CSE 326 Analysis and Design of Algorithms

Dr.Walid Gomaa

Name	ID
Mohamed Abdelmonem Makram	120220055
Abdelrahman Ahmed Shaheen	120220228
Abdelrhman Mohamed Eldenary	120220253
Anas Ihab Badr	120220360

Computer Science Engineering Department
Egypt-Japan University of Science and Technology

Contents

Schur Numbers Week 7 Report

April 11, 2025

1 Introduction

At the beginning of this week, we decided to tackle the number coloring problem using CUDA to leverage the power of parallel computing. Our goal was to efficiently explore a large decision space where numbers are assigned to color groups under a mathematical constraint (such as $x + y = z$).

We believed CUDA could greatly speed up this backtracking-heavy task. Throughout the week, we experimented with multiple algorithmic strategies to implement this on the GPU. We started with ambitious designs like recursive thread spawning and global queue sharing, then iterated over simpler or more manageable techniques as we faced practical issues.

This report documents our journey through three major failed attempts before arriving at our final, novel, successful solution. Each section discusses what we tried, why it seemed like a good idea at the time, the problems we encountered, and the lessons we learned. By the end, we introduce the final approach — a smart encoding approach to the problem — and explain how it effectively balances performance and correctness under CUDA's constraints.

2 Parallelism in General

As we stated before in Week 5 how our Parallelism should work recursively, we know have a graphical diagram of what the recursion tree should look like to try all valid colorings from $S(2)$ to reach $S(3)$. (Note: The figure couldn't be rendered due to high resolution please view in `images/recursionTree.png` or **here**

So when trying to simulate this recursive tree in Python, it was no problem, and we got the results immediately. But the problem was when we decided to start at $S(2)$ and simulate up to $S(3)$. It took 200 seconds to process all the recursion calls, but the output was successful at the end. Results are shown in Figure 1.
This is considered a problem since we are now way near $S(6)$, and it's taking a lot of time. So, we wanted to test how CUDA would perform in the same situation. We tried a lot of approaches and here's a summary of what we tried and why it didn't work.

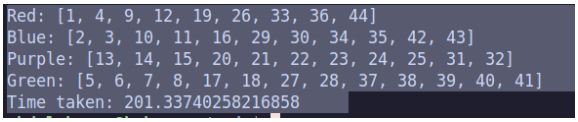

Figure 1: Recursion Tree for $S(2)$ to $S(3)$

3 Failed Attempt 1: Direct Tree-Based Recursive Thread Launching

3.1 The Idea

At an early stage in our project, we explored an approach that directly modeled the decision tree through recursive kernel launches in CUDA. The idea was simple:

- Each CUDA thread would attempt to place a number into a color group.
- If the placement was valid, that thread would **recursively launch child threads** to handle the next number.
- This would form a parallel tree of threads, where each level represented one number in the sequence.

3.2 Why It Seemed Promising

- CUDA offers **dynamic parallelism**, allowing kernels to launch other kernels from the device.
- This mirrored recursive backtracking algorithms used on CPUs and seemed like a natural mapping.
- We expected this would maximize GPU utilization and reduce decision latency.

3.3 Why It Failed in Practice

- **t1. High Launch Overhead:** Each dynamic kernel launch added significant latency, making deep trees extremely slow.
- **t2. Limited Stack Size:** Device-side stack sizes are small, leading to overflows on deep recursion.
- **t3. Resource Exhaustion:** Too many dynamic launches exceeded thread-/block/memory limits.
- **t4. Synchronization Issues:** CUDA lacks mechanisms for parent-child thread sync, making result aggregation difficult.
- **t5. Debugging Nightmare:** Recursive kernels were hard to debug and often failed silently.

3.4 Conclusion

While elegant in theory, recursive tree-based execution did not scale or behave reliably on the GPU. This pushed us to seek flatter, iterative strategies.

4 Failed Attempt 2: Global Queue-Based Work Sharing

4.1 The Idea

Next, we implemented a global work queue approach where each thread would take a configuration from a shared queue, expand it by trying valid placements, and then push the resulting configurations back.

4.2 Why It Seemed Promising

- It resembled a parallel BFS, often used in CPU multi-threading.
- Allowed exploration without deep recursion.
- Theoretically scalable with good memory capacity.

4.3 Why It Failed in Practice

- **t1. Atomic Contention:** Global atomic operations created massive delays.
- **t2. Race Conditions:** Queue corruption occurred frequently despite precautions.
- **t3. Memory Latency:** Queue operations were too slow due to high-latency global memory.
- **t4. Load Imbalance:** Some threads did all the work; others stayed idle.
- **t5. Debugging Complexity:** Race bugs were hard to identify or replicate.

4.4 Conclusion

Despite being more stable than dynamic launching, the queue model was still inefficient due to contention, imbalance, and high memory costs.

5 Failed Attempt 3: Thread-Per-Configuration Brute Force

5.1 The Idea

In this brute-force model, we pre-generated all possible color assignments and launched one thread per configuration to validate legality under the constraint.

5.2 Why It Seemed Promising

- Embarrassingly parallel — no inter-thread communication required.
- Easy to implement and free of control flow divergence.
- Seemed like a good fit for massive GPU parallelism.

5.3 Why It Failed in Practice

- **t1. Combinatorial Explosion:** k^n configurations quickly overwhelmed device capacity.

- **t2. Resource Usage:** Threads consumed too many registers and shared memory.
- **t3. Launch Limits:** CUDA limits were hit before real computation began.
- **t4. Wasted Compute:** Most threads performed useless work on invalid configurations.
- **t5. No Pruning:** No early exit from bad paths — unlike backtracking.

5.4 Conclusion

Although conceptually simple, this brute-force strategy was wasteful and completely unscalable.

6 Successful Approach: Smart Encoding DFS iterative Exploration with Shared Memory

6.1 The Idea

After the above failures, we developed an iterative method that simulated backtracking using stacks in shared memory. Each thread/block maintained its own local stack of partial assignments and explored them in a DFS manner. Our code would start at $S(2)$ and then work its way up to $S(4)$. Here are the novel approaches we took:

6.2 Path Encoding Scheme

We developed a compact binary encoding to represent color assignments efficiently in CUDA memory. Here's how we systematically designed it:

6.2.1 Core Components

- **2 Bits Per Number**: Each number (6–44) uses 2 bits to encode its color:

$$\begin{cases} 00 \rightarrow \text{Red} \\ 01 \rightarrow \text{Blue} \\ 10 \rightarrow \text{Green} \\ 11 \rightarrow \text{Cyan} \end{cases}$$

- **Two-Part Storage**:

Numbers	Storage	Total Bits
6–37	64-bit `part1`	32 nums $\times$ 2 bits = 64 bits
38–44	64-bit `part2`	7 nums $\times$ 2 bits = 14 bits

6.2.2 Walkthrough Example

Consider assigning:

- 6 $\rightarrow$ Green (10)
- 7 $\rightarrow$ Cyan (11)
- 8 $\rightarrow$ Red (00)

Encoded in `part1` as:

$$\underbrace{00}_{8}\ \underbrace{11}_{7}\ \underbrace{10}_{6}\ \ldots = \text{0xE (hex)} \quad \text{(Only first 6 bits shown)}$$

6.2.3 Key Benefits

- **Memory Efficient**: 78 bits total vs $\sim$400 bytes for arrays
- **Fast Validation**: Bitmask checks use native GPU operations
- **CUDA-Friendly**: Aligns with 64-bit memory transactions
- **Unique Signatures**: Enables duplicate detection via bitwise comparisons

6.3 Bitmask Color Encoding

In this technique, we represented each color as a bitmask instead of using arrays. For example, the coloring of $S(2)$ was represented as:

$$\texttt{red} = 0b1001 \quad \texttt{blue} = 0b0110$$

which corresponds to `red = [1, 4]` and `blue = [2, 3]`.

This representation significantly reduced memory consumption—by a factor of 16–32$\times$ in practice—while allowing for extremely fast bitwise operations on color sets. Using native bitwise logic, we could quickly validate constraints and apply transformations directly on GPU registers without additional memory overhead.

6.4 Iterative DFS with Shared Memory

Although the traditional approach would use recursion, we adopted a fully iterative design to comply with CUDA's limited support for recursion and to maximize performance.

Each thread block explored the space using a DFS traversal pattern implemented with explicit stacks stored in shared memory. Each level (starting from $z = 6$) would generate and validate candidate colorings for the next level. Once valid, the state was pushed to the local stack and passed forward for the next iteration.

This pattern continued up to $z = 44$, covering the required depth from $S(2)$ to $S(4)$. Notably, we began at $z = 6$ because the initial 5 nodes were assigned to a new color by definition. This approach allowed us to:

- Parallelize across threads, each handling a separate DFS path
- Maintain high GPU occupancy by reusing shared memory
- Avoid kernel launch overhead by staying within the device context

6.5 Why It Worked

- **Fast Memory:** Shared memory eliminated global latency and contention.
- **No Launch Overhead:** Entire search executed within one kernel.
- **Scalable:** Each block processed a separate subset of the space.
- **Controlled Stack Use:** We sized stacks properly to avoid overflow.
- **Debuggable:** Easier to trace and maintain than previous designs.

6.6 Implementation Highlights

- Per-thread stacks stored the current index and partial assignments.
- Warp-level synchronization used for safe shared memory updates.
- Valid configurations were atomically added to a global output array.

6.7 Performance Results

We tested our final implementation[1] on a range of instances, from $S(2)$ to $S(4)$. The results were promising acheving the same task that Python did in only 300 ms! The results are shown in Figure 2.

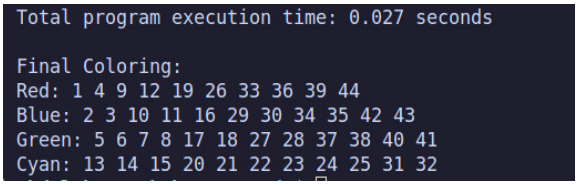

Figure 2: CUDA Time for $S(2)$ to $S(4)$

6.8 Conclusion

This method finally balanced performance, scalability, and maintainability. It took advantage of CUDA's strengths while avoiding known pitfalls, allowing us to solve much larger coloring instances efficiently.

[1]The code can be found in the repo in `Code/main.cu`