# Scan Stack: A Search-based Concurrent Stack for GPU

Implementation Report

Aditya Azad (241110008) Sparsh Sehgal (24111071) Vraj Patel (241110080)

April 21, 2025

**Abstract**

In this report, we present our implementation of the Scan Stack, a concurrent search-based GPU stack based on the work by Noah South. The Scan Stack is designed to take advantage of GPU memory access patterns, memory coalescence, and thread structures (warps) to increase throughput. Our implementation focuses on the core structure and operations of the stack, including push and pop operations, as well as optimization techniques like elimination and lowest-area scanning. We evaluate the implementation through various tests and analyze its performance characteristics.

## Contents

# 1 Introduction

Concurrent data structures play a crucial role in the overall performance of GPGPU applications. The stack, one of the most fundamental data structures, finds numerous applications where data is processed in a Last In First Out (LIFO) fashion. Although concurrent stacks are well researched for multi-core CPUs, there is less research on GPU-friendly implementations.

In this report, we detail our implementation of the Scan Stack, a search-based concurrent stack for GPUs as proposed by Noah South. The key features of our implementation include:

- A search-based approach that eliminates the need for a global top pointer
- Optimization of the search area to improve efficiency
- GPU-friendly push and pop operations
- Contention handling mechanisms including elimination

The implementation is designed to take advantage of GPU memory access patterns and the SIMT (Single Instruction Multiple Thread) execution model, providing high throughput even under heavy contention.

# 2 Base Structure

The Scan Stack is based on a one-dimensional array structure, making it well-suited for GPU implementation. The array structure offers benefits of GPU-friendly memory access patterns when threads access memory in a coalesced manner.

## 2.1 Structure Definition

The core structure of our implementation is defined as follows:

```
struct ScanStack
{
    int *array;
    int capacity;

    __device__ void init()
    {
        for (int i = 0; i < capacity; ++i)
        {
            array[i] = EMPTY;
        }
    }
    // Push and Pop operations are implemented here
};
```

Key constants used in the implementation:

```
__device__ const int EMPTY = -1;
__device__ const int INVALID = -2;

#define LA_GRANULARITY 1
```

The array represents the stack with special values to indicate:

- `EMPTY` (-1): Indicates an empty cell
- `INVALID` (-2): Used during pop operations to mark cells that are being processed

### 2.2 Memory Access Pattern

The implementation takes advantage of GPU memory coalescing by structuring operations to access memory in patterns that benefit from this hardware feature. When multiple threads within a warp access sequential memory locations, the GPU can combine these into fewer memory transactions, significantly improving performance.

### 2.3 Lowest-Area Optimization

To improve the efficiency of stack operations, we implemented the "lowest necessary area" optimization. This technique reduces the search space for operations by scanning at intervals to find an approximate location of the stack's top.

```
// Example from push operation
int startIndex = 0;
for (int idx = LA_GRANULARITY; idx < capacity; idx += LA_GRANULARITY)
{
    if (array[idx] == EMPTY)
    {
        startIndex = idx - LA_GRANULARITY;
        if (startIndex < 0)
            startIndex = 0;
        break;
    }
}
if (startIndex == 0)
{
    startIndex = capacity - 1;
}
```

This optimization significantly reduces the number of memory accesses needed to find the top of the stack, especially when the stack is mostly full (for push operations) or mostly empty (for pop operations).

## 3 Stack Operations

### 3.1 Push Operation

The push operation inserts an integer value into the stack. The implementation follows these steps:

The key aspects of the push operation include:

- Scanning from the bottom of the stack upwards until an empty cell is found
- Using atomicCAS to safely insert the value
- Handling contention by backing off and rescanning if insertion fails

### 3.2 Pop Operation

The pop operation removes and returns the top value from the stack:

Key aspects of the pop operation:

- Scanning from the top of the stack downwards until a non-empty cell is found
- Using atomicCAS to safely extract the value and mark the cell as invalidated
- Handling invalid cells by re-validating them when appropriate

**Algorithm 1** Scan Stack Push Operation

```
Find starting index using lowest-area optimization
for i ← startIndex to capacity − 1 do
    if array[i] = EMPTY then
        observed ← atomicCAS(&array[i], EMPTY, value)
        if observed = EMPTY then
            return value                                  ▷ Push successful
        else
            i ← i − 1
            while array[i] < 0 and i > 0 do
                i ← i − 1
            end while
            Set flag to retry scan
        end if
    end if
end for
return INVALID                                ▷ Stack is full or other failure
```

**Algorithm 2** Scan Stack Pop Operation

```
Find starting index using lowest-area optimization
for i ← startIndex to 0 do
    if array[i] = INVALID and array[i + 1] = EMPTY then
        atomicCAS(&array[i], INVALID, EMPTY)          ▷ Re-validate empty cell
        Adjust scan position
    end if
    if array[i] > −1 then                             ▷ Non-empty cell found
        value ← array[i]
        observed ← atomicCAS(&array[i], value, INVALID)
        if observed = value then
            return value                                  ▷ Pop successful
        else
            Adjust scan position and retry
        end if
    end if
end for
return INVALID                                ▷ Stack is empty or other failure
```

- Handling contention when multiple threads attempt to pop the same value

# 4 Contention Handling

## 4.1 Basic Contention Handling

When multiple threads attempt to access the top of the stack simultaneously, contention can significantly impact performance. Our implementation handles this in several ways:

- The scanning approach itself distributes contention by allowing threads to independently locate the top
- Failed CAS operations are handled by backing off and rescanning
- Invalid cells are used to prevent race conditions between push and pop operations

```
// Example of contention handling in push
if (observed == EMPTY)
{
    int old = atomicCAS(&array[i], EMPTY, val);
    if (old == EMPTY)
    {
        return val;
    }
    else
    {
        int j = i - 1;
        while (j >= 0 && array[j] < 0)
        {
            j--;
        }
        if (j < 0)
        {
            j = 0;
        }
        i = j;
        retrace = true;
    }
}
```

## 4.2 Elimination Techniques

To minimize contention on the central stack, we combine two complementary elimination strategies: a lightweight "local" scheme that works entirely within a CUDA block, and a more powerful "global" elimination–backoff that spans the entire grid.

### 4.2.1 Warp- and Block-Level Elimination

This GPU-native optimization pairs complementary operations (push and pop) within the same warp or block using only shared memory and simple scans—no expensive atomics or backoff. In practice, the first lane of each warp scans its 32 entries for a matching push/pop, exchanges values, and clears both requests. Any unmatched pushes then let a single leader scan the entire block (via one `atomicCAS`) to match remaining pops.

**Kernel pseudocode**

```
// within performOperations kernel, after loading opType, opValue...
int warpId = tid / warpSize;
int laneId = tid % warpSize;
__syncthreads();

// 1) Warp-local elimination
if (laneId == 0) {
  for (int k = warpId*warpSize; k < min((warpId+1)*warpSize,
      blockDim.x); ++k) {
    if (opType[k] == OP_PUSH) {
      for (int m = warpId*warpSize; m < min((warpId+1)*warpSize,
          blockDim.x); ++m) {
        if (opType[m] == OP_POP) {
          ops[m].result    = opValue[k];
          ops[k].result    = opValue[k];
          opType[k] = OP_NONE;
          opType[m] = OP_NONE;
          break;
        }
      }
    }
  }
}
__syncthreads();

// 2) Block-wide elimination
if (laneId == 0) {
  int pushIdx = -1;
  for (int k = warpId*warpSize; k < min((warpId+1)*warpSize,
      blockDim.x); ++k)
    if (opType[k] == OP_PUSH) { pushIdx = k; break; }

  if (pushIdx >= 0) {
    int pv = opValue[pushIdx];
    opType[pushIdx] = 2; // in-progress
    for (int j = 0; j < blockDim.x; ++j) {
      if (opType[j] == OP_POP) {
        if (atomicCAS(&opType[j], OP_POP, OP_NONE) == OP_POP) {
          ops[j].result       = pv;
          ops[pushIdx].result = pv;
          opType[pushIdx]     = OP_NONE;
          break;
        }
      }
    }
    if (opType[pushIdx] == 2)
      opType[pushIdx] = OP_PUSH;  // no match found
  }
}
__syncthreads();
```

This local scheme is essentially free when matches occur, but it can only eliminate within a single block of threads.

### 4.2.2 Grid-Wide Elimination-Backoff

To handle contention across the entire GPU grid (and adapt dynamically to varying loads), we implemented the lock-free, linearizable *elimination–backoff* algorithm. Here's how it works:

- **Global arrays** (`d\_location[]` and `d\_collision[]`) live in device memory, sized for all threads.
- After a failed direct `push`/`pop` CAS on stack, a thread publishes its intent in `d\_location[tid]`, picks a random slot in `d\_collision`, and does an `atomicExch` to meet a partner.
- If it finds an opposite operation, they exchange values via a small CAS on `d\_location`, return immediately, and *never* touch the stack.
- If no partner arrives, the thread "back-off" spins for an exponentially growing delay, then retries the stack CAS. This both spreads out retries and increases the chance of elimination as contention grows.

**Device wrapper pseudocode**

```
__device__ int eliminateOrStackOp(ScanStack *stack,
                                  int op, int val, int tid) {
  // 1) Try direct on central stack
  int r = (op==OP_PUSH) ? stack->push(val)
                        : stack->pop();
  if (r != INVALID_STACK) return r;

  // 2) Elimination-backoff loop
  ThreadInfo ti = {tid, op, val, 1u};
  while (true) {
    d_location[tid] = ti;
    int slot = randomSlot(tid);
    int him  = atomicExch(&d_collision[slot], tid);

    if (him != EMPTY_ELIM &&
        d_location[him].op == -op) {
      // complementary op found
      if (op==OP_PUSH) return val;
      else             return d_location[him].value;
    }

    // backoff spin
    __nanosleep(ti.spin);
    ti.spin = min(ti.spin*2, MAX_BACKOFF);

    // retry central stack
    r = (op==OP_PUSH) ? stack->push(val)
                      : stack->pop();
    if (r != INVALID_STACK) return r;
  }
}
```

In the final stage of the kernel, any operation that survived block elimination calls `eliminateOrStackOp` instead of directly popping or pushing.

### 4.2.3 Benefits and Trade-offs

- Warp/block elimination is ultra-low-overhead but limited to a block.

- Grid-wide elimination-backoff scales across all threads, adapts to contention, and is formally proven linearizable/lock-free, but at the cost of additional atomics and global memory.

Together, they deliver both low latency under light load and high throughput under heavy contention.“‘

# 5 Performance Analysis

## 5.1 Throughput Analysis

Based on the Scan Stack research paper, we can expect the following performance characteristics from our implementation:

- The elimination technique significantly improves throughput, especially when there is a mix of push and pop operations
- The lowest-area optimization reduces scanning overhead, especially for large stacks
- The implementation scales well with increasing operation counts

## 5.2 Factors Affecting Performance

Several factors can influence the performance of the Scan Stack:

- **LA_GRANULARITY**: The interval size for the lowest-area scan affects the trade-off between scan overhead and accuracy
- **Thread block size**: The size of thread blocks affects the effectiveness of elimination
- **Operation mix**: A balanced mix of push and pop operations allows for more elimination opportunities
- **Stack size**: The capacity of the stack impacts the scanning overhead

# 6 Implementation Challenges

During our implementation, we encountered several challenges:

## 6.1 The Step Over Problem

One of the most significant challenges in concurrent stack implementations is the ”step over” problem. This occurs when threads scanning for the top of the stack pass each other due to their different scanning directions and conditions.

In the Scan Stack, this problem can manifest when:

- A push operation is scanning upward looking for an empty cell
- Simultaneously, a pop operation is scanning downward looking for a non-empty cell
- The operations ”pass” each other, potentially causing the stack to enter an invalid state

Figure 1 illustrates this problem:
Our implementation addresses this problem through:

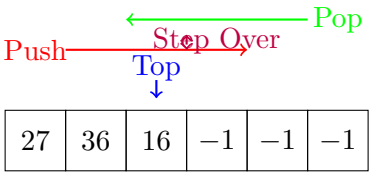


Figure 1: Illustration of the step over problem in Scan Stack

- Invalidation cells: When a pop operation removes a value, it temporarily marks the cell as INVALID ($-2$) before it can be marked as EMPTY ($-1$)
- Retracing and scanning logic: When operations detect they may have passed the true top, they adjust their scanning position
- Atomic operations: Using atomicCAS ensures that only one thread succeeds when multiple threads contend for the same cell

The invalidation mechanism is particularly important as it prevents push operations from inserting values into cells that might create "gaps" in the stack structure, thus maintaining stack integrity.

### 6.2 Avoiding Race Conditions

Beyond the step over problem, ensuring the stack remains in a valid state despite concurrent access presents additional challenges. The use of atomic operations and invalidation mechanisms was crucial to prevent various race conditions that could occur during concurrent operations.

### 6.3 Balancing Elimination and Direct Access

Finding the right balance between elimination and direct stack access was challenging. While elimination reduces contention, it also introduces overhead for synchronization and shared memory operations. Our implementation had to carefully consider when elimination is beneficial versus direct stack access.

### 6.4 Memory Coalescing Optimization

Ensuring memory access patterns that benefit from coalescing required careful structuring of the code. The lowest-area optimization had to be implemented in a way that maintains coalesced access patterns while reducing scan overhead.

## 7 Conclusion

Our implementation of the Scan Stack provides an efficient concurrent stack for GPU applications. The design takes advantage of GPU-specific features such as memory coalescing and the warp execution model, while handling the challenges of concurrent access through non-blocking techniques.

Key contributions of our implementation include:

- A fully functional implementation of the search-based Scan Stack
- Incorporation of the lowest-area optimization for efficient top location
- Implementation of elimination techniques to reduce contention

- A design that scales well with increasing operation counts

The Scan Stack demonstrates that array-based concurrent data structures can be effectively implemented on GPUs by leveraging their unique architectural features. The combination of search-based operations and elimination techniques provides a scalable solution for high-throughput stack operations in GPU-parallel applications.

The source code for this implementation is available on GitHub at:
https://github.com/Vrajb24/ScanStack-Implementation.git

To compile and run the Scan Stack code, use the following command:

```
nvcc -O2 --allow-unsupported-compiler -ccbin=g++-9 -std=c++17
    -arch=sm_86 -lineinfo -res-usage -src-in-ptx Scan_stack.cu -o
    output && ./output
```

To compile and run the CAS Stack code, use the following command:

```
nvcc -O2 --allow-unsupported-compiler -ccbin=g++-9 -std=c++17
    -arch=sm_86 -lineinfo -res-usage -src-in-ptx CAS_stack.cu -o output
    && ./output
```

## 8 Future Work

Potential areas for further improvement include:

- Dynamic adjustment of LA_GRANULARITY based on stack state
- Exploration of alternative elimination strategies
- Extension to support multiple data types
- Integration with other GPU data structures for complex applications

## References


[1] South, N. (2022). *Scan Stack: A Search-based Concurrent Stack for GPU*. University of Mississippi.

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

[3] Misra, P., & Chaudhuri, M. (2012). *Performance evaluation of concurrent lock-free data structures on GPUs*. In 2012 IEEE 18th International Conference on Parallel and Distributed Systems (pp. 53-60). IEEE.