Design and Implementation of Parallel List Data Structure using Graphics Accelerators

Tamás Varga
John von Neumann
Faculty of Informatics
Óbuda University
Budapest, Hungary
Email: tomi94521@gmail.com

Sándor Szénási
John von Neumann
Faculty of Informatics
Óbuda University
Budapest, Hungary
Email: szenasi.sandor@nik.uni-obuda.hu

***Abstract*—One of the biggest shortcomings of the CUDA environment is the lack of a data structure having effective parallel insert and remove operations. This paper focuses on the insertion. The traditional vector-based lists are not applicable for this kind of addition; however, the well-known linked list data structure has the ability to handle parallel insertions at the same time. But nowadays, there is not any "official" linked list data structure in the current CUDA runtime library. This paper presents a novel way to create a multi-layered linked list (called Parallel List) using the NVIDIA CUDA framework. As our experiments show, the implemented data structure is able to do multiple insert operations 2–30 times faster than the traditional sequential CPU implementation of a general list object. Another advantage of the new data structure is the support of Random Access pattern to access contents directly.**

I. Introduction

First, to demonstrate the problem of parallel insertion, let's see the following example. We have a list a containing the following numerical values (Eq. 1):

$$a = (1, 3, 1, 4, 5, 2) \tag{1}$$

As an example for parallel insertion, we have to insert numbers (the new value is 9) before all items where the following condition (predicate) is true: the content of the given item is greater than 2. After the operation, the result is the following list a' (Eq. 2):

$$a' = (1, 9, 3, 1, 9, 4, 9, 5, 2) \tag{2}$$

In the case of traditional vector-based lists, the insert operation is very resource intensive. To insert a new element into the middle of the list (if we have to keep the order of the already existing elements), it is necessary to shift all further elements one step forward. In the case of remove operation, the problem is very similar, but we have to shift to the opposite direction.

We measured the runtime performance of the C# List and the Thrust vector data structures from this point of view [1]. As a preliminary test, we created a list/vector using 1,000,000 elements and inserted new items before each of the already existing elements in the data structure (to simulate the worst case scenario). These operations lasted for 16 minutes (C# implementation using a Core i5 processor) and 22 minutes (Thrust implementation [2] using an Nvidia GTX750ti graphics card). Based on these preliminary results, it is clear that the traditional vector based list is not applicable for our purposes to implement mass insertion.

Linked lists are more adequate to this purpose [3]–[6]. It is possible to enumerate the linked list only once and do the necessary changes locally in the immediate neighbourhoods of the affected items. We ran the same test using linked lists, and the execution needs only 100-150ms (C# implementation) and 4 seconds (C++ implementation). These are significantly better results, so it is easy to insert new items into linked lists. The only issue is the high time requirement of the linked list enumeration.

The goal of our project is to design and implement a novel parallel linked list data structure in the memory of a graphics accelerator to parallelise the enumeration of the list to decrease the runtime of the insert operation.

The rest of this paper is structured as follows: Section 2 contains the detailed description of the new linked list; Section 3 presents the results of our experiments (speed comparison); and finally, the last section contains the conclusions and our further plans.

II. Methodology

Linked lists are usually implemented as dynamic data structures: the list allocates and deallocates memory in runtime according to the number of currently stored items. However, the CUDA framework [7] gives the opportunity to allocate GPU memory with kernel-inside malloc() function calls [8], but this leads to poor performance and unacceptable runtime in the case of a high number of stored items.

There are some (more-or-less stable) third-party solutions [9] for more efficient memory allocations, but these are not discussed in this paper. Our solution is based on a multi-layer data structure which eliminates this bottleneck. This section presents the three layers of this Parallel List (PL) structure.

A. Layers and memory regions

The memory consumption of one item is as follows: 4 byte *data* field, 4 byte *next* field in the first (bottom) layer; 4

978-1-5090-1216-9/16/$31.00 ©2016 IEEE

byte *pointer* and 4 byte *distance* field in the second layer; an additional 4 byte *pointer* in the third (top) layer. The data structure also needs some *boolean* variables; therefore, the memory consumption of the whole package is $N \times 24$ byte (where N is the number of the currently stored items in the list). This is fairly a lot, but the cost of the parallelization is the high memory demand.

The third layer is the reading layer, based on the followings:

$$Layer_3[index - Layer_2[index].distance] = \&Layer_2[index] \quad (3)$$

According to the above, the presented data structure can access the elements randomly. During the operations, the pointers of the bottom and middle layer and the already mentioned distance values are changing. Finally, the values of the third layer change.

This third layer is designed to help the parallel insert method and to make some monitoring process available for further research purposes.

B. Levels of the data structure

The relation between layers and levels ensures the availability of parallel operations to a PL. Different lists are distinguishable at the highest level. As a further development, we would like to create a particular list variant storing string content using the already existing framework.

The level hierarchy also ensures that not only the nodes can be the targets of the different operations, but two or more PLs also can act as nodes.

- Level 1 – contents of the Parallel List Container (PLC)
  The PLC contains the followings: *HeadList*, *LastList*, *ListCounter*, *ListNextFree*, *ContainerSize*. The memory regions of the necessary layers of the PLC are allocated based on the *ContainerSize* value.
  - Layer 1 – Bottom layer of the PLC
    - *Next* pointers of the list.
  - Layer 2 – Middle layer of the PLC
    - *Pointers to the next fields* of the lists.
    - *Distance* data.
  - Layer 3 – Top layer of the PLC
    - Pointers pointing to an *address* of the second layer calculated by the distance and index values.
    - This layer ensures that we can access any elements of the PLC.
- Level 2 – contents of the Parallel List (PL)
  The PL contains the followings: *HeadNode*, *LastNode*, *NodeCounter*, *NodeNextFree*, *ListSize*.
  The memory allocation of the three presented layers is based on the *ListSize* value.
  - Layer 1 – Bottom layer of the PL
    - Nodes (the actual *data* stored by the list).
    - The *next* pointers.
  - Layer 2 - Middle layer of the PL
    - Pointers to the *next* fields
    - Distance value.
  - Layer 3 – Top layer of the PL
    - Pointers pointing to an *address* of the second layer calculated by the distance and index values.
    - This layer ensured that we can access any elements of the PL.

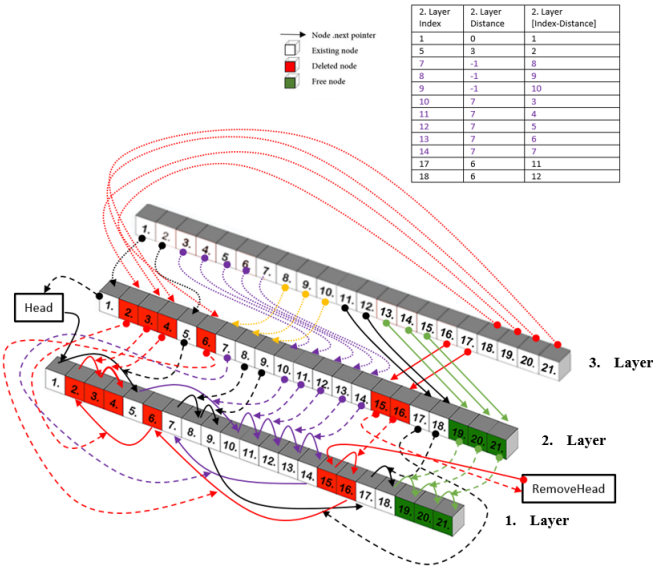

Fig. 1: The three layers of the data structure after basic operations.

C. Reusing deleted items

Deleted items of the list in Fig. 1 are 2-3-4-5-15-16. These fields are based on the "Index based remove" method which does not need a preliminary scan method. The question is that how can we access these memory regions without reallocation (free and allocate memory regions again)?

After the deletion, we attach them to the end of the third layer. This is an additional process, which slows down the delete operation, but this effect is not significant. However, thanks to this, we can use all allocated space without any reallocation operations.

D. Parallel insertion

The main goal of the parallel insert operation can be one of the followings [10]:

- Insert a constant number of elements.
- Insert a variable number of elements.

This method needs a particular scan procedure [11]. The algorithm uses the result of this scan execution [12] to calculate the distance values according to the new item contents.

As a short overview, Fig. 2 shows an example for the "Insert variable number of elements" case. The figure shows only the first and second layers.

The main steps of the simpler method ("insert constant number of elements") are the followings:

1) First kernel execution

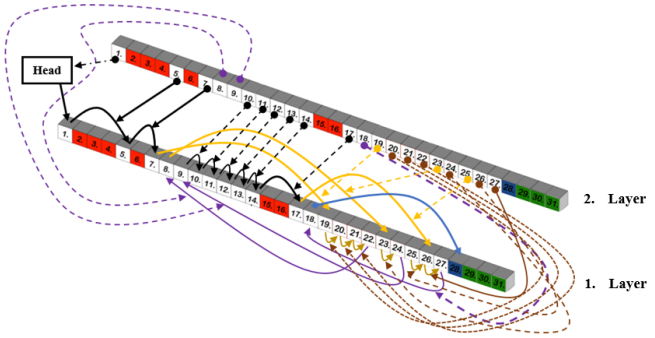

Fig. 2: The first and second layers of the data structure after a parallel insertion.

The kernel counts the number of affected elements based on the predicate given by the caller. It also changes the distance data of the items based on the result of the scan method.

2) Second kernel execution
The second kernel execution contains the following steps:
   a) The new values are stored in the first layer using the $newindex$ position.
   b) The new next pointers of this layer will reference to the memory addresses where the given predicate is true (the addresses of the first layer referenced by the third layer).
   c) The new next pointers of the first layer reference to the new positions of the second layer.
   d) The new distance data of the second layer is calculated by the new index value and the result of the scan operation.
   e) The next pointers of the first layer (according to the $index - 1$ item in the third layer) reference to the memory address of the $newindex$ items of the first layer.

   As a special case, sometimes it is necessary to change the head pointer too.
3) Third kernel execution
   Reconfigures all items of the third layer based on the index values and the distance data of the second layer.

III. Evaluation

We have ran several tests to compare the runtime of the insert operation in the case of the new PL data structure (implemented using the GPU) and original sequential algorithm (implemented as a C# code for the CPU).

We used the following hardware configurations for the tests:

- "CPU" implementation
  - Processor: Core i5-3470 3.2GHz
  - Memory: 8GB
  - Compiler: Microsoft C#
- "GPU" implementation
  - Processor: NVIDIA GTX750Ti
  - Memory: 2GB
  - Compiler: CUDA C

TABLE I: Average runtime of inserting multiple items into PL. Initial list size is 1,000,000 elements.

#	Number of inserted items	GPU runtime (ms)	CPU runtime (ms)
1	1 - 50000	4.262	7
2	50000 - 75000	4.616	9
3	75000 - 130000	4.669	10
4	130000 - 222500	4.848	14
5	222500 - 489500	5.725	57
6	484950 - 747000	6.055	102
7	747000 - 921800	6.981	152
8	921800 - 1000000	7.706	210

We used NVIDIA Visual Profiler 7 to measure the runtime of different operations.

It can be suspected that the PL is efficient only in the case of large data sets. In our runtime tests, we created a linked list of 1,000,000 elements in the first phase and another linked list of 10,000,000 elements in the second phase.

The methodology was the same in both cases:

1) Create two linked lists in the CPU and in the GPU memory.
2) Insert 1M/10M random values between a given interval to the list.
3) Start timer.
4) Execute the sequential/parallel insert operations using a given predicate (insert a given value after all elements corresponding to this statement).
5) Stop timer.
6) Free all previously allocated memory.
7) Store timer value.

As is visible in Table I, the GPU was significantly faster in all cases. It is worth noting that the speed-up heavily depends on the number of inserted items (this number is based on the given predicate and the contents of the already existing elements). It takes significantly more time for the CPU to execute the insertion of a large number of items. As is visible, the GPU can perform these operations more efficient, thanks to its parallel architecture. Fig. 3 shows the same values as a bar chart.

The results of the second test are very similar (see Table II for the details and Fig. 4 for a quick overview). The number of randomly generated initial items in the linked list is 10 times more than in the previous test. It is visible from the detailed data, that the runtime values are linearly dependent from this. It takes about 10 times more for the CPU to insert the given items into the 10 times larger linked list.

The GPU code was always faster; therefore, the same can be concluded as in the previous test.

IV. Conclusions

The goal of this research was the development of a novel linked list like data structure (called Parallel List) which is able to execute parallel insert and delete operations efficiently. As

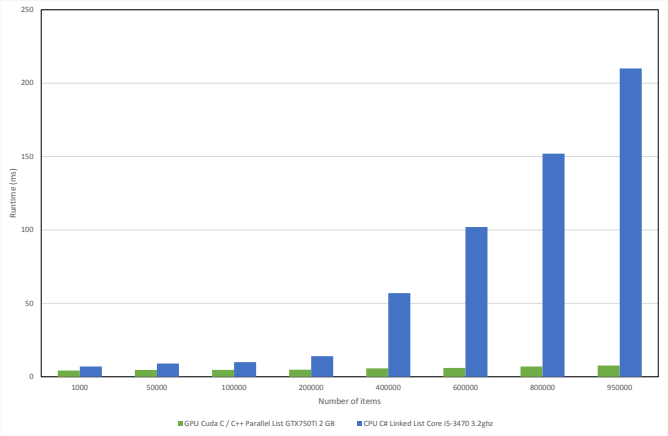

Fig. 3: Average runtime of inserting multiple items into PL. Initial list size is 1,000,000 elements.

TABLE II: Average runtime of inserting multiple items into PL. Initial list size is 10,000,000 elements.

#	Number of inserted items	CPU runtime (ms)	GPU runtime (ms)
1	10000	70	36.383
2	100000	70	36.638
3	500000	160	38.096
4	1000000	216	39.27
5	2000000	421	42.053
6	4000000	847	46.607
7	6000000	1356	53.486
8	8000000	2113	60.035

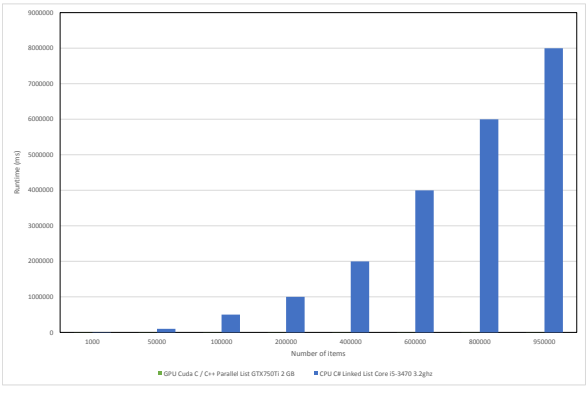

Fig. 4: Average runtime of inserting multiple items into PL. Initial list size is 10,000,000 elements.

the results presented in the evaluation section show we were able to design and implement this kind of list.

The insertion is about 2–30 times faster than the original sequential algorithm (the speed-up depends on the number of new items and the actual content of the list), and our preliminary tests show that the parallel implementation of the delete operation has a similar potential.

It is worth noting that the required storage space is significantly higher in the case of PL, this is the cost of the efficient modification operations.

As a further plan, we would like to implement a "kernel-inside" data insertion method. Recent GPU architectures support kernel launching inside another kernel. Using this option we can speed-up the already existing data insertion method.

We also would like to develop and implement a special relationship between multiple Parallel Lists, called "strong relation". In this case, these lists have some information about each other, and this can lead to more efficient operations.

Acknowledgements

The authors would like to thank NVIDIA Corporation for providing graphics hardware for the GPU benchmarks through the GPU Education Center (formerly known as CUDA Teaching Center) program. The authors also wish to thank the members of the Center at Óbuda University for their constructive comments and suggestions. The authors would like to say thanks for the help of the Hungarian National Talent Program (NTP-HHTDK-0015-0036).

References

[1] Á. Beszédes, "Global Dynamic Slicing for the C Language," *Acta Polytechnica Hungarica*, vol. 12, no. 1, pp. 2015–117, 2014.

[2] N. Bell and J. Hoberock, "Thrust: A Productivity-Oriented Library for CUDA," in *GPU Computing Gems Jade Edition*, 2012, pp. 359–371.

[3] T. H. Cormen, *Introduction to algorithms*. MIT Press, 2009.

[4] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The design and analysis of computer algorithms*. Addison-Wesley Pub. Co, 1974.

[5] T. B. Jablin, J. A. Jablin, P. Prabhu, F. Liu, and D. I. August, "Dynamically Managed Data for CPU-GPU Architectures," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, 2012, pp. 165–174.

[6] Z. Wei and J. JaJa, "Optimization of linked list prefix computations on multithreaded GPUs using CUDA," *Proceedings of the 2010 IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010*, 2010.

[7] NVIDIA, "CUDA C Programming Guide," 2014.

[8] ——, "Whitepaper. NVIDIA's Next Generation CUDATM Compute Architecture: Kepler TM GK110," 2012.

[9] M. Steinberger, M. Kenzel, B. Kainz, and D. Schmalstieg, "ScatterAlloc: Massively parallel dynamic memory allocation for the GPU," in *2012 Innovative Parallel Computing, InPar 2012*, 2012.

[10] G. Kövesdán, M. Asztalos, and L. Lengyel, "Architectural design patterns for language parsers," *Acta Polytechnica Hungarica*, vol. 11, no. 5, pp. 39–57, 2014.

[11] M. Harris, S. Sengupta, and J. D. Owens, "Parallel Prefix Sum (Scan) with CUDA Mark," *Gpu gems 3*, no. April, pp. 1–24, 2007. [Online]. Available: http://dl.acm.org/citation.cfm?id=1407436

[12] G. E. Blelloch, "Prefix Sums and Their Applications," *Computer*, pp. 35–60, 1990. [Online]. Available: http://www.cs.cmu.edu/{~}blelloch/papers/Ble93.pdf