Concurrent Manipulation of Dynamic Data Structures in OpenCL

Henk Mulder
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
h.mulder-1@student.utwente.nl

ABSTRACT

With the emergence of general purpose GPU (GPGPU) programming, concurrent data processing of large arrays of data has gained a significant boost in performance. However, due to the memory architecture between the host and GPU device and other limitations in the instructions available on GPUs, the implementation of dynamic data structures, like linked list and trees, is not evident. In GPU programming a high number of computing elements (work items) all execute a single instruction on multiple data (SIMD). With dynamically changing data structures, and the absence of locks in GPU programming, one of the biggest challenges is to ensure each work-item operates on its own piece of data. In this paper we show that it is possible to implement (shared) dynamic data structures on GPUs. A kernel memory allocator is used to manage the memory on the GPU device. Lock-free algorithms have been used to implement a linked list. A study to the performance of this implementation showed that the massive parallelism, as found on GPUs, does not improve the performance for the basic functionality of the linked list that was developed. However, dynamic data structures like linked lists, are often used to store intermediate results. Being able to implement these data structures on GPUs can aid developers in implementing other algorithms, that could benefit from parallelisation but use linked lists, on GPUs. By using a kernel memory allocator, memory can be reused during the execution of the program. Thereby offering the possibility to make more efficient use of the memory available on the device, limiting the memory overhead.

Keywords

concurrent, dynamic data structures, GPGPU, OpenCL

1. INTRODUCTION

With the emergence of General Purpose Graphic Processing Units (GPGPUs), a highly parallel processing resource has become available for software developers. However, making use of this computational power is not evident due to the memory architecture between the host system and the GPU, and other limitations of the platform.
Part of a solution to this problem is the OpenCL standard [3], which introduces an open, royalty-free, standard for parallel programming on a wide variety of devices like GPUs, CPUs and FPGAs. In OpenCL the host program manages the execution of kernel programs on the computing devices. In the computing devices multiple work-items, grouped in work-groups, execute the same kernel program. All work-items can access the GPUs global memory. The global memory is similar to what the main memory is for a CPU: it is large, but slow compared to the cache. All work-items in a work-group can also access the work-groups local memory. This local memory is faster, like what the level 2 cache is for a CPU, but only the work-items in the work-group can access the data in that memory. Therefore managing which work-items work on what data is very important. Each work-item also has its own piece of private memory, similar to the level 1 cache on a CPU.
To make sure no two work-items try to alter the same memory simultaneously, OpenCL manages reading and writing to memory in load and store operations. These load and store operations can be ordered with the mem_fence method [1]. For completeness we mention the barrier-function [2]. All work-items in a work-group executing a kernel program must execute this function before they are allowed to proceed. This allows for synchronisation within a work-group.
Since the memory of the host system and the GPU are separated, the data on which the GPU has to work needs to be transferred to the GPU device. If the GPU needs to work on predefined sized blocks of data this is not a problem, since the host can copy the data to the device, let the GPU do its work and retrieve the same size block from the same place. However, in many programs we would like to make use of dynamic data structures, which are often connected by pointing to the memory address of the next data block. Creating and manipulating these structures requires a way to dynamically allocate and connect these blocks. A possible solution to this problem has been introduced by R. Spliet with the kernel memory allocator (KMA) [5]. In this concept a heap is initialized in which the kernel can allocate and free memory with the kernel memory allocator.
Since the location of the nodes in a dynamic data structure are not predetermined, dynamic data structures must be handled in global memory, in order for all work-items to be able to access the data. Managing which work-item can alter what data is very important. With the absence of global locks in OpenCL this brings an extra challenge in implementing dynamic data structures on GPUs. However OpenCL does support several atomic operations in-

23rd Twente Student Conference on IT June 22, 2015, Enschede, The Netherlands.

[1] https://www.khronos.org/registry/cl/sdk/1.0/docs/man/xhtml/mem_fence.html | 2015-05-11
[2] https://www.khronos.org/registry/cl/sdk/1.0/docs/man/xhtml/barrier.html | 2015-05-11

cluding the atomic compare-and-swap (CAS) operation [3]. The CAS-operation compares the contents of a memory location to a given value and, only if they are the same, modifies the contents of that memory to a given new value. Because the swap would fail if the contents of the memory were changed in the meanwhile, this guarantees the operation is done on up-to-date information.
For the implementation of the semi-linked list, the atomic compare-and-swap is used together with the mem_fence to synchronize the changes to the list. In Section 6 we elaborate on the implementation of the semi-linked list, and we take a look at its performance.

2. PROBLEM STATEMENT

Dynamic data structures come in many forms, from linked list to multi-dimensional graphs. To investigate the challenges in implementing dynamic data structures on GPUs, we start by dividing this challenge in smaller problems. We limit our study to implementing the data structures in OpenCL [3]. We make a distinction between semi-linked data structures, in which a node is only referenced once, and double-linked data structures, in which a node can be referenced in two places. As basis we looked at linked lists. For the semi-linked list, we investigated the following research questions:

- How can the basic add, get and delete operations on linked lists be implemented in OpenCL?
- What is the performance of this implementation in time and space?
- How does the performance of this implementation compare to a similar implementation for CPUs?

3. BACKGROUND

3.1 OpenCL

The open computing language (OpenCL) [3], is an open, royalty-free standard that provides a platform-independent standard for parallel programming [4]. With a wide range of compatible devices this enables developers to focus their work on functionality without having to worry about how to comply with different hardware. OpenCL is based on a subset of the C99 programming language. The most noticeable limitations are the absence of recursion and the limitations in using pointers.
In OpenCL the host program controls the available computing devices (for instance the CPU or GPU). These devices consist of computing units, which can have multiple computing items, depending on the architecture. OpenCL abstracts and manages these resources as work groups and work items. The work items in a work group execute kernel code in parallel.
Since GPUs do not have direct access to the memory of the host system, OpenCL also provides methods for transferring data between the host system and the computing devices.
For more details about the programming model and memory architecture we refer to the introduction to the OpenCL programming model [6] by Tompson et al.

[3]https://www.khronos.org/registry/cl/sdk/1.2 /docs/man/xhtml/atomicFunctions.html | 2015-03-27
[4]https://www.khronos.org/opencl/ | 2015-03-27

3.2 Kernel memory allocator

The kernel memory allocator (KMA) [5] is a proof of concept for a heap memory allocator that provides the basic malloc and free operations for OpenCL kernels. The host instantiates a heap, whose size is set by the user specifying the number of, and the size of the superblocks, in the heap. Because the compute device cannot asynchronously communicate with the host it is not possible to increase the heap size after initialisation.
KMA also provides means for managing an array list on the GPU. Our research will involve extending functionality to support dynamic data structures like linked lists and trees.
For more details about the design and underlying design choices in KMA we refer to the thesis [4] by Spliet.

4. RELATED WORK

Research in the performance of concurrent lock-free data structures on GPUs has been done by P. Misra and M. Chaudhuri[2]. They used NVIDIA's platform CUDA, and achieved a significant speedup in manipulating the data structures compared to multi-threaded CPU implementations. In their research memory on the GPU is pre-allocated by the host system and pointers to these blocks are stored in an array. Also deleted nodes are not reused, causing considerable memory overhead for frequently changing structures.
By using KMA, our implementation is able to reuse memory that was no longer needed. Also the size of the blocks of memory to allocate does not have to be fixed, and the size of the data structure is not limited by the size of an array in which pointers are stored. By implementing our data structures in OpenCL, we also bring it to a more general platform.

5. APPROACH

To show how dynamic data structures can be implemented in OpenCL, we developed a kernel for a semi-linked list. To build the structure, we need to allocate memory in order to store the connections between the items in the list. Since OpenCL does not offer functionality to allocate memory on the GPU, we choose to use the Kernel Memory Allocator (KMA) [5]. KMA makes it possible for work-items to allocate, and free memory from a predefined heap of memory. To manage the blocks in the heap, KMA uses a lock-free queue. The implementation of this queue was inspiration for the implementation of the linked list.

We first focussed on the add and get operations of the list. When inserting a node in the list, care must be taken no two work-items try to insert a node at the same place, at the same time. This would result in one node overriding the reference to the other node, and then the other node would drop out of the list. After having developed a way to build a list we focussed on how to remove, or delete items from this list. When a node is being removed it essentially means tying the tail of list, after the to-be-deleted node, to the point where the to-be-deleted node was attached to the list. However, when two work-items simultaneous try to delete two adjacent nodes, care must be taken that the tail of the latter node is not attached to the node being deleted in front of him. This would mean the entire tail of the list would be lost.

To test if the kernels behave as expected, test-cases were developed to simulate the situations described above. Kernels were written to test for concurrent insertion of items in a list, and concurrent deletion of (adjacent) items in a list. After the insertion, or deletion, the number of items in

the list are counted to check if the actual number of items in the list matches with the expected number of items.

The next step was to look at the performance of the implementation, to see if the parallelisation can effectively increase the performance. For this purpose a semi-linked list was implemented in C, with the same functionality as the OpenCL implementation. For a clear view of the effect of the parallelisation, we choose to implement the functionality in the same way as the OpenCL implementation works. This mean that to add a node in the C implementation, also the entire list is being traversed to add the node at the end of the list. Even though this is not the most efficient way to implement this operation, the similarities will give us better insight in what the effect of the parallelisation is on the execution time of the operation. The performance of future improvements can then be compared to these results.

We also looked at memory usage. In OpenCL, data needs to be transferred from the host to the compute device, and back. When memory is required to dynamically create data structures in, this could result in a large memory overhead. In Section 6, we discuss the considerations when implementing dynamic data structures in OpenCL with the use of KMA.

6. RESULTS

6.1 Semi-linked data structures

6.1.1 Implementation

This section presents the design of the semi-linked list. The list consists of the lists entry point, which has a pointer to the first node in the list. The nodes in the list contain a key and a pointer to the next node, with NULL marking the end of the list.
Three operations are defined on the linked list: add, get and delete. The add operation will add a node to the end of the list. The get operation will retrieve the first occurrence of a node with the specified key value. The delete operation will delete the specified node from the list.

In listing 1 is the code used to add a node to a list. When a node is added to the list, a compare-and-swap(CAS) is performed on the pointer to the next node, to replace NULL (marker of the end of the list) with the address of the to-be-inserted node (line 8). After the CAS is a mem_fence (line 9). This mem_fence ensures the store of the new value is committed, before checking if the CAS succeeded. If this CAS succeeds the node is added to the list. If the CAS fails the next node is visited and the process is repeated until it succeeds. Since the next-pointer of the to-be-inserted node is set to NULL, this node is marked as the last node in the list. Thereby the list always has an end, and therefore the add operation will terminate. In listing 2 is the code for a kernel program, to concurrently insert keys in a list. On line 4, memory is allocated for a list node. The key field is set to the correct values, and the add method is used to insert the node in the list. This method is also used to measure the kernel execution time, for performance comparison.

Retrieving a node with the get method means traversing the list until a node with the specified key is found, or the end of the list is reached. If a node is found the address of this node is returned, else the method returns NULL.

To delete a node, the reference to the to-be-deleted node must be replaced with the reference to the list after the node. However, when concurrently removing nodes, care must be taken that the tail of the list is not attached to a node that is also being deleted. Since the memory in OpenCL is 4-byte aligned, the least significant bit in the next-field of a node can be used to mark the node as being in deletion. In listing 3 is the code to delete a node from the list. On line 6 a check is done if the node is already marked to be deleted by another thread. If not, on line 8 the node is being marked to be deleted. Next the list is traversed while trying to swap a reference to the to-be-deleted node with the pointer to the tail of the list. If the swap succeeds, the memory for the node is freed, and 1 is returned to indicate success. If the swap did not succeed, we need to evaluate the next node in the list. On line 16 a check is done if the next pointer is marked for deletion. If that is the case, the node with the focus might currently be deleted by another thread. Therefore, we need to start over again from the start of the list, since we cannot be sure this part of the list is still valid. If the next-field was not marked for deletion, we can continue to the next node.

6.1.2 Testing

To test the implementation of the semi-linked list several tests have been developed. A count method has been implemented to traverse the list and count the number of nodes in the list. This method is used to count the elements in the list after concurrent insertion of multiple nodes and concurrent removal of (adjacent) nodes. If the actual number of nodes in the list matches with the expected number of nodes (size of the old list plus the number of items inserted, or minus the number of items deleted), then the list is still connected, and no nodes dropped out of the list.

6.1.3 Performance

To evaluate the performance, in execution time, a similar implementation of a linked list was developed in C. This implementation has the same computational complexity for inserting nodes. In this implementation, the list needs to be traversed till the end, to insert a node at the end of the list. In measuring the execution time we looked at two different devices:

- **Intel(R) Core(TM) i3 CPU M 370.** This is a CPU with 4 cores, with a clock frequency of 2.40GHz. It has access to a global memory of 4.0 Gb and allows for a maximum workgroup size of 1024 work-items. This CPU is the host system for the OpenCL platform. The CPU is used with the OpenCL platform, and is also the CPU used for comparison with the CPU implementation in C.
- **NVDIA GeForce 310M.** This is a 2 core GPU with a clock frequency of 1468 MHz. The GPU has access to a global memory of 512 Mb, and allows for a maximum workgroup size of 512 work-items.

OpenCL provides means to accurately measure kernel execution time by so-called event profiling. Since the system time of the PC is not updated frequently enough to allow for accurate measurement of building small linked lists with the CPU implementation, we measured the time it took to build multiple of these small lists. The number of lists to build was determined by the size of the list to build. For small lists this number was higher, to gain an execution time that is long enough to provide for accuracy. For building the lists in OpenCL, KMA needs memory to allocate the structures in. In our measurements we used a heap with 64 super blocks, each with a default size of 4096 bytes. This way, the memory to be transferred only differs

Listing 1. Semi-linked list, add node.

```
1  int clSemiLinkedList_add(clSemiLinkedList __global *list, clSemiLinkedListNode __global
       *node) {
2          volatile uintptr_t __global *cursor = (volatile uintptr_t __global *) &(list->
               head);
3          volatile clSemiLinkedListNode __global *cNow;
4          uintptr_t cNode;
5          unsigned int ret=0;
6          node->next = 0;
7          while(1) {
8                  cNode = atom_cmpxchg(cursor, NULL, (uintptr_t) node);
9                  mem_fence(CLK_GLOBAL_MEM_FENCE);
10                 if(cNode==NULL) {
11                         return 1;
12                 } else {
13                         cNow = (clSemiLinkedListNode __global *) cNode;
14                         cursor = (volatile uintptr_t __global *)&(cNow->next);
15                 }
16         }
17         return ret;
18 }
```

Listing 2. Semi-linked list, concurrently add keys.

```
1  __kernel
2  void clSemiLinkedList_addKey(struct clheap __global *heap, clSemiLinkedList __global *
       list, int __global *keys) {
3          int global_x =get_global_id(0);
4          clSemiLinkedListNode __global *node = (clSemiLinkedListNode __global *)malloc(
               heap, sizeof(clSemiLinkedListNode));
5          node->key = keys[global_x];
6          clSemiLinkedList_add(list, node);
7  }
```

Listing 3. Semi-linked list, delete a node.

```
1  int clSemiLinkedList_delete(struct clheap __global *heap, clSemiLinkedList __global *
       list, clSemiLinkedListNode __global *node) {
2          volatile uintptr_t __global *cursor = (volatile uintptr_t __global *) &(list->
               head);
3          volatile clSemiLinkedListNode __global *cNow = (volatile clSemiLinkedListNode
               __global *)(list->head);
4          uintptr_t cNode;
5          uintptr_t tail = node->next;
6          unsigned int ret=0;
7          if((tail &1) == 0) {
8                  /* mark node as in deletion */
9                  node->next = (node->next)+1;
10                 while(cNow != (clSemiLinkedListNode __global *)0) {
11                         cNode = atom_cmpxchg(cursor, (uintptr_t)node, (tail));
12                         mem_fence(CLK_GLOBAL_MEM_FENCE);
13                         if(cNode == (uintptr_t)node) {
14                                 free(heap, (uintptr_t)node);
15                                 return 1;
16                         } else {
17                                 if((cNode & 1) ==0) {
18                                         /* Pointer is clean; Proceed to next */
19                                         cNow = (clSemiLinkedListNode __global *) cNode;
20                                         cursor = (volatile uintptr_t __global *)&(cNow->
                                               next);
21                                 } else {
22                                         /* Pointer is dirty; Start from beginning of
                                               list. */
23                                         cursor = (volatile uintptr_t __global *) &(list
                                               ->head);
24                                 }
25                         }
26                 }
27                 /* failed. Restore next field for node */
28                 node->next = tail;
29         }
30         return ret;
31 }
```

in the size of the array with the keys to insert. In table 1 are the results for building list with the given number of nodes. Building the list was done five times. The averages of those times are in the results. For lists above 2300 nodes the memory on the GPU was not sufficient enough to cope with the transfer of the heap and the array with keys to insert, in combination with other processes using the GPU. Therefore, building larger lists failed.
We can see that the CPU implementation is, by far, the fastest. This could be explained by the amount of memory that needs to be transferred between the host and the computing device in the OpenCL implementations. This memory transfer is not needed in the CPU implementation.
Because the list needs to be traversed to insert a node in the list, we expect a quadratic time complexity for sequential insertion of nodes. As we can see in table 1 this is the case. For parallel insertion we would like to see a better performance. However, we can see that, for parallel insertion, even for larger lists, the time it takes is still quadratic to the number of nodes in the list. This can be explained by the amount of conditional statements in the OpenCL code. Parallel hardware can benefit the most when each work-item can execute the same instruction. If work-items diverge because they branch differently on a conditional statement, hardware might require that one branch is evaluated before the other. Thereby reducing the effect of parallelisation on the performance.

OpenCL manages the transfer of data between the memory of the host system and the GPU. The Kernel Memory Allocator, in its turn, manages the memory on the GPU for the kernels using KMA. However therefore, if a kernel wants to use KMA, all the data in the memory that KMA uses needs to be transferred between the host and the GPU. Since there is no way to asynchronously communicate, from the kernel to the host, it is not possible for KMA to increase the available memory during kernel execution. Therefore the initial heap size must be large enough to meet the memory demands for all work-items. This brings a considerable memory overhead. However, because the memory can be used by al work-items, there is no need for each work-item to reserve the maximum possible amount of memory required. When, for instance, a data set needs to be partitioned into multiple lists, each item will only be in one list at a time. Therefore the amount of memory required is the memory to store all items, and the memory to connect them as a semi linked list. The extra memory required by KMA to do its bookkeeping does need to be taken into account. By default KMA uses superblocks of 4096 bytes. The first superblock is used for bookkeeping. Also each superblock contains a header and a footer with information about the blocks within that superblock.

Using KMA to implement dynamic data structures on GPUs can thus come with a considerable performance penalty in execution time. However it is unlikely that parallelisation will be the choice for optimizing the pure functionality of linked lists. nevertheless linked lists are a common choice for storing intermediate results, or implementing scheduling policies. In algorithms that could benefit from parallelisation, but also need the flexibility of these dynamic data structures, the parallelisation on the entire algorithm could outweigh the loss of speed of using linked list on GPUs. Furthermore, being able to use extra memory when required, and also being able to give this memory back for other work-items to use, can be beneficial for cases where the amount of memory needed per work-item differs a lot.

There is also room for improving the implementation of the linked list. Among the basis of GPGPU programming in OpenCL, The OpenCL Programming Book [7] also contains various subjects for optimizing OpenCL code. Since this study was primarily to explore the caveats in implementing dynamic data structures on GPUs, and to look at the performance issues involved, there has not been a lot of effort in optimization. The implementation is available at http://fmt.ewi.utwente.nl/education/bachelor/208/.

6.2 Double-linked data structures

In our study we also looked at double-linked structures. In such structures a node can be referenced from two places. For example, consider a linked list in which each node points to the next node, and points to its predecessor. This means we can traverse the list forward and backward. However, this also means, when inserting or deleting a node, two pointers must be updated. It is not possible to update two pointers simultaneous, with one atomic CAS. Therefore the synchronisation method as used for the semi-linked list is not applicable. However, there are other techniques to implement lock-free data structures, as described by Sederman et all [1]. In this study, we did not get around on implementing any of these techniques. It should however, be clear that, by using KMA, it is also possible to develop implementations for double-linked data structures, using (known) lock-free algorithms.

7. CONCLUSION

With this study we showed that it is possible to implement dynamic data structures on GPUs using a dynamic kernel memory allocator. The implementation as developed for this study shows that the massive parallelisation on GPUs does not effectively improve the performance of a linked list. A straightforward CPU implementation is faster and brings less memory overhead. However the fact that it is possible to implement dynamic data structures on GPUs, means they can be used for processes where the paralellisation could offer performance benefits. In case of linked lists, this means they could, for instance, be used to store intermediate results, either per work-item, per workgroup or globally shared between all work-items. The effect of the parallelisation on the performance of (lock-free) implementations of other dynamic data structures is left for future research.

8. FUTURE WORK

To improve the current implementation of the semi-linked list in OpenCL, it could be investigated how to reduce the number of conditional statements in the code. Reducing the number of conditional statements should result in less branch diverging, thereby better adhering to the single instruction, multiple data (SIMD) model. It could also be investigated if the computational complexity can be reduced for the add method. By introducing a pointer to the end of the list, the list would not have to be traversed to add a node to the list. However, policies need to be developed for cases in which the last node in a list is removed. The pointer to the end of the list then points to a non-existing node. The list needs to be traversed to find the new last node of the list, but in the meanwhile the end-of-list pointer cannot be used.

In this study we only looked at linked lists. Future re-

Table 1. Inserting nodes in empty list. CPU: Intel(R) Core(TM) i3 CPU M 370. GPU: GeForce 310M.

Nodes	OpenCL CPU (usec)	OpenCL GPU (usec)	CPU C implementation (usec)
128	2144	14198	99
256	9547	40396	375
384	21792	105686	800
512	37940	167632	1325
640	56706	244790	2033
768	80483	362636	2840
896	103054	424999	4000
1024	120602	550810	5250
1152	146733	633239	6466
1280	175352	860757	7333
1408	208957	1042040	9000
1536	242173	1010622	10600
1664	285065	1439403	12100
1792	333750	1642655	14300
1920	369299	1449156	16301
2048	432494	1687129	17301
2176	484262	1883452	18801
2304	545609	2059861	21001
2432	612020	0	23601
2560	669474	0	26601
2688	744410	0	29001
2816	818043	0	31201
2944	901208	0	33002
3072	972537	0	34001
3200	1054012	0	36602
3328	1139842	0	40002
3456	1238072	0	39602
3584	1324289	0	42002
3712	1419699	0	44802
3840	1529863	0	49202
3968	1643151	0	52203
4096	1799220	0	54203

search could focus on more complex structures like trees or graphs. For these, more complex, structures the parallelism found on GPUs could possibly improve the performance of manipulating these structures.

9. REFERENCES

[1] D. Cederman, A. Gidenstam, P. Ha, H. Sundell, M. Papatriantafilou, and P. Tsigas. Lock-free concurrent data structures. *arXiv preprint arXiv:1302.2757*, 2013.

[2] P. Misra and M. Chaudhuri. Performance evaluation of concurrent lock-free data structures on GPUs. 2012.

[3] A. Munshi et al. The OpenCL specification. *Khronos OpenCL Working Group*, 1, 2009.

[4] R. Spliet. *A comprehensive study of dynamic memory management in OpenCL kernels*. PhD thesis, Master's thesis, Delft University of Technology, 2013.

[5] R. Spliet, L. Howes, B. Gaster, and A. Varbanescu. Kma: A dynamic memory manager for OpenCL. 2014.

[6] J. Tompson and K. Schlachter. An introduction to the opencl programming model. *Digital version*, 2012.

[7] R. Tsuchiyama, T. Nakamura, T. Iizuka, A. Asahara, and S. Miki. The OpenCL programming book. 2011.