GGArray - A Dynamically Growable GPU Array

Enzo Meneses[1]

Instituto de Informática, Universidad Austral de Chile, Chile

Abstract. We present an implementation of a dynamically growable array for the GPU that can be fully utilized from the device without the need for synchronization through the host. It offers an interface similar to an array and it is the first step into the construction of a structure similar to a C++ vector that works on a GPU and takes advantage of its architecture. This seeks to benefit the programming of GPU applications that require dynamic memory by providing an option for it and reducing the required memory by avoiding the approach of pre-allocating for the worst case scenario. Our structure is inspired by the LFVector and consists of an array combining multiple of them, taking advantage of the GPU architecture and thread blocks. This produces a structure with reasonable speed on the growth and insertion operations but a really slow access to its elements, the most important characteristic of arrays, making it unsuitable as a general purpose array. However it still offers benefits to applications that need its dynamic aspect and also applications that can use it only for the growth and then use a normal array for static work. This structure is compared with other alternatives such as a pre-allocated static array and a semi-static array that needs to be resized through communication with the host.

Keywords: GPGPU · Dynamic Array · Dynamic Memory · Parallel Algorithms

1 Introduction

Due to their high compute capacity, GPUs have become vitally important in HPC, scientific simulations and other applications that require its capabilities [14]. Furthermore, recent improvements of these device, such as tensor cores and ray tracing cores, have cemented their use in certain areas that receive an even greater benefit from these technologies. GPUs are especially useful when dealing with grid structured data like arrays or matrices offering huge speedups in comparison with other architectures. However, when dealing with graphs, sparse matrices and other structures that doesn't follow the same structure as the GPU memory, it proves difficult to obtain speedups of the same magnitude [4].

This problematic also extends to the use of dynamic memory. Given the impossibility to maintain data contiguously in memory when dynamically allocating it without any kind of global synchronization, it's natural that the use of dynamic memory doesn't provide the same speedups as the ones obtained with

grid structured data. While there are plenty of studies about graph algorithms and sparse matrices [3,1,12,16] for GPUs, some of which also explore the use of dynamic memory. There has been no attempt to implement a dynamic array that works on GPU. Dynamic arrays or C++ vectors are one of the most commonly used structures in programming languages and some of them don't even include static arrays (e.g. python). Dynamic arrays provide an easier way of programming and simpler code designs due to its capacity to grow or shrink as required.

That simplicity is an important benefit for people who's work require programming as a tool, but their effort shouldn't be focused on writing the needed programs. Usually scientists fall under these kind of people, they focus on the study of certain phenomenon which may require extensive computer simulations and the use of GPUs. However, currently it's highly difficult for scientists to take advantage of the benefits of dynamic arrays on GPU, and if they are absolutely necessary they must devote a lot of effort to it's implementation. Usually a very specific implementation that's impossible to use on other applications and that can only be resized from the host.

A dynamic array is slower than a static one due to the work required for its operation, especially on a GPU where it is necessary to deal with thousands of parallel operations. In exchange it offers a better memory usage, due to its capacity to adjust its size to the amount of data contained. This facilitates the implementation of application where some data structures grow over time while others shrink, or programs where the size of two data structures is inversely related. This also allows to run more applications simultaneously in the GPU as long as the peak memory consumption doesn't occur at the same time.

This work explores the alternatives of using a static and semi-static arrays and implements a dynamically growable array for the GPU . The first step into the construction of a data structure with an interface similar to the C++ vector that works on a massively parallel architecture. This allows us to familiarize with the use of dynamic memory on the GPU and the limitations that this kind of architecture proposes for the implementation of a dynamic array. The dynamic structure is divided in two parts, first the insertion algorithms and second the data structure and allocation of memory. The data structure is inspired by the LFVector [6], the first implementation of a parallel dynamic array and also very important for all other research on the topic of parallel dynamic arrays. In this paper we present our solution which is divided by blocks, benefiting of GPU thread blocks and diminishing global synchronization issues, but bringing other drawbacks like slow access to its elements. We also compare it with other alternatives for arrays where elements are inserted over the run of a program, like a static solution where all needed memory is reserved from the start, and a semi-static approach where the array can be resized only from the host. The remaining sections cover related work, the different approaches, results and conclusions.

2 Related Work

While there are no attempts to implement a general dynamic array resizable from GPU device, there are implementations of arrays resizable from the host and plenty of investigation of parallel array for CPUs and of problems that deal with dynamic memory on the GPU.

2.1 GPU Resizable Array

The closest data structures to a dynamic array on a GPU device are offered by the libraries *trhust*[2] and *stdgpu*[17], but none of these tries to benefit of the architecture of the GPU. *Trhust* is a well known CUDA library that implements useful data structures to simplify CUDA programming. Among these structures are *host_vector* and *device_vector*, dynamic arrays that reside on the host and device memory respectively. But *device_vector* works like a doubling-array and its methods can only be called from the host. On the other hand *stdgpu* implements structures from the C++ STL in CUDA. In this case the vector implementation allows operations to be called from the device, *push_back* being one of them, but they are implemented with locks ignoring any possibly benefits from parallelization.

2.2 Parallel Dynamic Array

Lock-Free Vector (LFVector) [6] was the first proposed parallel dynamic array and the catalyst for further research on them. It proposes and idea similar to doubling arrays by duplicating the size each time more memory is needed. Differently from doubling-arrays it abandons the idea of storing an array contiguously, and divides it by buckets. This avoids the necessity of moving the elements to the new array when resizing and doesn't require as much synchronization between the distinct threads.

Further research on the topic include improvements on the LFVectors like Wait-Free Vector [7] or RCUArray [11] which uses the Read-Copy-Update mechanism. However, with each improvement these arrays relay more on synchronization methods for CPU architectures not applicable in GPU architectures.

2.3 GPU Synchronization

Global synchronization is usually avoided in the GPU, because of the overhead that it introduces. Unfortunately, it is not uncommon for it to be unavoidable. The simplest way to synchronize all treads is by dividing an algorithm in different kernels and using the host as a barrier, requiring the slow communication between the device and host. Therefore it is desired for global synchronization to occur inside the device.

Research on synchronization includes Fast Barrier Synchronization [20] and methods proposed for memory allocation [8] among others. The first work proposes two algorithm for inter-block synchronization. A lock-based method with

the use of atomic operations and a lock-free one, which dedicates one block of threads and global memory to indicate whether threads from other blocks are allowed to pass the barrier.

The second work focuses on memory allocation, which they separate in two stages. In the first stage, accounting the available resources, global synchronization is needed, for which they implement semaphores that allow concurrency in the critical section diminishing the principal bottleneck of semaphores. This paper also shows the importance of global synchronization when dealing with dynamic memory or memory allocators.

2.4 GPU Memory Allocator

Winter et al. (2021) [19] compares and evaluates various memory allocators for NVIDIA GPUs including the allocator provided by the CUDA-Toolkit and non-proprietary allocators starting from XMalloc [10] the first non-proprietary GPU up Ouroboros [18], one of the latest. Our work doesn't focus on the details from memory allocators, but it's important to keep them in consideration with their advantages and disadvantages. Also depending on the construction of the dynamic array, it could be benefited by different styles of memory allocators.

2.5 GPU Dynamic Applications

When a GPU application absolutely requires a dynamic array or similar solution it must be implemented, and usually these implementation are highly specific for the application they are built, for example when working with triangular meshes [9,13]. The first work introduces a general idea, using parallel prefix-sum to obtain the indexes at which each threads inserts an element. On the other hand, the second one instead of dealing with dynamic memory, introduces handles to each of the graphs elements in a way that modifying the handles offers a similar result to managing dynamic memory.

Given these works, it is clear that a vector-like structure is missing in GPU programming, with the capacity of being resized dynamically and taking advantage of parallelism and the GPU architecture. In this work, we focus on studying the growing aspect of this structure.

3 Proposing a Growable GPU Array

Our solution to construct a parallel growable array is divided in two parts. One that deals with the insertion of elements and updating the size of the array. And other that deals with the data structure to contain the elements and the memory allocation for resizing. These are presented below starting with the data structure.

3.1 Data Structure

For the data structure we decided to compare the proposed approach with other state of the art methods; the use of a static structure with all necessary memory pre-allocated from the start that only deals with insertion and a semi static structure that avoids dealing with dynamic memory inside the device by making all allocation through the host, which also acts as a barrier synchronization. Lastly we propose GGArray, a fully dynamic structure inside the GPU that always has the capacity to allocate more memory as needed within the hardware limits.

Static The static data structure consists of a simple C array allocated with *cudaMalloc* at the start of the program and using an insertion algorithm when elements need to be inserted from the device. This doesn't support any kind of resize and it's necessary to know the maximum possible size beforehand for it to not result in a segmentation fault.

Semi-static For the semi-static structure it is possible to use any sequential algorithm for the memory allocation since it must be done from the CPU. However, the API for low-level virtual memory management provided by CUDA [15] offers a lot of advantages over other structures. It allows to modify mappings between virtual and physical memory. Therefore instead of making a new array and moving the existing elements, it is possible to allocate only the desired extra memory and remap the virtual memory in such a way that the elements are contiguous in virtual memory even if they aren't physically contiguous.

Dynamic LFVector is based on the idea of Doubling-Arrays, where the size of the array is doubled whenever more space than its current capacity is needed. This is usually done by creating a new array with double the size and moving all elements of the previous array before deleting it. However when multiple threads are accessing the elements of the array at the same time, it is problematic to have two copies as it is necessary for all threads to know when the array changes. LFVector eliminates this problem by dividing the array in non-contiguous blocks, each double the size of the previous one, and allocating them when needed.

The original idea of the LFVector lies on the use of Compare-And-Swap (CAS) with every thread trying to allocate memory and deleting all except from the first allocated memory. This approach is not possible on GPU given that there is no enough memory for thousands of threads to try allocating it. Instead we tried busy waiting for the threads to synchronize when allocating memory. But this approach only works inside a block, where there is a high synchronization between threads. When trying to escalate further that, the execution scheduling starts to cause problem, because of the introduction of busy-waiting the vector isn't non-blocking and NVIDIA GPUs move to background blocks that are waiting for read/write operation. This provokes blocks that seem busy

to the GPU to replace the block in charge of allocating the memory and locks the execution.

There are two ways to solve this problem, the first is globally synchronizing all the blocks and the other is avoiding the synchronization by dividing the problem. We decided to further divide the array and take advantage of the thread division in blocks by creating multiple LFVectors, one for each block of threads as can be seen in Fig. 1. The diagram for the structure is shown in Fig. 2. This limits the parallelization of the problem since there is a fixed number of threads that can work on each block, but given the amount of cores in a GPU the bigger the array the less noticeable this is. This also allows us to synchronize LFVectors with builtin CUDA instructions. The operations are the same from the original LFVector except for the simplification of only supporting inserting elements as shown in Algorithm 1 and Algorithm 2.

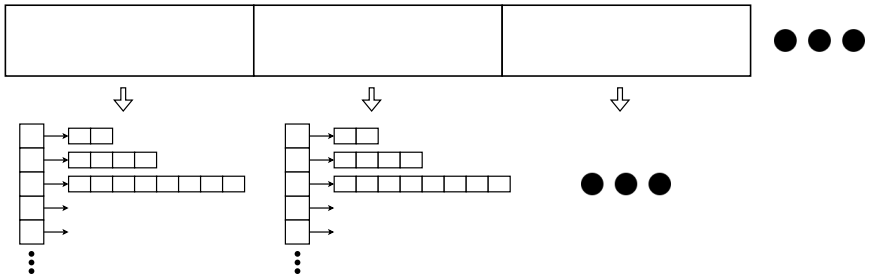

Fig. 1. GGArray, one LFVector per block.

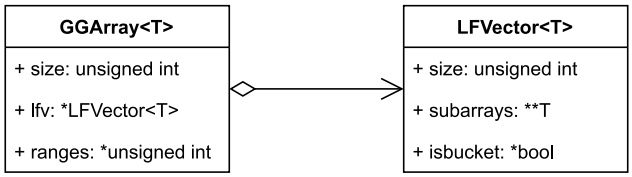

Fig. 2. GGArray structure, each LFVector maps to a GPU block and it's independent from other LFVectors.

Since each LFVector is constrained to its own block, the dynamic array requires a new structure to keep track of its size and the ranges encompassed by each LFVector. This structure is a prefix-sum of the sizes of all LFVectors, it contains the index of the first element contained by them. We are using a C-style array, which offer great amount of parallelism for updating its values, but it is

Algorithm 1 LFVector push_back

Require: e
  $idx = get\_insertion\_index()$
  $b = get\_bucket(idx)$
  if $bucket[b] = nil$ **then**
    $new\_bucket(b)$
  end if
  $synchronize()$
  $vector(idx) = e$

Algorithm 2 new_bucket for an LFVector with size of the first block = 32

Require: b
  if $not\ CAS(isbucket(b), False, True)$ **then**
    $bsize = 2^{log(first\_block\_size)+b}$
    $bucket[b] = malloc(bsize * type\_size)$
  end if

needed to search over this array to locate the LFVector that contains a certain index. Using a prefix-sum allows us to partially reduce the time needed for this search using binary search.

The insertion method is no longer called *push_back* since it is delegated to each of the vectors making it unsuitable for ordered arrays, but only needs to update the global size and update the prefix-sum of indexes because the actual insertion is taken care locally by the LFVectors.

3.2 Insertion

The main objective of the insertions algorithms is to update the size of the array while giving each inserting thread a unique index greater than the previous size and less than the subsequent size, such that each thread inserts its element in a different position maintaining the appearance of a contiguous array.

Atomic The simplest way of obtaining a unique position for each new element is to use the CUDA instruction *atomicAdd*, which takes as parameters a memory address and an addend. It returns the value stored in the address and updates its value by adding the addend. For the insertion algorithm each inserting threads adds 1 to the size of the array, obtaining an index where to insert the element and updating the size of the array.

Prefix-sum Another more efficient algorithm for inserting elements consist of considering numbers of insertion per thread as an array with 0s or 1s depending if the threads need to insert an element and calculating the prefix-sum of this array. We implemented this algorithm locally with the warp *__shfl_up_sync* instruction and globally with atomic operations.

Tensor-cores prefix-sum As demonstrated by Dakkak et al. (2019) [5] its possible to accelerate the prefix-sum computation with CUDA tensor cores by representing the problem as matrices multiplication. Although this approach works better with denser problems that have more elements than threads and benefit from the added workforce, we think this is a good opportunity to test the possible applications of tensor cores outside machine learning.

4 Theoretical Memory Usage

One of the most important benefits of the GGArray is its ability to dynamically grow from kernel code according to the needs of the program. This allows programmers to run applications without concern about the amount of memory to pre-allocate nor if the program will fail due to an invalid memory address. This isn't a big issue for static methods when it is known beforehand the insertion behaviour of each thread, since with an statistical analysis it is possible to determine the biggest amount of memory needed with a very small probability of failing and with big enough amounts of data and often it will be less than double the expected memory usage.

However, when there isn't enough information or there is only a rough idea of the growing behaviour of the array, the worst case for the static or semi-static methods start to grow excessively. Fig 3 shows the memory needed for an example where the amount of insertions are given by the size of the array times a factor given by a log-normal distribution with parameters $\mu = 0$ and $\sigma \in [0, 2]$. It shows how with a bigger standard deviation and uncertainty about the amount of insertions realized, the more memory it is needed for the static method to fail only 1% of the times it is executed. While GGArray, only needs in the worst case approximately double of the expected outcome.

5 Experimental Performance Evaluation

All performance tests were ran on two NVIDIA GPUs, Titan RTX and A100. The specifications of these GPUs is shown in Table 1.

	TITAN RTX	A100
CUDA Cores	4608	6912
Tensor cores	576	432
Memory	24 GB	40 GB
FP16 performance	32.62 TFLOPS	77.97 TFLOPS
FP32 performance	16.31 TFLOPS	19.49 TFLOPS
Clock Speed	1350 MHz	765 MHz

Table 1. GPUs specifications

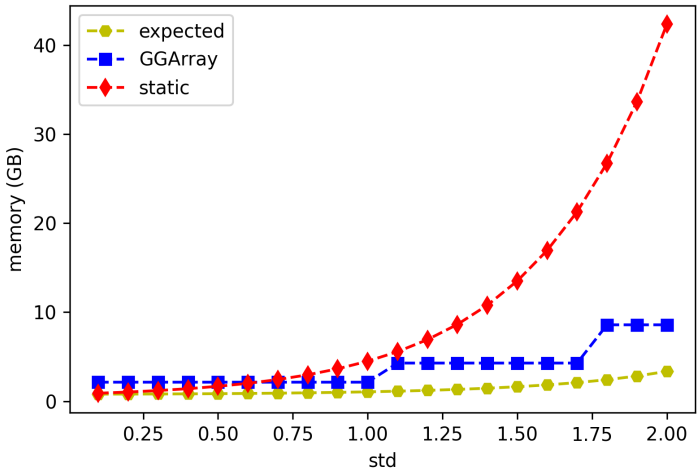

Fig. 3. Theoretic memory usage of our structure and static/semi-static array.

5.1 Insertion Algorithms

For testing the performance of the different insertion algorithms only the static array was used. The reason for not testing in other structures is that insertion algorithms are independent of the underlying structure and the static array is the simplest, allowing to only measure the time of the insertion algorithm without being affected by the time needed to access the structure elements. The test consist of an array with $1e6$ elements and a sufficient capacity for duplicating its size 10 times, finishing with an array of 1.024e9 elements, and time measurements of each iteration of duplication. Fig. 4 contains the results obtained for the algorithms using only atomic operations, and scan with two implementations (one using warp shuffling and another with tensor cores). The slowest is the one that only uses atomic operations while the shuffle scan is the fastest closely followed by the tensor core implementation.

Regarding the scan operation being slower when implemented with tensor cores than with the usual algorithms, as opposite from other studies in the state of the art, it's due to not meeting the necessary workload for this specific case. The size of the problem for the insertion algorithm is the amount of threads participating in the insertion. Thus, when using tensor cores that multiplies 16x16 matrices per warp, there are not enough elements to fill all matrices from all warps. In the tensor scan algorithm only one eighth of the warps are realizing the algorithm while the rest are idle, not taking advantage of the full potential of tensor cores. It's also important to note that the difference between the two scan versions is lower in the A100 GPU. This is due to the bigger improvement

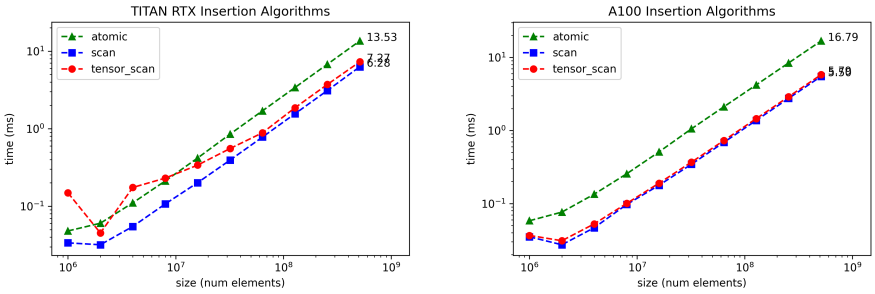

Fig. 4. Time for insertion algorithms over size.

in tensor cores from the previous generations than the improvement in CUDA cores.

5.2 Number of LFVectors

The variables that affect time execution of the GGArray are its size, the amount of blocks in which it is divided and the amount of memory allocations previously realized. The size impacts read/write and insertion operations since the more elements the array contains, more operations are needed to operate over the whole array. Similarly more concurrent blocks allows a bigger amount of parallelization in these operations, except for atomic operations. In the case of memory allocation more parallelization means more allocations that can't be parallelized due to limitations of current technology.

Since the amount of blocks used to divide the array has benefit and demerits in distinct operations first we ran tests to determine the optimal amount of blocks shown in Fig. 5. The figure shows the amount of time it takes to duplicate the amount of elements in the array utilizing different numbers of blocks. The duplication process includes the memory allocation and insertion of elements. The figure also shows the time spent to realize read/write operations in two ways. The first one (rw_g) utilizes the structure as if it were an array with one thread per element. On the other hand, rw_b follows the block structure and uses one GPU block per array block avoiding the process of determining which block contains an element, which is quite slow. In general, a low number of blocks implies the growth of the structure is slower due to the lack of parallelization in insertion and the figure show two minimums with 32 and 512 blocks. With over 32 blocks, read/write operations by block are faster and their time is inversely related to the number of blocks.

5.3 Growable Array Operations

The experiment to test the performance of array operations consists of starting with an array of size $1e6$ and duplicating (with scan algorithm) its size 10 times.

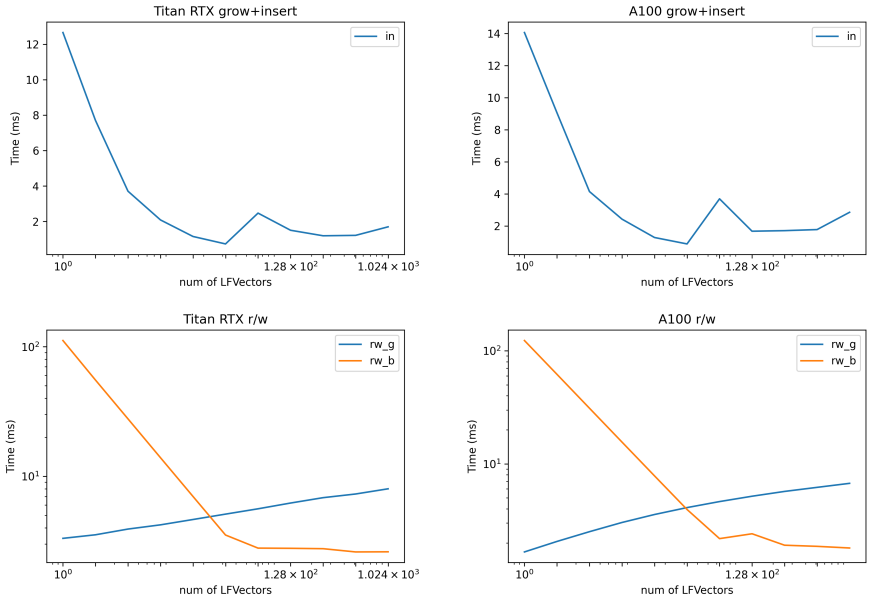

Fig. 5. Time over number of LFVectors

Inserting less elements than the size of the array doesn't reduce the time taken, because even threads that don't inert elements play a role in the insertion algorithm and are also needed for synchronization. The duplication of the array is divided in the grow operation and the insertion operation. Also for each size the time to operate on each of its elements is measured. The results are displayed in Fig. 6 and Table 2 shows the exact time taken by each operation on the last iteration. In accordance with the previous results 32 and 512 blocks are utilized and read/write operations per block. In the legend 'GGArrayXX' correspond to our proposed structure with the numbers of blocks in which it is divided, and 'memMap' to the semi-static array using the NVIDIA low-level memory management API. The first two figures show the time to duplicate the capacity of the arrays. The two in the middle depict the time needed for the insertion of elements filling the capacity of the array. And the last two plots display the time required to realize operations in all elements of the array. The operation used correspond to a kernel that adds 1 thirty times to each element.

It draws attention that the third resize barely takes time. The explanation is that the growth in capacity of the GGArray is not a constant factor, but it tends to two as the size increases, in this case no resizing took place because the capacity from the previous iteration was enough. The biggest problem of the proposed structure are the slow read/write operations. While allocating memory for a large amount of LFVectors and inserting elements are slower than the other structures the difference it's not big enough to cause a bottleneck, especially

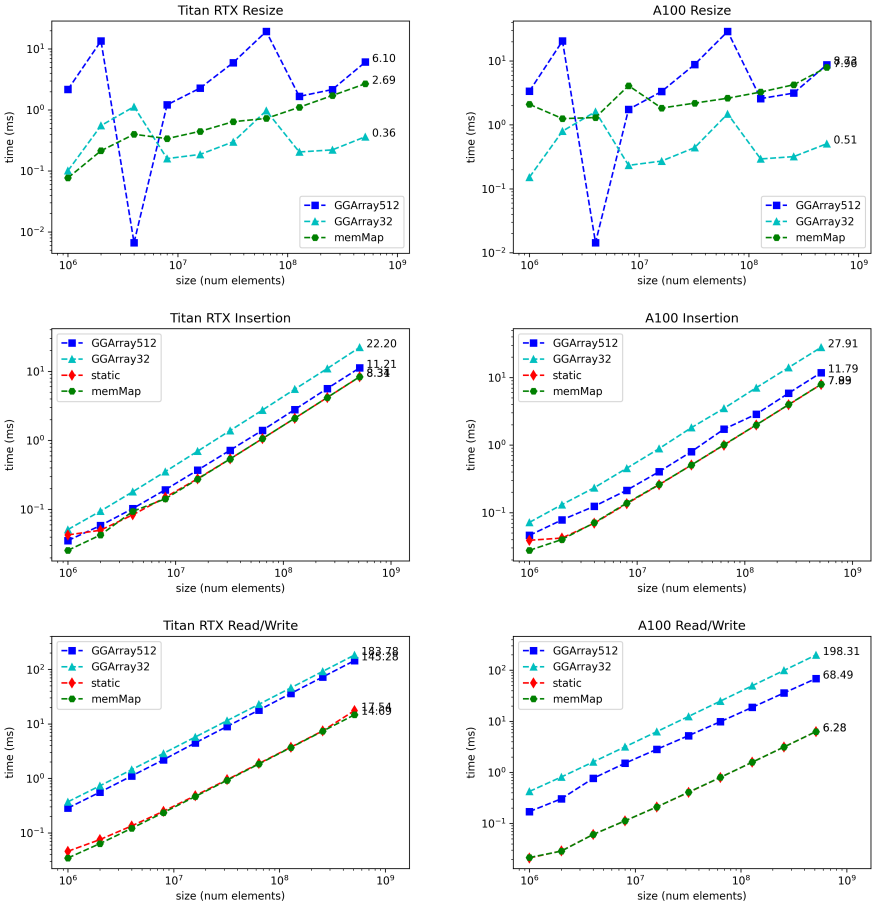

Fig. 6. Time of operations to duplicate array size each iteration

Table 2. Time (ms) of operations to duplicate array size (5.12*e*8) in the last iteration using NVIDIA A100

	grow	insert	read/write
static	—	7.07	6.27
memMap	5.21	7.87	6.28
GGArray512	8.76	11.79	69.73
GGArray32	0.52	27.90	198.32

when realizing more complex operations in-between resizing. However, to realize work with the contents of the structure it's absolutely necessary to read and write its elements. And this operations are extremely slow, more than 10 times slower, even when working by block without the need to search which LFVector contains each element. This is produced by a more complex indexing operation, a worse cache locality and the need to pass over multiple pointers to reach an element. Something that may only be resolved by a truly contiguous array, but this introduces several synchronization issues.

Still, there are some applications that my benefit from this structure. For applications that need a dynamic array and that don't have a way to confidently know beforehand the maximum size, or the uncertainty of the maximum size is big enough, our structure offers a way to dynamically grow the array from inside the kernel and using no more than double the necessary memory. Also applications that can be defined in phases where one phase only inserts elements and the other phases realize work on a static structure could be benefited by moving the elements between our structure and a static array. This reduces the read/write operations to only a few per each growth phase and can still take advantage of the characteristics of static arrays in work phases, although now the worst case for memory usage is three times the optimal. Applications that meet these conditions may be encountered in computer geometry and triangular mesh refinement.

5.4 Example Use Case

Following the above, using a dummy example where the GGArray512 was used for the insertion phases and the semi-static structure for the work phase. The work phase simply consisted of a kernel that adds 1 to every element of the array called multiple times (between 1 and 1000) corresponding to the x axis of Fig. 7. The figure indicates the speedup of this approach over using only the semi-static structure. It is not a case that favours the GGArray in terms of memory, but it illustrates how the overhead added by the dynamic structure can be disregarded when the amount of work on the other phases is big enough. The experiment was designed with 5 repetitions and a starting array size such that after all iterations and independent of the amount per insertion per thread per iteration the final size is $1e9$. Inserting 1, 3, or 10 times the size of the array each iteration doesn't have an impact on the speedup.

6 Conclusions and Future Work

In this work we proposed a fully dynamic array for the GPU that offers the interface of a growable array and works inside the GPU without the need of synchronizing through the host. This allows to allocate memory when needed from kernel code without the possibility to pre-allocate all necessary memory. However the structure has an important drawback, its slow access to the elements, which makes it unsuitable as a general purpose array. But there are certain applications

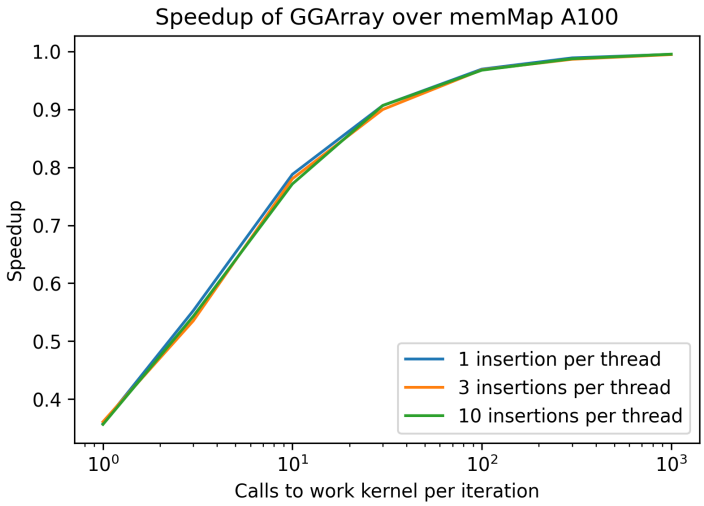

Fig. 7. Speedup in example use case in application divided in phases

for which it can provide benefits, especially applications where dynamic allocation is crucial and applications that can be divided in phases where most work is static and the insertion can be done separately. Also it is important to note that this structure affects the programming of CUDA kernels due to working by blocks and requiring all threads to stay in execution for warp synchronization and insertion algorithms.

Further improvements are needed for accessing elements faster, one idea is to use shared memory to cache segments of the array. Also, NVIDIA has made progress that favours dynamic applications in the lasts years, for example RT cores, and it is being researched to use them outside ray tracing. This may be an option to implement a dynamic data structure. On the other hand the issue of accessing elements doesn't exist if a contiguous array is utilized, although it brings a lot of synchronization issues, that may could be solved with cooperative groups. NVIDIA has also recently unlocked the GPU System Processor, a chip similar to a CPU inside the GPU. It could bring a lot of benefits if it is used for synchronization instead of the CPU. And there are improvements that could be made combining the two different approaches, for example, a new feature called thread block cluster, that allows to cluster threads in groups bigger than a thread block. Finally, separating the data structure and allocation from the insertion algorithm leaves open the possibilities for the use of any scan algorithm already studied or even other algorithms that outputs an unique index per thread.

Acknowledgments

This research was supported by the Temporal research group, the ANID Fondecyt grant #1221357 and the Patagón supercomputer of Universidad Austral de Chile (FONDEQUIP EQM180042).

References

1. Muhammad A. Awad, Saman Ashkiani, Serban D. Porumbescu, and John Douglas Owens. Dynamic graphs on the gpu. *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 739–748, 2020.
2. Nathan Bell and Jared Hoberock. Chapter 26 - thrust: A productivity-oriented library for cuda. In Wen mei W. Hwu, editor, *GPU Computing Gems Jade Edition*, Applications of GPU Computing Series, pages 359–371. Morgan Kaufmann, Boston, 2012.
3. Federico Busato, Oded Green, Nicola Bombieri, and David A. Bader. Hornet: An efficient data structure for dynamic sparse graphs and matrices on GPUs. *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–7, 2018.
4. NVIDIA CORPORATION. CUDA C++ best practices guide 11.6.1, 2022.
5. Abdul Dakkak, Cheng Li, Isaac Gelado, Jinjun Xiong, and Wen mei W. Hwu. Accelerating reduction and scan using tensor core units. *Proceedings of the ACM International Conference on Supercomputing*, 2019.
6. Damian Dechev, Peter Pirkelbauer, and Bjarne Stroustrup. Lock-free dynamically resizable arrays. In *OPODIS*, 2006.
7. Steven D. Feldman, Carlos Valera-Leon, and Damian Dechev. An efficient wait-free vector. *IEEE Transactions on Parallel and Distributed Systems*, 27:654–667, 2016.
8. Isaac Gelado and Michael Garland. Throughput-oriented GPU memory allocation. *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, 2019.
9. Bilal Hatipoglu and Can C. Özturan. Parallel triangular mesh refinement by longest edge bisection. *SIAM J. Sci. Comput.*, 37, 1997.
10. Xiaohuang Huang, Christopher I. Rodrigues, Stephen Jones, Ian Buck, and Wen mei W. Hwu. Xmalloc: A scalable lock-free dynamic memory allocator for many-core machines. *2010 10th IEEE International Conference on Computer and Information Technology*, pages 1134–1139, 2010.
11. Louis Jenkins. RCUArray: An RCU-like parallel-safe distributed resizable array. *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 925–933, 2018.
12. James King, Thomas Gilray, Robert Michael Kirby, and Matthew Might. Dynamic sparse-matrix allocation on gpus. In *ISC*, 2016.
13. Mohamed-H. Mousa and M. Hussein. High-performance simplification of triangular surfaces using a gpu. *PloS one*, 16 8:e0255832, 2021.
14. Cristóbal A. Navarro, Nancy Hitschfeld-Kahler, and Luis Mateu. A survey on parallel computing and its applications in data-parallel problems using GPU architectures. *Communications in Computational Physics*, 15:285–329, 2014.
15. Cory Perry and Nikolay Sakharnykh. Introducing low-level gpu virtual memory management. `https://developer.nvidia.com/blog/introducing-low-level-gpu-virtual-memory-management/`, 2020.

16. Mo Sha, Yuchen Li, Bingsheng He, and Kian-Lee Tan. Accelerating dynamic graph analytics on gpus. *Proc. VLDB Endow.*, 11:107–120, 2017.
17. Patrick Stotko. stdgpu: Efficient stl-like data structures on the gpu. *ArXiv*, abs/1908.05936, 2019.
18. Martin Winter, Daniel Mlakar, Mathias Parger, and Markus Steinberger. Ouroboros: virtualized queues for dynamic memory management on GPUs. *Proceedings of the 34th ACM International Conference on Supercomputing*, 2020.
19. Martin Winter, Mathias Parger, Daniel Mlakar, and Markus Steinberger. Are dynamic memory managers on GPUs slow?: a survey and benchmarks. *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2021.
20. Shucai Xiao and Wu chun Feng. Inter-block GPU communication via fast barrier synchronization. *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–12, 2010.