1-1-2022

Scan Stack: A Search-based Concurrent Stack for GPU

Noah Brennen South

SCAN STACK: A SEARCH-BASED CONCURRENT STACK FOR GPU

A Thesis
presented in partial fulfillment of requirements
for the degree of Masters of Science
in the Department of Computer and Information Science
The University of Mississippi

by

Noah South

December 2022

ABSTRACT

Concurrent data structures play a critical role in the overall performance of GPGPU applications. Stack is one of the basic data structures and finds numerous applications where data is processed in a Last In First Out (LIFO) fashion. Although concurrent stack is well researched for multi-core CPUs, there is little research pointing to the conversion of CPU stacks into a GPU-friendly form. In this paper, we propose a concurrent search-based GPU stack named "Scan Stack." The proposed stack is designed to take advantage of GPU memory access patterns, memory coalescence, and thread structures (i.e., warps) to increase throughput. Our experiments on an NVIDIA RTX 3090 shows that our proposed scan stack significantly improves the throughput and scalability for all benchmarks when reducing the search area. However, the greatest improvements are shown when elimination is possible, and this improvement reaches nearly 39 times what a non-optimized structure is capable of.

TABLE OF CONTENTS

ABSTRACT ii

LIST OF FIGURES iv

Introduction 1

Related Work 3

Design and Implementation 6

3.1 Base Structure 6

3.2 Stack Operations 8

3.3 Contention Handling 10

Experimental Results 16

4.1 Correctness 19

Conclusion 22

BIBLIOGRAPHY 23

VITA . . . . .

LIST OF FIGURES

3.1 An example of the Scan stack storing integer values. The empty cells are designated by a value of -1. 6

3.2 (a) A sample stack with an example top location shown. (b) Shows how the lowest necessary area scans at intervals of X to find where an empty cell is. This empty cell shows that the scan has passed where top could be located. In (c), an empty cell is observed at the 8th interval, new pushes will then begin at the start 7th interval (i.e., below the top) while pops begin from the 9th interval (i.e., above the top). 7

3.3 Push operation attempting to insert the value 16 into the stack. The stack's top is found when an empty cell (-1 integer value) is encountered. Once the top is found, an atomicCAS() is performed to exchange the insertion value (i.e., 16) with the present one (i.e., -1). 8

3.4 (a) Push operation has failed its task to atomicCAS() 16 into an empty cell as the cell is no longer empty (b). In (c), the thread regresses to 36's cell finding a non-empty value, so it proceeds to move forward in the stack once more until an empty cell is found (d). 10

3.5 Pop operation attempting to remove the top integer value from the stack. Once the top is identified, an atomicCAS() is performed to extract the value. 10

3.6 (a) Pop operation that has failed its task to atomicCAS() top and create an invalid cell. This is due to another thread completing (b). In (c), the thread climbs to view an empty cell (i.e., -1) showing it is above the stack before scanning continues down the stack. An invalid cell (i.e., -2) is observed and re-validated (d). (e) After renewing the invalid cell, the operation climbs the stack again repeating the process in (c). (f) the operation succeeds in creating the invalid cell and returning the value from the cell. 12

3.7 Multiple threads attempting an operation upon the top of the stack. Operations are numbered in order of occurrence. Pushes then pops 13

3.8 Multiple threads attempting an operation upon the top of the stack. Operations are numbered in order of occurrence. Mixed pushes and pops 14

4.1 Throughput and execution time of randomly chosen operation benchmark under various optimizations (No Opt - No-Optimizations, LA - Lowest-Area, EL+LA - Elimination-LA, and EL+LA+PX - Elimination-Proxy) 17

4.2 Throughput and Time of only-pushes benchmark with optimizations. No Opt - No-Optimizations, LA - Lowest-Area, EL+LA - Elimination-LA, and EL+LA+PX - Elimination-Proxy 18

4.3 Throughput and Time of only-pop benchmark with optimizations. No Opt - No-Optimizations, LA - Lowest-Area, EL+LA - Elimination-LA, and EL+LA+PX - Elimination-Proxy 18

4.4 Throughput and Time of only-pop benchmark with optimizations, without invalidation. (No Opt - No-Optimizations, LA - Lowest-Area, EL+LA - Elimination-LA, and EL+LA+PX - Elimination-Proxy). Copy of Figure 4.2 provided for reference above. 21

Chapter 1

Introduction

Data parallelism has grown to become a very significant topic in the field of computer science. As more data is gathered, it becomes harder to process in a serial manner while retaining efficiency. Considering the topic of data structures, CPUs resolve the desired increase in work and efficiency by delegating the work across its cores for concurrent progress. However, CPUs have an inherent limitation due to the number of cores. GPUs, on the other hand, are built specifically to process large amounts of data across thousands of dedicated cores and threads.

For this reason, the application of GPU programming becomes more prevalent as big data expands. This has led previously CPU-based programs and structures to expand by facilitating a GPU implementation. While GPUs have become more popular with their capabilities of massively parallelized operations, much of the research for concurrent data structures, especially the stack, has been performed with CPUs in mind. This leads to an intrinsic part of this paper.

The stack is one of the simplest data structures. It has a high degree of potential when pairing a stack's design with the GPU's compatibility with simple and repetitive tasks. A GPU stack would still suffer from the prevalent issue of thread contention as now thousands if not tens or hundreds of thousands of threads are attempting to access the stack and complete. This is only a single example of design problems which must be addressed.

In this thesis, we propose a lock-free concurrent search-based stack for GPU. This proposed stack brings a design and optimizations tailored to take advantage of the GPU friendly memory access patterns of the array structure. The techniques presented address major concerns like thread contention and memory access patterns.

First, we discuss the stack's general structure and the search area reduction technique called ¡*lowest*-

area¿. The array representing a significantly large stack led to the optimization of searching for the stack's top. This improves the efficiency of operations no longer having to search over mostly invalid space. We also observe this property benefiting from coalesced memory accesses as a group of operations will be searching the same space.

Secondly, we present a GPU-friendly iterative push and pop operation that exploit the array's memory structure on GPU. This results in an efficient searching technique that does not require the usage of a high contention global top reference.

Lastly, we design an elimination structure to take advantage of the thread arrangement on GPU. We reduce contention by using a thread-block-level elimination design that does not require global synchronization.

Our experiments show that our proposed scan stack has a high throughput which scales with elimination. Comparing its design to a baseline with none of the proposed optimizations, the elimination method results in nearly 39 times the throughput than if no optimizations were performed.

Chapter 2

Related Work

Hendler et al. [3, 4] proposes the usage of a backoff elimination stack for CPU to create a scalable lock-free algorithm. They then go on to propose two methods for the backoff mechanism - elimination backoff and adaptive elimination backoff. The first results in a failed attempt to perform an action on the stack being added to the elimination array and cycling through until a successive operation. The second allows for individual threads in the collision to "independently chooses a subrange of the collision layer it will map into" and local counters, influenced by collisions, to determine the thread's collision lifespan. Colvin et al. [2] would later visit the prior algorithm both simplifying data structure and elimination aspects and rectifying an error they determined in the elimination of like operations. In the end, the overall goal of this backoff elimination implementation is to reduce the overall contention over the head of the stack

Shafiei [13] presents a non-blocking stack algorithm for an array based stack. The operations are straightforward taking in an outer object "top" which will be updated by either a pop or push through compare and swap (CAS). As CAS is used for the update of the top's contents, the contention of the threads at a given top with a single successful return while all other threads will require another iteration of the loop. This implementation retains the contention where all threads will fight over who succeeds first, but it has the benefit of not using a linked list which requires pointers between the nodes.

Much like Shafiei, Michael and Scott [8, 9] presented a blocking and non-blocking concurrent queue algorithm. Their blocking technique serialized the head and tail pointers to reduce the number of that could occur at those positions to 1 each. While their algorithm uses a lock on both head and tail positions, they argue that it is non-blocking due to the nature of failing to perform the action. By this, a failed action instead performs a loop to instead of waiting for the lock to be unlocked. Michael and Scott [7] later go on to create a lock-free deque, no longer using locks to create a non-blocking structure.

Misra et al. [10] presents a listing of 4 concurrent data structures on GPU using the CUDA programming language. They review the nature of CUDA GPU programming homing in on prime issues which are not as prevalent on the CPU. One of the most prevalent is the way in which the Streaming Multiprocessors (SM) handle the threads and thread blocks which they are given. To start, an individual SM has a Register, L1 cache, and Read-only memory; and a L2 cache is connected to all SMs then it is connected to what amounts to global memory for a GPU. As such the memory blocks assigned to a specific SM are entirely disjoint from one another and cannot be accessed unless the L2 cache or global memory is used. Thus, a penalty of slow access to global memory must be paid in order to do so, such as in the event of creating a pointer from one memory address locally to another memory address on another SM and attempting to access the latter.

Massalin and Pu [6] presented a multiprocessor OS kernel that uses compare-and-swap for synchronization. They go on to detail the several parts of their kernel, and the most significant here was their description of how a LIFO stack was implemented. By using a global reference, their LIFO stack uses a top reference to get the position, and it then gets the adjacent positions for validation. By evaluating if nearby positions have been altered, the operations are able to discern if the state of the stack has changed.

Peng [12] presented an array-based stack. Using a global variable T for top, operations would fetch-and-add or read and attempt to compare-and-swap their values into the array. Peng introduces a method of creating "invalid" cells that are the result of the target cell of pop containing a specific value or CAS success signifying an empty cell. This paper puts forth the main benchmarks that will be used withing Section 4 later within this paper.

Park and Lin [11] present a GPU stack for IoT (Internet of Things), and mobile devices. Targeted towards machine learning and security, they discuss the challenges and problems related to a GPU stack. Many Problems in this instance refer to their usage of the CPU in tandem with the GPU for mobile application's machine learning.

Abhinav and Nasre [5] present a garbage collection program for reclaiming memory that cannot be reached by a running program. In this, they show a hybrid stack tailored for GPU that uses a global top variable. This stack is hybrid in the case that it is used in both local and global memory, and it swaps between them depending on local memory's fullness. As they use a reference based stack array for referring to graphs, the array also operates as a linked list between the nodes of the graph.

Borisenko et al. [1] present a GPU solution to the branch-and-bound problem. The method for solving uses a degree of serialization to determine the subtrees of each thread. Their granularity determines the depth of the tree that they begin parallelization on. Granularity is then noted to heavily affect the

productivity of the solution. They show that their recursive method was generally faster.

Troendle et al. [14] present a specialized concurrent queue, and they detail key features of concurrent data structures and management techniques. Noting that thread divergence, caused by threads in wavefront not sharing the same code path (i.e., thread a has an if condition-true while others do not), can heavily affect performance. They put forth the idea of using a proxy thread, mentioned in Section 3.3.1, to perform the tasks of other threads within a wavefront.

Chapter 3

Design and Implementation

In this section, we describe the details of the base structure, operations, and optimizations of our proposed stack, named Scan stack.

3.1 Base Structure

The proposed Scan stack has a simple base structure that is based on a one dimensional array. Figure 3.1 shows a simplified example of the Scan stack that holds integer values. This array structure offers a benefit of GPU friendly memory access patterns from a group of threads when GPU executes threads in a Single Instruction Multiple Threads (SIMT) fashion.

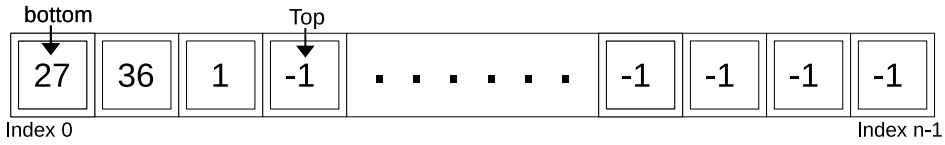

Figure 3.1. An example of the Scan stack storing integer values. The empty cells are designated by a value of -1.

The scanning refers to the process of searching for the top of the stack which can be located anywhere in the array. The requirements to perform a stack operation will be discussed in the relevant subsections in Section 3.2. The proposed scanning technique helps avoid an ABA problem. *The ABA problem refers to when a location is read twice, has the same value for both reads, and another thread has changed the value and performed work between the two reads before resetting to the original value.* Compared to using a global variable for the top, this scanning approach helps decrease thread contention due to the threads independently searching for the top. As a thread scans each cell of the stack array, most of the process

involves reading from memory unless the operation conditions are met. Contention occurs mostly when multiple threads reach the top and attempt to perform their action. This behavior is discussed further in Section 3.3.

Due to the array structure that makes up the Scan stack, scanning through the array to find the top becomes much more efficient on GPUs. This is due to a high likelihood of threads performing the same operations within the same warp accessing sequential or identical positions. This behavior of the Scan stack allows exploitation of the memory coalescing property implemented on modern GPUs.

To limit the number of unnecessary scans from the operations, threads implement a proposed *lowest necessary area* optimization shown in *Figure 3.2* when performing operations. Implementing this results in operations beginning much closer to where the top could be, and lowest-area then significantly decreases time wasted scanning when the stack is mostly full (push) or empty (pop). By scanning at relatively large intervals (i.e., intervals of X = 25,000) to find a new starting position, an operation will need less time to find the top than if it were to start at indices 0 or n-1. The new starting position is determined to be 1 interval before (for push) or after (for pop) the first empty cell discovered this way. However, no thread knows exactly where the top is located in the stack.

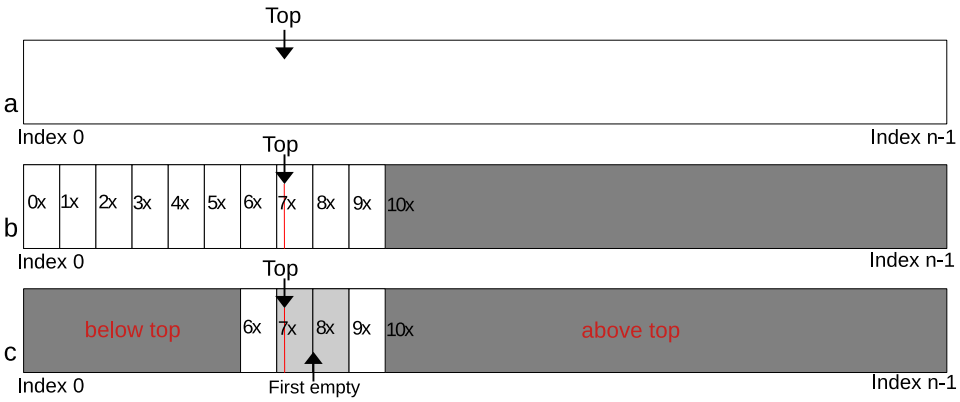

Figure 3.2. (a) A sample stack with an example top location shown. (b) Shows how the lowest necessary area scans at intervals of X to find where an empty cell is. This empty cell shows that the scan has passed where top could be located. In (c), an empty cell is observed at the 8th interval, new pushes will then begin at the start 7th interval (i.e., below the top) while pops begin from the 9th interval (i.e., above the top).

Lowest-area comes with a fundamental problem alongside the reduction of redundant scans. After a starting point has been determined, the point could have had its state altered by another operation. For instance, there exists the case where push's start has been evaluated to be 1 less than the actual top due to that point's scan seeing a valid value. This would create a problem if active pop's then cause the top to now be below this starting point. The push would then scan the cell as being empty and immediately attempt to push its value even though there are non-filled values below the starting cell. This occurring with a pop

operation is unlikely as there will always be at least 1 interval between top and pop when the pop start is initially evaluated. However, an invalidation-cell is discussed in Section 3.2's pop operation that mitigates this resulting in an invalid stack state.

3.2 Stack Operations

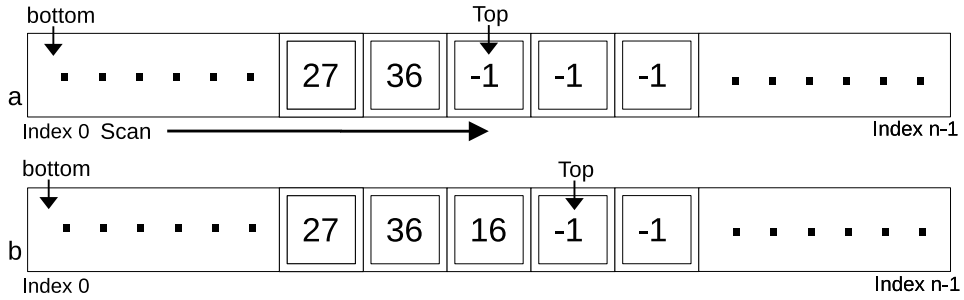

Figure 3.3. Push operation attempting to insert the value 16 into the stack. The stack's top is found when an empty cell (−1 integer value) is encountered. Once the top is found, an atomicCAS() is performed to exchange the insertion value (i.e., 16) with the present one (i.e., −1).

Two main operations supported are push and pop, both of which incorporate the scanning.

The *push* operation inserts an integer value into the stack (pseudo code is shown in Algorithm 1). This is done by scanning from the lowest point (i.e., index 0) in the stack and moving upwards until the empty cell (i.e., −1) is found. Then, an atomic compare-and-swap (CAS) operation is attempted upon that point as in *Figure 3.3*. If successful, the operation will cease and return the value that was inserted to the top of the stack as a sign of success. If CAS fails (shown in *Figure 3.4*), the push operation will take a step away from its previously found top, in the direction of index 0. Afterwards, the thread will continue to take steps toward index 0 in search of a non-empty cell before beginning to scan toward index n-1 (deeper analysis of this handling is covered in Section 3.3). Due to the finite nature of an array, it is possible that the push operation will result in a "failure" due to a full stack, and this case is handled by returning a value indicating such a failure.

The *pop* operation removes the top value from stack (pseudo code is shown in *Algorithm 2*). Similar to push, a pop operation scans from the highest point (i.e., index n-1) in the stack moving downwards until a non-empty value, represented by X > -1 is found. However, a pop operation also scans for an invalid cell, represented by X = -2, to be found.

After identification of a non-empty cell, a CAS is performed attempting to extract and replace the value with an invalidated value (i.e., -2). If successful, the operation will cease and return the value that

Algorithm 1 Scan Stack Push()

```
1: n ← Stacksize;
2: gran ← LAGranularity;
3: temp ← −2;
4: val ← InsertValue;
5: star ← 0;
6: track ← false;
7: i ← 1;
8: while i ∗ gran < n do
9:     if stack[i ∗ gran] = −1 then
10:        star ← (i − 1) ∗ gran;
11:        break;
12:    else if (i + 1) ∗ gran ≥ n then
13:        star ← i ∗ gran;
14:        break;
15:    end if
16:    i ← i + 1;
17: end while
18:
19: for i ← star → n − 1 do
20:    blip ← stack[i];
21:    if stack[i] = −1 then
22:        if atomicCAS(&stack[i], −1, val) = −1
   then
23:            temp = val;
24:            break;
25:        end if
26:        temp ← −3;
27:        i ← i − 1;
28:        while stack[i] < 0 && i > 1 do
29:            i ← i − 1;
30:        end while
31:        if i = 1 then
32:            i ← i − 1;
33:        end if
34:        track ← true;
35:        blip ← stack[i];
36:    end if
37:
38:    if i = n − 1 then
39:        if track = true then
40:            if blip ≠ −1 then
41:                temp ← −2;
42:                break;
43:            end if
44:            temp ← −3;
45:            break;
46:        end if
47:        if track = false then
48:            temp ← −2;
49:            break;
50:        end if
51:    end if
52: end for
53: return temp;
```

was extracted as shown by *Figure 3.5*. If a CAS failure occurs, the pop operation will take a step away from its previously found $top - 1$, in the direction of n-1, as shown in *Figure 3.6*. Afterwards, the thread will continue to take more steps towards index n-1, until an empty cell is discovered, before beginning to scan toward index 0 (deeper analysis of this handling is covered in Section 3.3). This is to account for if the stack has increased in size since the last observation of the stack. As it is possible that a pop will result in nothing due to an empty stack, a return (i.e., -2) is designated for such an event.

In the case of an invalid cell being identified, the cell is evaluated to have an empty cell above itself in the stack. If so, a CAS is performed attempting to re-validate the cell by replacing the invalidated value (i.e., -2) with an empty value (i.e., -1). Whether successful or not, the pop operation will repeat the process as if a CAS on a non-empty cell had failed. This behavior is to account for the cell above the invalid cell possibly having a value pushed to it during the process.

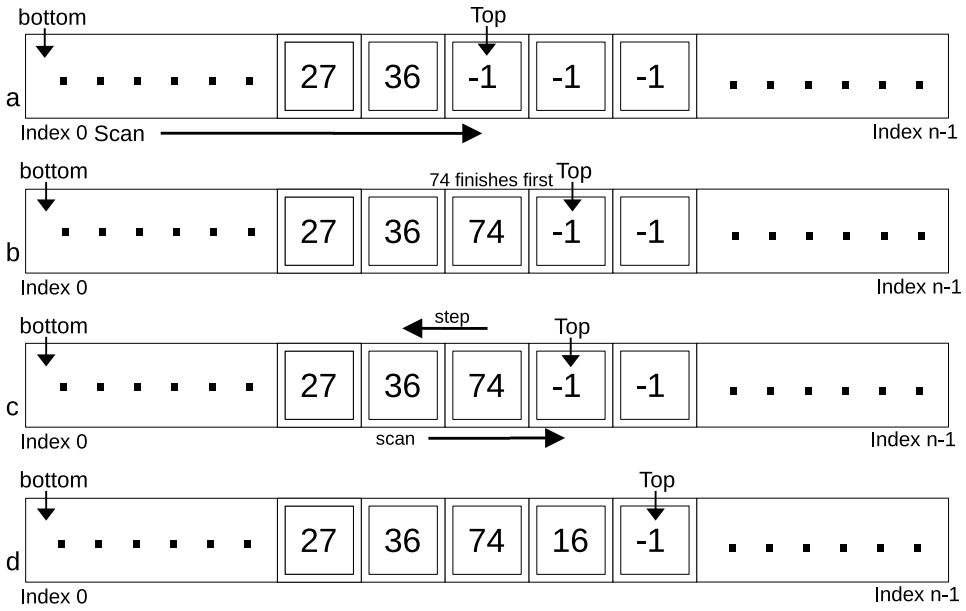

Figure 3.4. (a) Push operation has failed its task to atomicCAS() 16 into an empty cell as the cell is no longer empty (b). In (c), the thread regresses to 36's cell finding a non-empty value, so it proceeds to move forward in the stack once more until an empty cell is found (d).

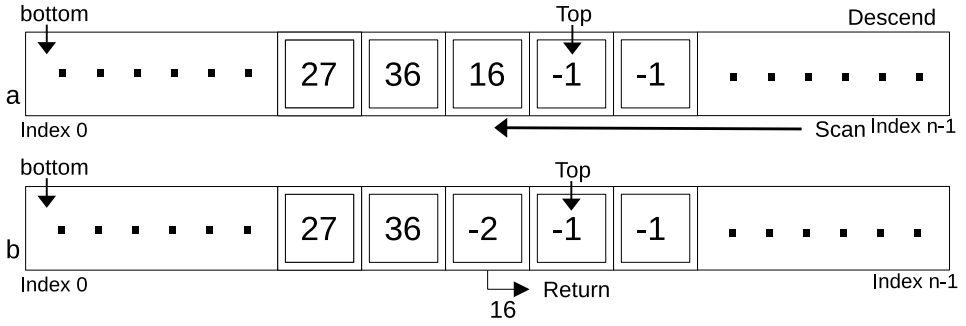

Figure 3.5. Pop operation attempting to remove the top integer value from the stack. Once the top is identified, an atomicCAS() is performed to extract the value.

3.3 Contention Handling

The concurrent access of the stack ensures that cases described within Figure 3.4 and Figure 3.6 are likely to occur. Thus, the top position which represents the most concerning cell of opposing operation contention should be handled carefully. To take a snippet of code from Algorithm 1 shown in Algorithm 3.

If the current cell of the stack is scanned and seen to contain an empty value (i.e., -1) or invalid value (i.e., -2) while performing a push operation, this means that the stack has shrunk from a successful pop operation. The thread must now re-scan to a point which pushing would be a valid action to perform. Notice on line 1, that even if the push failed due to another push succeeding, the failed operation will always

Algorithm 2 Scan Stack Pop()

```
1: n ← Stacksize;
2: gran ← LAGranularity;
3: star ← n − 1;
4: track ← false;
5: i ← 1;
6: while i ∗ gran < n − (gran + 1) do
7:     if stack[i ∗ gran] = 0 then
8:         star ← (i + 1) ∗ gran;
9:         if (i + 1) ∗ gran ≥ n then
10:            star ← n − 1;
11:        end if
12:        break;
13:    end if
14:    i ← i + 1;
15: end while
16:
17: for i ← star → 0 do
18:    if i < n − 1 && stack[i] = −2 && stack[i + 1] = −1 then
19:        if atomicCAS(&stack[i], −2, −1) = −2 then
20:        end if
21:        i ← i + 1;
22:        while stack[i]! = −1 && i < n − 2 do
23:            i ← i + 1;
24:        end while
25:        if i = n − 2 then
26:            i ← i + 1;
27:        end if
28:    end if
29:    blip ← stack[i];
30:    if stack[i] > −1 then
31:        temp = stack[i];
32:        if atomicCAS(&stack[i], temp, −2) = temp then
33:            break;
34:        end if
35:        temp ← −3;
36:        i ← i + 1;
37:        while stack[i] > −1 && i < n − 2 do
38:            i ← i + 1;
39:        end while
40:        if i = n − 2 then
41:            i ← i + 1;
42:        end if
43:        blip ← stack[i];
44:        track ← true;
45:    end if
46:
47:    if i = 0 then
48:        if track = true then
49:            if blip = −1 then
50:                temp ← −2;
51:                break;
52:            end if
53:            temp ← −3;
54:            break;
55:        end if
56:        if track = false then
57:            temp ← −2;
58:            break;
59:        end if
60:    end if
61: end for
62: return temp;
```

step back at least one step purely due to failing. In doing so, threads that have failed a push will step to 1 cell below the highest non-empty cell, and failed pops will have stepped to 1 cell above the first empty cell. Pop operations use an inverse of the Algorithm 3 shown above. Threads will then be setup to re-scan the local top area in the event of alterations.

Figure 3.7 shows an example case of multiple operation types contending for their viewed top of the stack. What can be seen is that 4 operations (2 pushes and 2 pops) are all attempting their own operation. The first push succeeds exchanging the empty cell with the new value (i.e., 16). Following this, the next push now attempts its atomicCAS() yet the value appearing at the old top is no longer empty. As the second insertion fails, the thread must now regress below the current cell. However, the pop's had begun to scan

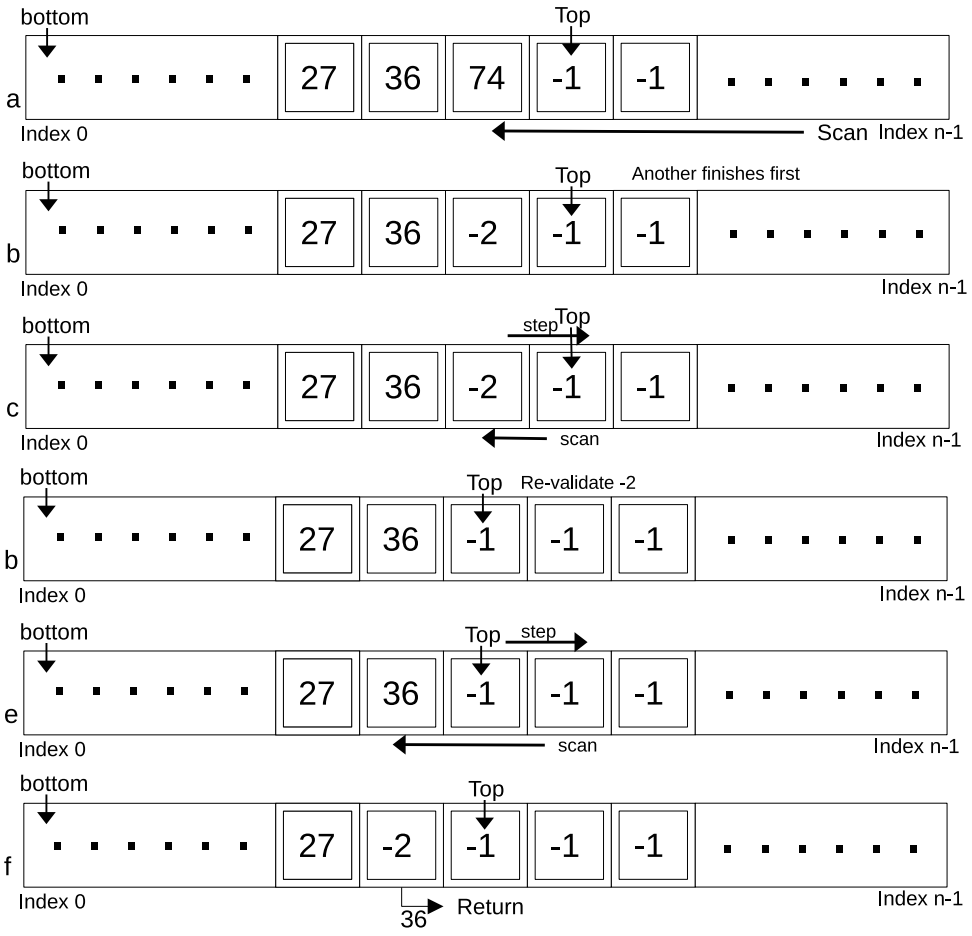

Figure 3.6. (a) Pop operation that has failed its task to atomicCAS() top and create an invalid cell. This is due to another thread completing (b). In (c), the thread climbs to view an empty cell (i.e., -1) showing it is above the stack before scanning continues down the stack. An invalid cell (i.e., -2) is observed and re-validated (d). (e) After renewing the invalid cell, the operation climbs the stack again repeating the process in (c). (f) the operation succeeds in creating the invalid cell and returning the value from the cell.

the cell and saw an empty cell thus scanned below the Top finding a non-empty cell. Both pop operations will save the current value and attempt to atomicCAS() and create an invalid cell (i.e., -2). However, only one will succeed returning the cell's value while the other must ascend to find an empty cell.

The case previously described is very well structured. However, the operations occurring in push followed by pop is not always guaranteed. Figure 3.8 explores a different case of this problem. Say that the order of completion occurs in a push, pop, pop, and push fashion. Then, the first push now scans an empty cell, and it attempts its atomicCAS(). The following pop's scan shows a empty cell resulting the operation moving on to the cell below, and it will pop the value below top creating an invalid cell. The second pop will have also observe an empty cell and moved below, but it will fail then ascend the stack due to the cell being invalid. Finally, the last push sees that the cell is now non-empty and regresses down the stack before

Algorithm 3 Push Contention Handler:

```
1: i ← i − 1;
2: while stack[i] < 0 && i > 1 do
3:     i ← i − 1;
4: end while
5: if i = 1 then
6:     i ← i − 1;
7: end if
```

ascending once more.

This functionality works due to an atomic operation such as atomicCAS() forcing other threads performing an atomic operation on a cell to wait until the winner thread has updated the cell's memory address. This benefits from the *lowest necessary area*(LA) discussed in Section 3.1's Figure 3.2 as the operations will have a degree of space due to being spread out, so threads are less likely to be contending with one another.

The space provided by LA does not completely rectify the potential race condition if the invalid cells were not present. This can be seen in the case of Figure 3.8 when the opposing operations step passed one another due to their search conditions being evaluated as true. If the pop operation did not invalidate the previous top, then the stack would now be in an invalid format. This functionality also serves as a way to cause pop operations which have stepped over to ascend the stack once more to pop the concurrently pushed new top. The previous top was the proper top to target, for the first pop operation to view it, as there was no value above it at the time that the operation scanned that cell. By invalidating, the stack will remain in the proper format rather than creating possible gaps of empty cells.

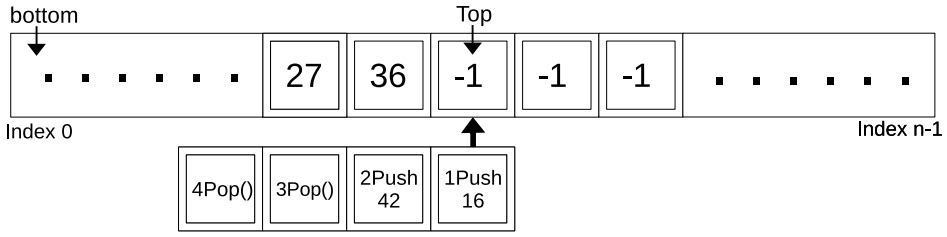

Figure 3.7. Multiple threads attempting an operation upon the top of the stack. Operations are numbered in order of occurrence. Pushes then pops

3.3.1 Elimination

Elimination is performed by implementing an organization structure called a proxy thread into the warps. In CUDA, a warp is (typically) the smallest unit of organization representing 32 threads. The next degree of organization is thread block which can contain a user defined number of threads and can be broken down

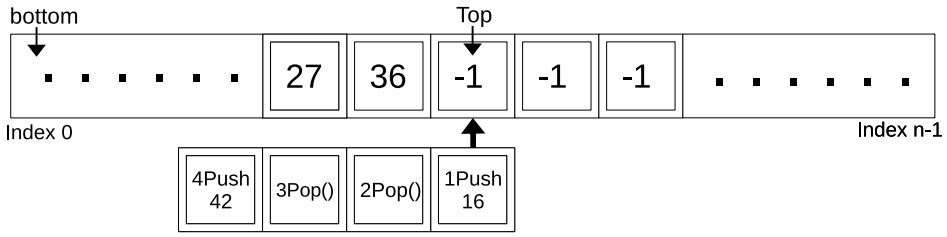

Figure 3.8. Multiple threads attempting an operation upon the top of the stack. Operations are numbered in order of occurrence. Mixed pushes and pops

into warps, but it typically has a power of 2 number of threads designated. Next comes the Grid, but this structure is not significant to this process of elimination used.

As blocks and warps are loaded into local memory for the cores, an access to local memory can be performed that is much faster than attempting to constantly reference a global memory object. As such, a __shared__ array referred to as LIMP (Local IMPacts) is created for each block which will contain the list of operations and their data that the block in question will be performing. This reduces the need to ascend the memory structure and possibly increasing overhead by putting off higher accesses until truly necessary.

By using the warp and block layout through LIMP, an elimination structure that rarely relies on accessing non-local data can be formed. To start, the proxy thread of each warp will scan through its warp attempting to find a pair of push and pop operations that can successfully eliminate. If a pair is found, the push value is handed directly to the pop, but it is possible for nothing to occur if no eliminations are found in the warp. After warp-elimination has completed, elimination ascends to the block level where a non-eliminated operation is first invalidated before it scans looking for its opposite. This invalidation behavior is to prevent another proxy-thread within the block from eliminating an operation that is currently looking for a partner. If no partners are found for an operation, the proxy thread will make the operation valid again while exiting the attempt as it is unlikely that any further pair will be found. A sufficiently large block would be given ample chance to eliminate the majority of its operations. After elimination has ran its course, the remaining operations that have not been eliminated will begin accessing the stack to add their data.

Total grid-based elimination was not implemented. The reasoning for this is due to the requirement of accessing a global array holding all operations to be done, the time necessary for scanning such a space, and the cost if there are no pairs remaining. At a certain total operation count, it is believed that the cost of scanning the totality of operations presented for a pair would far outweigh the benefit.

Another attempt at optimization is the expansion of a proxy thread's duties through serializing the remaining operations after elimination. In doing so, the number of threads which will actively enter contention onto the stack are decreased. However, instilling a sense of order into this structure comes at the

price of performance when elimination is not possible, and this is further discussed later within Section 4.

Chapter 4

Experimental Results

In this section, we outline the benchmarks, performance metrics, and tests that are used to gauge the efficacy of the proposed Scan stack. An Nvidia RTX 3090 was used for all benchmark evaluations presented in this section. The operating system for the test machine was Ubuntu 20.04.2 LTS with a Linux 5.15.0-46-generic kernel, and CUDA version installed is release V11.4.

We used a similar benchmarks as Peng[12] did. They are described below.

- RandomMix – a random mix of operations where there is about 50% chance of either a push or pop operation being performed.
- Push Only – a set number of only push operations being performed.
- Pop Only – a set number of only pop operations being performed.

All benchmarks are performed and stressed by increasing the number of operations. By starting from a relatively small operations of 1000 and scaling to 10 million total operations, the data produced aides in measuring the scalability and efficiency of the proposed design. The benchmarks mentioned above are tested and compared under 3 different optimization designs: No-Optimizations, Lowest-Area, EL+LA (Elimination + Lowest Area), and EL+LA+PX (Elimination + Lowest Area + Proxy Thread). To note, all designs after lowest-area include the optimization of lowest-area. The benchmarks of RandomMix and Push Only will be performed on a completely empty stack, but Pop Only will use the stack produced by Push Only to perform its benchmark.

The primary performance metric we used to evaluate the effectiveness of the proposed stack is throughput (i.e., Operations per Second), but execution time (milliseconds) to complete the benchmark is also measured for plotting and reference. By generating a graph based on the data collected from scaling the

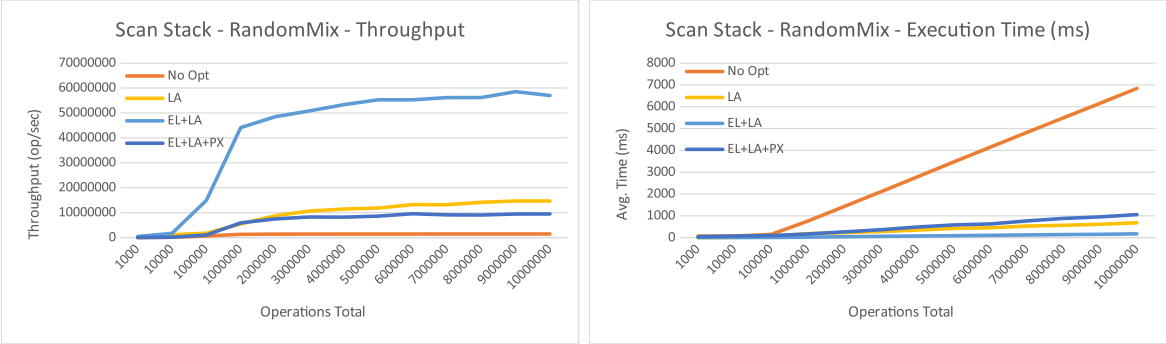

Figure 4.1. Throughput and execution time of randomly chosen operation benchmark under various optimizations (No Opt - No-Optimizations, LA - Lowest-Area, EL+LA - Elimination-LA, and EL+LA+PX - Elimination-Proxy)

Total Operation Count's throughput (RandomMix)						
Method	$5 * 10^6$	$6 * 10^6$	$7 * 10^6$	$8 * 10^6$	$9 * 10^6$	$10 * 10^6$
No-Optimizations	1,439,356.776	1,444,920.521	1,449,340.998	1,454,797.223	1,460,019.483	1,462,312.536
LowestArea-Only	11,840,408.35	13,215,394.26	13,199,229.99	14,144,443.29	14,673,944.94	14,676,680.08
EL+LA	**55,253,767.66**	**55,265,768.48**	**56,151,819.55**	**56,170,150.62**	**58,541,509.83**	**56,974,191.83**
EL+LA+PX	8,604,756.021	9,560,992.68	9,113,620.944	9,065,912.927	9,452,892.822	9,475,859.868

Table 4.1. Operations/second for total operation counts of 5 to 10 million. Data referenced in Figure 4.1

operation count, a plateau on the program's throughput may be discovered leading to further opportunities for enhancement. Following data presented by the benchmark tests described, a promising behavior emerges when more than one type of operation is performed within the thread blocks and the proxy threads usage limited only to elimination.

By comparing the data represented by Figure 4.1, 4.2, and 4.3 at its highest load, throughput observations of the RandomMix benchmark that uses EL+LA+PX method shows roughly 648% the capability of the no-optimization method. However, the lowest-area method by itself boasts roughly 1003% no-optimizations capability when the EL+LA+PX implements this feature. Meanwhile, by limiting the proxy thread activity to EL+LA (Method shown in Figure 4.1, 4.2, and 4.3) yields a significant increase in efficiency to roughly 3896% of no-optimizations. This shows that the EL+LA is not only over 6 times greater than EL+LA+PX but also nearly 39 times more efficient than no-optimizations.

Another significant boon to limiting the proxy thread's activity was observed when only a single type of operation is performed. While the throughput of the EL+LA Method was observed to be roughly 376% of the non-optimized's capability for pushing, the EL+LA+PX form performs only 97% of the same metric. The lowest-area on the other had was shown to be 414% of no-opt. Finally, the pop-only behavior is evaluated to be the worst performing across the board. EL+LA+PX operates at roughly fifth of its pushing performance at 19%, lowest-area though is the effectively tied in efficiency with no-optimizations with 99%, and EL+LA is very similar to lowest-area as it has 100% of non-optimized.

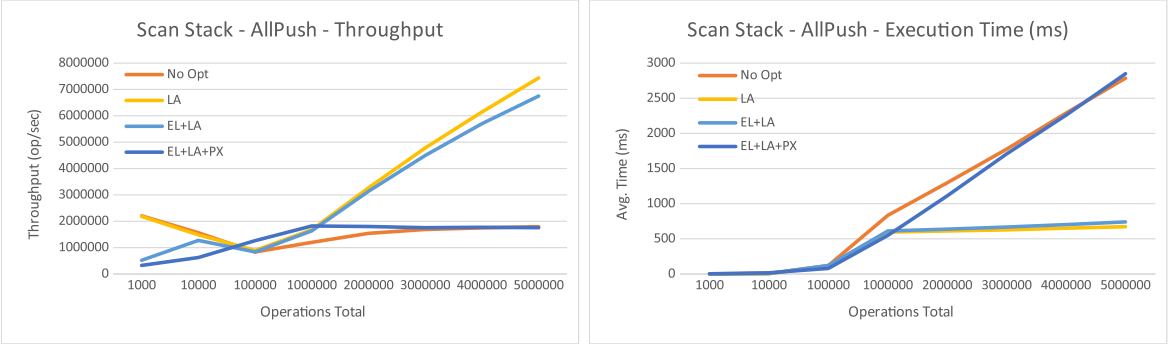

Figure 4.2. Throughput and Time of only-pushes benchmark with optimizations. No Opt - No-Optimizations, LA - Lowest-Area, EL+LA - Elimination-LA, and EL+LA+PX - Elimination-Proxy

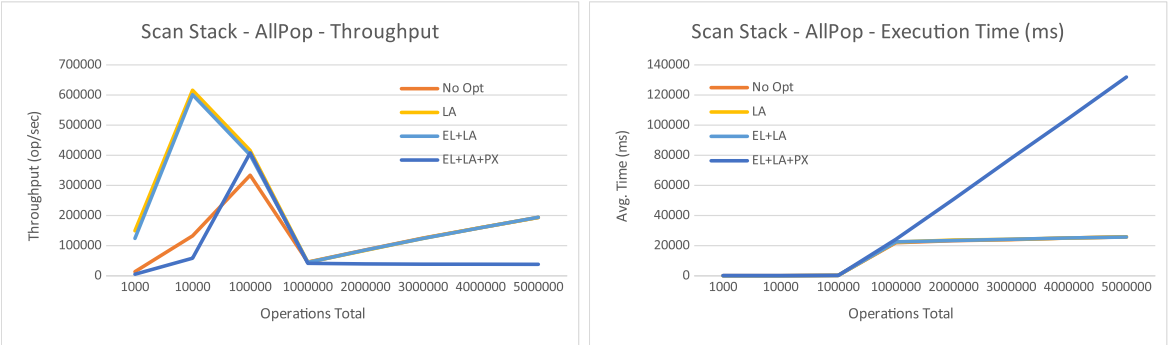

Figure 4.3. Throughput and Time of only-pop benchmark with optimizations. No Opt - No-Optimizations, LA - Lowest-Area, EL+LA - Elimination-LA, and EL+LA+PX - Elimination-Proxy

An analysis of the EL+LA+PX method vs. EL+LA provides insight to why the EL+LA+PX throughput is worse than no-optimizations. The key difference between the two is only within the final structure after elimination has completed. EL+LA, after synchronizing a thread block, will allow each remaining valid operation of that block to access the stack. All threads that still have a valid operation will be the same operation within that block, so a block that has more push operations than pops will have only pushes remaining. This leads to the beneficial memory coalescence property discussed in Section 3. However, the EL+LA+PX, after synchronizing a thread block, uses 2 proxy threads for organizing same operation's entry into the stack. This results in the case where a single push and pop can be done at a time for each warp. While this lowers contention by limiting the stack access, the data of Figure 4.1 supports the claim that doing such results in a significant loss in efficiency. This is the case as well when observing Figure 4.2 as EL+LA+PX's throughput actually appears to decrease rather than increase or plateau. Meanwhile, EL+LA shares a very similar structure to no-optimizations after the elimination process. As such, the efficiency loss between the two when performing only a single operation type is due to the elimination process attempting to find canceling pairs, and this also results in a method that heavily benefits from a mix of operations being

performed.

It is shown in Figure 4.3 that the throughput of all methods presented plummet at the 1 million operation mark for pop. The examination of this is due to the invalidation process that is implemented to ensure the stack is in the form of a stack. Once invalidations begin to occur, the amount of atomic operations that a pop operation has to perform is potentially doubled. This is due to the method in which invalidations are cleared, as only pop operations create and clear them. By guaranteeing the safety of stack top's race condition mentioned briefly in Section 3.3, the overall efficiency of all methods is affected negatively. Analyzing both Figure 4.2 and Figure 4.3, it is noted that, even though all-pop initially plummets, both all-push and all-pop result in a scalable increase in throughput for the LA and EL+LA implementations. In the case of all-pop, the no-optimization method operates equivalently to the LA and EL+LA methods showing that they are no worse in this event. However, All-push results in the LA and EL+LA methods to outpace the efficiency of the no-optimization and EL+LA+PX methods that appear to create a flat line. All-push does not see the same dive in throughput as all-pop due to its lack of interaction with invalidated cells. For a push operation, these invalid cells are simply ignored.

This data shows that by having mixed batches of operations, the proposed EL+LA method creates an efficient increase in the scan stack throughput. It also benefits from being roughly equivalent, if not better, to that of the no-optimizations form when a single operation type is performed.

4.1 Correctness

Correctness was determined by evaluating that there were no empty gaps within the stack, and all operations completed in some capacity by either missing or having a non-empty value for its success. The exact ordering of completion is ignored as the non-blocking approach and high concurrency of GPUs is unlikely to create the same stack between tests. However, a problem of the proposed implementation was determined during correctness testing.

The proposed scan method does not guarantee absolute correctness. This is due to the invalidation process not completely resolving the step-over problem discussed in Section 3.3. As shown with Table 4.2, the invalidation process appears to benefit the method where elimination is included the most. It is also to be noted how volatile the invalidity is even when evaluated at a large amount of iterations at all intervals. There is also how the LA Interval size itself affects the invalidity rate with no other alterations made, and this is primarily shown as a lower average execution time with a higher invalid amount at lower intervals. However, the invalidation of cells within the stack is shown to be highly detrimental when elimination is not involved as there is a significant increase in the invalid stack structure count in Table 4.2, but the opposite

Average Amount of Invalid Stacks (X out of 1000)											
LA Interval:	N/A	2500	5000	7500	10000	12500	15000	17500	20000	22500	25000
LA	0	93	2	0.67	0.33	0	0	0	0	0	0
EL+LA	1.67	482	320.33	174	90	39	36	34	24	47	22.33
Inval+LA	0	470.33	155.67	120.67	33.33	15.33	4.33	0.33	0.67	0	0
Inval+EL+LA	3	126.67	68.33	11.33	3	0.33	0.33	0	0.67	0	0
Average Execution time (Sec) of 1000 iterations											
LA Interval:	N/A	2500	5000	7500	10000	12500	15000	17500	20000	22500	25000
LA	572.15	79.02	85.69	90.52	94.11	98.69	103.49	108.11	113.52	118.13	122.96
EL+LA	165.28	4.39	5.89	7.41	9.17	10.96	12.53	14.35	16.28	17.86	19.67
Inval+LA	699.32	145.62	147.89	154.75	164.40	173.85	183.11	189.26	197.51	205.59	213.92
Inval+EL+LA	212.70	5.61	7.81	10.66	13.33	15.96	18.71	21.07	22.05	24.23	26.60

Table 4.2. By performing 1000 iterations of 1 million operations with varying LA Interval sizes, the above data is produced. Only the RandomMix was used for this data. N/A shows failures and times when LA was not performed, this means LA is actually No-Optimizations for N/A.

is seen for the method including elimination.

Analyzing this behavior, the lack of out-of-bounds protection for LA becomes much more prevalent at lower intervals due to operations beginning their work at closer spaces. This leads to the higher possibility where a push has already exceeded where pop is starting or vice versa. However, completely removing LA does not resolve this even with Invalidation implemented as well. This means that there are cases where pops are re-validating a cell which push has just passed and began attempting to push after. For the El+LA and its invalidation counterpart where LA is N/A Table 4.2, the invalid stack structure still occurs even though the LA and Inval+LA do not appear to share this behavior. It is speculated that this is because the number of active threads within a block has possibly been reduced, so there are blocks loaded onto the cores with a number of threads attempting to perform work that may be significantly less than the block size. While the elimination proposed leads to less contention and quicker action for the block overall, it is possible that there are greatly varying workloads between blocks that can lead to a similar situation as LA scoping not being protected. While much more rare than a low interval of 2500, the invalid structure still occurs meaning that the stack does not behave as it should.

While not guaranteeing complete correctness, the invalidation of cells does appear to be highly beneficial to the EL+LA method. However, this comes at the cost of efficiency. The observed execution time for a low interval EL+LA method yields high throughput for the 1000 sets of 1 million operations, even with the performance detriment of writing to a global array to be transferred for validation. As reference, the 2500 LA interval in Table 4.2's execution time reveals that 1 billion total operations took 4.39 seconds on average, and this calculates that throughput for this EL+LA reaches 227.79 million operations/second. The invalidation counterpart of EL+LA performs the same in 5.61 seconds with a throughput of 178.25, and it has an overall higher success rate compared to without invalidation. For target purposes that do not require

absolute correctness such as graphics processing, the EL+LA method provides an efficient storage structure for data.

Figure 4.4. Throughput and Time of only-pop benchmark with optimizations, without invalidation. (No Opt - No-Optimizations, LA - Lowest-Area, EL+LA - Elimination-LA, and EL+LA+PX - Elimination-Proxy). Copy of Figure 4.2 provided for reference above.

By referencing Figure 4.3 and its difference to Figure 4.2 and Figure 4.4, the invalidation method used is shown to severely degrade performance of the pop operation. By removing the invalidation altogether, the worst-case scenario of all-pop operations does not suffer as greatly from the contention shown in Figure 3.6 and will function more similar to the failure of push in the amount of actions Figure 3.4. This leads to an overall increase in the pop performance to become more scalable and like the push operation's graph. However, Figure 4.2 shows that the RandomMix benchmark at a high amount of iterations has relatively close execution times for EL+LA and Inval+EL+LA, but LA is significantly more efficient than its invalidation counterpart at all intervals. The difference observed within the LA and Inval+LA is most likely due to the time spent clearing invalidated cells by pop, and the efficiency of LA further highlights the benefits of removing invalidation when correctness is not a concern.

Chapter 5

Conclusion

As data continues to grow and needs more efficient processing, the desire for GPU friendly data structures grows as well. In this paper, we presented an efficient and scalable search-based stack for GPUs, tested using an NVidia RTX 3090. We determine that the scanning implementation's performance improves greatly applying the elimination and lowest-area methods proposed. However while highly productive, it relies heavily on elimination process when it comes to the greatest performance increases. Also, the one-to-one thread mapping also showed consistently better improvements compared to EL+LA+PX's one or two threads performing the tasks of others.

–BIBLIOGRAPHY–

BIBLIOGRAPHY

[1] A. Borisenko, M. Haidl, and S. Gorlatch. A gpu parallelization of branch-and-bound for multiproduct batch plants optimization. *The Journal of Supercomputing*, 73(2):639–651, 2017.

[2] R. Colvin and L. Groves. A scalable lock-free stack algorithm and its verification. In *Fifth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2007)*, pages 339–348. IEEE, 2007.

[3] D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 206–215, 2004.

[4] D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. *Journal of Parallel and Distributed Computing*, 70(1):1–12, 2010.

[5] A. Jangda and R. Nasre. Fastcollect: Offloading generational garbage collection to integrated gpus. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems, CASES*, volume 16, pages 1–10, 2016.

[6] H. Massalin and C. Pu. A lock-free multiprocessor os kernel. *ACM SIGOPS Operating Systems Review*, 26(2):108, 1992.

[7] M. M. Michael. Cas-based lock-free algorithm for shared deques. In *European Conference on Parallel Processing*, pages 651–660. Springer, 2003.

[8] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 267–275, 1996.

[9] M. M. Michael and M. L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *journal of parallel and distributed computing*, 51(1):1–26, 1998.

[10] P. Misra and M. Chaudhuri. Performance evaluation of concurrent lock-free data structures on gpus. In *2012 IEEE 18th International Conference on Parallel and Distributed Systems*, pages 53–60. IEEE, 2012.

[11] H. Park and F. X. Lin. Tinystack: A minimal gpu stack for client ml. *arXiv preprint arXiv:2105.05085*, 2021.

[12] Y. Peng and Z. Hao. Fa-stack: A fast array-based stack with wait-free progress guarantee. *IEEE Transactions on Parallel and Distributed Systems*, 29(4):843–857, 2017.

[13] N. Shafiei. Non-blocking array-based algorithms for stacks and queues. In *International Conference on Distributed Computing and Networking*, pages 55–66. Springer, 2009.

[14] D. Troendle, T. Ta, and B. Jang. A specialized concurrent queue for scheduling irregular workloads on gpus. In *Proceedings of the 48th International Conference on Parallel Processing*, pages 1–11, 2019.

VITA

Noah South:

Degrees:

2018 A.A. in Engineering, Northeast Mississippi Community College, Booneville, MS

2020 B.S. in Computer Science, University of Mississippi, Oxford, MS

Employment:

2019 Undergraduate Teacher's Assistant, University of Mississippi, Oxford, MS

2020 Graduate Teacher's Assistant, University of Mississippi, Oxford, MS