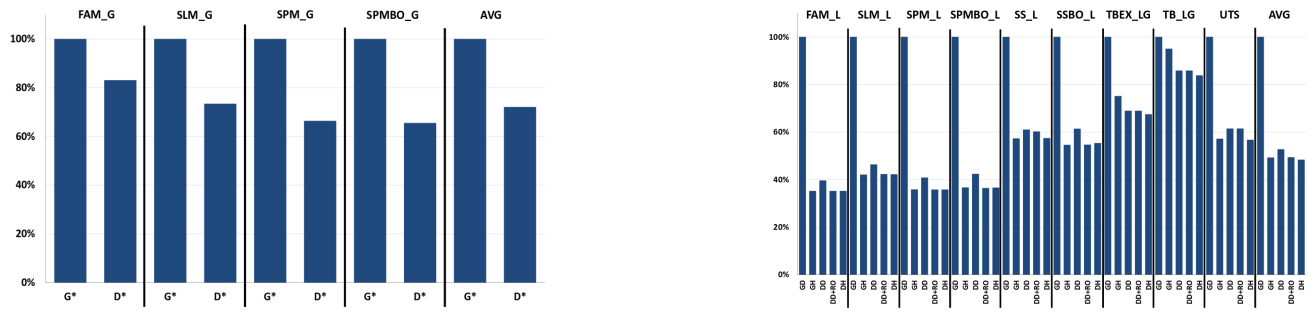

Fig. 6. (L): Runtime of DeNovo-DRF in benchmarks using global synchronization. Runtime is relativ to traditional GPU coherence and consistency. (R): Runtime of HRF and DeNovo-HRF models in benchmarks using global and locaandl synchronization. Runtime is relativ to traditional GPU coherence and consistency. (Images from [8])

is Registered in another core. DeNovo immediately sets the state to Registered and then informs the registry (LLC), which is then also immediately updated. The registry then sends an invalidation request to the previous owner. Due to the knowledge that there are no races, we know that this can not cause a conflict. [3]

C. Non-Determinism

One of the basic problems of parallel programming is the need to safely modify shared variables that might be accessed by multiple threads during the execution and one solution to this problem are Locks. Locks however, are inherently racy which is something the baseline DeNovo eliminates in parallel phases. This so called non-determinism needs to be possible, therefore DeNovoND was introduced. Additional hardware is used to realize these non-deterministic data accesses. The basic requirements for this are (1) a strong isolation of deterministic and non-deterministic operations, (2) the ability to put the deterministic operations into a sequential ordering, even if they contain or are within a non-deterministic construct. In DeNovoND this isolation is created by locks with hardware support, namely Queue-on-Sync-Bit (QOSB) locks. This lock builds a queue across all cores that request access to a certain word, which implicitly serializes all atomic access operations.

To track which addresses are involved in an atomic write operations of this phase, a bloom-filter is used. A bloom filter uses a series of simple hash functions on the address to create an access signature. When the lock is passed between cores, the filters are exchanged and combined. [9] [10]

D. DeNovo for GPU

Using DeNovo coherence, the team showed in a series of simulations that a consistency model with the same complexity with as the CPU's can be implemented for GPU's, without loosing any significant performance in traditional GPU applications with the only point of global synchronization is a kernel restart. However, in scenarios with mid-kernel global synchronization, DeNovo provided a significantly lower execution time, together with lower dynamic energy consumption and decreased network traffic. The left side of figure 6 shows the runtime of benchmarks using DeNovo, relative to the execution with existing models. The *G** represents the reference model, *D** DeNovo. Returning to the terms of the traditional GPU consistency models, DeNovo coherence with DRF on GPUs has only the global synchronization scope, ignoring CU locality. However, DeNovo is also compatible with HRF's scoped synchronization, allowing higher performance at the cost of increased complexity for the user. Again, besides the lower execution time, dynamic energy and network efficiency are improved. The right-hand side of figure 6 shows the relative performance of benchmarks that have local and global synchronization points and compares the reference model *GD* with HRF(*GH*), Denovo-DRF (*DD*), DeNovo-DRF with region optimization (*DD+RO*) and DeNovo with HRF (*DH*) Further research also shows methods to cache the local memory, thus integrating the local address space and it's performance advantages without introducing the increased complexity of scoped synchronization. [8] [5]

V. Summary

When looking at how the programming experience for the user has developed on GPUs in respect to the memory consistency behaviour, a development into a more convenient direction can be noticed. Newer concepts like OpenCL 2.0 and Heterogeneous-Race-Free give more formalization and information on how memory ordering has to be managed by the user. NVidia's CUDA, which has a wide acceptance in GPGPU computing is still poorly documented and leaves much room for interpretation to the user. With HRF-Indirect researches provided a way to prevent a source of unintuitive errors and already increased the user experience. But all the models ultimately try to work around the restrictions given by current hardware and programming frameworks.

A new model is explored by the DeNovo group that leaves the beaten paths and demonstrate that new concepts for software and hardware could enable easier usability by bringing the CPUs more intuitive behaviour to GPGPU programming without loosing performance. Alternatively we can keep the complicated behaviour and gain even more performance, while saving energy. Although the concept looks promising for both CPU and GPU, it has to be accepted by the hardware, compiler and language developers in order to be successful.

REFERENCES

[1] Sarita V. Adve and Mark D. Hill. Weak ordering—a new definition. *SIGARCH Comput. Archit. News*, 18(2SI):2–14, May 1990.

[2] Hans-J. Boehm and Sarita V. Adve. Foundations of the c++ concurrency memory model. *SIGPLAN Not.*, 43(6):68–78, June 2008.

[3] Byn Choi, Rakesh Komuravelli, Hyojin Sung, Robert Smolinski, Nima Honarmand, Sarita V. Adve, Vikram S. Adve, Nicholas P. Carter, and Ching-Tsun Chou. Denovo: Rethinking the memory hierarchy for disciplined parallelism. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, 2011.

[4] Derek R. Hower, Blake A. Hechtman, Bradford M. Beckmann, Benedict R. Gaster, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. Heterogeneous-race-free memory models. *SIGPLAN Not.*, 49(4):427–440, February 2014.

[5] Rakesh Komuravelli, Matthew D. Sinclair, Johnathan Alsop, Muhammad Huzaifa, Maria Kotsifakou, Prakalp Srivastava, Sarita V. Adve, and Vikram S. Adve. Stash: Have your scratchpad and cache it too. *SIGARCH Comput. Archit. News*, 43(3):707–719, June 2015.

[6] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, September 1979.

[7] OpenCL-Group. Opencl 2.0 specification.

[8] Matthew D. Sinclair, Johnathan Alsop, and Sarita V. Adve. Efficient gpu synchronization without scopes: Saying no to complex consistency models. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 647–659, New York, NY, USA, 2015. ACM.

[9] Hyojin Sung and Sarita V. Adve. Denovosync: Efficient support for arbitrary synchronization without writer-initiated invalidations. *SIGARCH Comput. Archit. News*, 43(1):545–559, March 2015.

[10] Hyojin Sung, Rakesh Komuravelli, and Sarita V. Adve. Denovond: Efficient hardware support for disciplined non-determinism. *SIGPLAN Not.*, 48(4):13–26, March 2013.