

Parallel Programming: Assignment 1

Sirjan Hansda

February 2026

Contents

1	Introduction	3
2	Algorithm	3
2.1	Naive Algorithm	3
2.2	Rabin-Karp Algorithm	3
2.2.1	Polynomial Rolling Hash	3
2.2.2	Prefix Hash Optimization	3
2.3	Hash of Substring	4
2.4	Pseudo Codes	5
2.5	Correctness	5
2.6	Short Note on Modulus	5
3	Implementation Details	6
3.1	String Construction	6
3.2	CPU Implementation (OpenMP)	6
3.3	GPU Implementation (CUDA)	7
4	CUDA-Specific Optimizations	7
4.1	Minimizing Data Transfers	7
4.2	Memory Coalescing	7
4.3	Warp Divergence Minimization	8
4.4	Structure of Arrays	8
4.5	Radix Sort Utilization	8
4.6	Filter-and-Refine Strategy	8
5	Results	9
5.1	Program Outputs	9
5.2	CPU	9
5.3	GPU	10
5.4	Speedup	10
6	CUDA Profiling & Observations	10
6.1	Memory Transfers v/s Computation	10
6.2	Kernel wise Time Split	11
6.3	computeRollingHashes Kernel	12
6.4	parallelSearch Kernel	13

6.5 Overall Run Pattern	14
7 Conclusion	14

1 Introduction

The Longest Common Substring problem involves finding the largest common substring in between 2 strings. Formally, given two strings X and Y of length m , we seek to find the maximum length k such that there exists a substring S of length k where S is present in both X and Y .

2 Algorithm

2.1 Naive Algorithm

The naive approach for this problem involves using the Dynamic Programming paradigm. DP has a space complexity of $O(m^2)$. For a string of length 100 Million (10^8), this would require 10^{16} bytes = 10⁷GB. This much memory is unfeasible. Therefore, we used the **Rabin-Karp** Algorithm. This algorithm is a member of the Rolling Hash class of Algorithms. This reduces the memory requirement to $O(m)$.

2.2 Rabin-Karp Algorithm

To efficiently compare substrings, we utilize the Rabin-Karp algorithm, which employs a rolling hash function. The main idea is to map every substring of characters to a single integer value (the hash). This allows us to compare two substrings of length(L) in $O(1)$ time by comparing their hash values, rather than the $O(L)$ time required for a character-by-character comparison.

2.2.1 Polynomial Rolling Hash

We use a polynomial rolling hash function. For a string S of length L , the hash H is calculated as:

$$H(S) = (S[0] \cdot B^{L-1} + S[1] \cdot B^{L-2} + \dots + S[L-1] \cdot B^0) \pmod{M} \quad (1)$$

Where:

- $S[i]$ is the ASCII value of the character at index i .
- B is a constant base. In our implementation, we use $B = 121$.
- M is a large prime modulus. Our implementation utilizes **unsigned long long** (64-bit integers), which relies on the overflow behaviour for unsigned integers for the modulo operations.

2.2.2 Prefix Hash Optimization

To avoid recalculating the hash from scratch for every substring, we precompute a **prefix hash array** P and a **powers array**.

- $P[i]$ stores the hash of the prefix $S[0 \dots i]$:

$$P[i] = \sum_{k=0}^i S[k] \cdot B^{i-k} \quad (2)$$

This represents: $P[i] = S[0] \cdot B^i + S[1] \cdot B^{i-1} + \dots + S[i] \cdot B^0$

- $Powers[k]$ stores the precomputed value of B^k .

2.3 Hash of Substring

Using these precomputed arrays, we can compute the hash of any substring $S[i \dots j]$ in constant time:

$$\begin{aligned} Hash(S[i \dots j]) &= \sum_{k=i}^j S[k] \cdot B^{j-k} \\ &= S[i] \cdot B^{j-i} + S[i+1] \cdot B^{j-i-1} + \dots + S[j] \cdot B^0 \end{aligned} \quad (3)$$

To express this using the prefix hash $P[j]$, observe that:

$$P[j] = S[0] \cdot B^j + S[1] \cdot B^{j-1} + \dots + S[i-1] \cdot B^{j-i+1} + \underbrace{S[i] \cdot B^{j-i} + \dots + S[j] \cdot B^0}_{Hash(S[i \dots j])} \quad (4)$$

Similarly:

$$P[i-1] = S[0] \cdot B^{i-1} + S[1] \cdot B^{i-2} + \dots + S[i-1] \cdot B^0 \quad (5)$$

Multiplying $P[i-1]$ by B^{j-i+1} :

$$P[i-1] \cdot B^{j-i+1} = S[0] \cdot B^j + S[1] \cdot B^{j-1} + \dots + S[i-1] \cdot B^{j-i+1} \quad (6)$$

Therefore:

$$\boxed{Hash(S[i \dots j]) = P[j] - P[i-1] \cdot B^{j-i+1} = P[j] - P[i-1] \cdot Powers[j-i+1]} \quad (7)$$

For the special case when $i = 0$, we simply have $Hash(S[0 \dots j]) = P[j]$.

2.4 Pseudo Codes

Here, we include the Pseudo Code for the outer binary search that finds the largest common substring. The **CHECKLENGTH** function is different in the openmp and cuda implementations and will be described in the following sections.

Algorithm 1 Longest Common Substring using Rabin Karp

Require: Two strings A and B of length n , hash base $base$

Ensure: Length of longest common substring and its positions in A and B

```

1: Preprocessing:
2: Compute  $Powers[k] = base^k$  for  $k = 0, 1, \dots, n$ 
3: Compute prefix hash  $P_A[i] = \sum_{k=0}^i A[k] \cdot base^{i-k}$  for  $i = 0, 1, \dots, n-1$ 
4: Compute prefix hash  $P_B[i] = \sum_{k=0}^i B[k] \cdot base^{i-k}$  for  $i = 0, 1, \dots, n-1$ 
5:
6: Binary Search on Length:
7:  $lo \leftarrow 0, hi \leftarrow n$ 
8:  $bestLength \leftarrow 0, bestPosA \leftarrow -1, bestPosB \leftarrow -1$ 
9: while  $lo \leq hi$  do
10:    $mid \leftarrow \lfloor (lo + hi)/2 \rfloor$ 
11:    $(found, posA, posB) \leftarrow \text{CHECKLENGTH}(mid, A, B, P_A, P_B, Powers)$ 
12:   if  $found$  then
13:      $bestLength \leftarrow mid$ 
14:      $bestPosA \leftarrow posA, bestPosB \leftarrow posB$ 
15:      $lo \leftarrow mid + 1$  ▷ Try longer substring
16:   else
17:      $hi \leftarrow mid - 1$  ▷ Try shorter substring
18:   end if
19: end while
20: return  $(bestLength, bestPosA, bestPosB)$ 

```

2.5 Correctness

For a substring of length L , the errors by an LCS algorithm can be of two types: False Positives and False Negatives.

Now, False Negatives are impossible because any two common substrings are guaranteed to have the same hash value($\text{Hash}(S[i..j])$ depends only on length of the substring).

False Positives are also impossible because once a possible match is found, we explicitly match the characters one by one from the two strings.

2.6 Short Note on Modulus

We rely on unsigned integer overflow for implicit modulus operation, effectively using $M = 2^{64}$. The choice of modulus involves a trade-off between collision resistance and computational efficiency. It has been noted in the literature that using powers of 2 as the modulus allows adversarial attacks that can force many substrings to hash to the same

value, degrading performance. However, this choice is justified for our implementation because:

- Our inputs are randomly generated, not adversarially constructed
- Avoiding explicit modulo operations provides substantial speedups since explicit division modulus operations on integers are costly in terms of cpu/gpu cycles used up for division/modulus.

3 Implementation Details

3.1 String Construction

The two input strings of 100 million characters each are generated pseudo-randomly using the Mersenne Twister random number generator (`std::mt19937`). Each character is uniformly sampled from the range ASCII [65, 90], that corresponds to uppercase English letters A–Z.

To ensure reproducibility while maintaining no correlation between strings, we use a seeding strategy where string *A* is generated using a user-provided seed, while string *B* uses the reversed version of the same seed (e.g., seed 123456 produces 654321 for string *B*). This approach guarantees that results are reproducible across runs with the same seed while ensuring the two strings are statistically independent. *If the user doesnt provide a seed in the argument of the program, 123456 is used as the seed for string A, and 654321 as the seed for string B.*

3.2 CPU Implementation (OpenMP)

We implemented the Rabin-Karp algorithm on the CPU using OpenMP to parallelize the hashing and search phases. The implementation consists of three primary stages, that are each parallelized seperately:

1. **Parallel Hash Generation:** We iterate through string *A*, computing rolling hashes for every substring of length *L*. This loop is parallelized using `#pragma omp parallel for schedule(static)`. The results (hash value and start index) are stored in a flat vector of structs.
2. **Parallel Sorting:** Instead of using a hash map (to avoid complex and expensive locking mechanisms in hashmaps), we sort the vector of hashes from string *A*. We utilized the GCC parallel extension `__gnu_parallel::sort` (from `<parallel/algorithm>`).
3. **Parallel Search and Verification:** We then iterate through string *B* in parallel. For each substring of *B*, we perform a binary search (`std::lower_bound`) on the sorted hashes of *A*.
 - If a matching hash is found, we perform an explicit character-by-character comparison (`check_string_equality`) to handle hash collisions.
 - To ensure thread safety, we use an atomic boolean flag `found`. If one thread finds a match, other threads can skip further processing (`if (found) continue;`), minimizing wasted computation.

3.3 GPU Implementation (CUDA)

The GPU implementation takes a "Sort-and-Scan" approach to maximize memory bandwidth utilization. The process is as follows:

1. **Data Preparation:** We precompute the prefix hashes for strings A and B on the host and transfer them to the GPU global memory.
2. **Hash Generation:** In each iteration of the binary search on length L , a kernel `computeRollingHashes` generates rolling hashes for all substrings of length L from *both* strings. These are stored in a single unified array, `d_substringHashes`, alongside an index array `d_sortedIndices` tracking their original positions.
3. **Radix Sort (Thrust):** We utilize `thrust::sort_by_key` to sort the unified hash array. This groups identical hashes from both strings together in contiguous memory locations.
4. **Matching Kernel:** A lightweight kernel, `parallelSearch`, scans the sorted array in parallel. It checks adjacent elements:

$$\text{if } (Hash[i] == Hash[i + 1]) \wedge (\text{Origin}(i) \neq \text{Origin}(i + 1)) \quad (8)$$

If two adjacent hashes match and originate from different strings (one from A , one from B), it is flagged as a candidate match.

5. **Manual Checking:** On finding the candidate matches, we manually check them on the CPU. This could have been performed on the GPU but this was not done to avoid having to transfer the strings to GPU. For the very few matches that are found, it was deemed to not be worth it to create separate kernels for the string matching.

4 CUDA-Specific Optimizations

Here, we list the cuda-specific optimizations that were utilized keeping the architecture in mind.

4.1 Minimizing Data Transfers

Data from RAM to VRAM is minimized and happens only once at the program initialization. Only 3 arrays of 100M size each are passed: `d_hashA`, `d_hashB`, `powers` all of datatype(`uint64_t`), hence the total transfer size is **2.4GB**.

4.2 Memory Coalescing

- Our "Sort-and-Scan" approach ensures perfect memory coalescing. In the `parallelSearch` kernel, thread k accesses index k and $k + 1$. Therefore, each thread requests 2 `uint64_t` loads, one of which is common with the next thread, therefore in total, a warp requests 33 `uint64_t` loads from contiguous memory. Now, this translates to $33 * 8 = 264$ bytes of data loading. This can be serviced by the memory controller in 3 128-byte transaction, achieving peak global memory bandwidth. *Very very few threads enter the conditional branch where the hashes are same, so the extra loads are avoided almost all of the time.*

- The `substringHasher` device function preserves global memory coalescing for all input arrays. Prefix hash accesses to `hash[endIdx]` and `hash[startIdx-1]` are coalesced because consecutive threads access consecutive memory addresses, resulting in four 128-byte transaction per warp for each term. The powers array access `powers[length]` is broadcast-optimized since all threads read the same address.

4.3 Warp Divergence Minimization

We have minimized warp divergence in each cuda function that we have used, ensuring near 100% bus utilization.

- In `computeRollingHashes` and `substringHasher`, branching conditions depend on thread indices (e.g., `idx < N` or `idx == 0`). This confines divergence to a single warp at the boundary (`idx == 0` or `idx == N`), while all the warps execute purely uniform paths.
- In `parallelSearch`, the collision check `if(hash[i] == hash[i+1])` relies on the rarity of 64-bit hash collisions. For random inputs, virtually all warps evaluate this as uniformly false, allowing almost all warps to skip the collision handling logic in lock-step without serialization.

4.4 Structure of Arrays

We utilized a Structure of Arrays data layout, maintaining separate arrays for hashes and indices. This allowed `thrust::sort_by_key` to operate efficiently and enabled the scanning kernel to load only the hash values (most of the time) conserving cache lines and memory bandwidth.

4.5 Radix Sort Utilization

By using `thrust::sort` on primitive `uint64_t` types, we leveraged the library’s internal Radix Sort implementation. This provides an $O(N)$ sorting complexity (specifically $O(kN)$) which scales significantly better than comparison-based $O(N \log N)$ sorts for large datasets of 10^8 elements

4.6 Filter-and-Refine Strategy

We used a Filter-and-Refine strategy to separate the search phase from the verification phase. The GPU rapidly scans 64-bit hashes to identify potential substring matches. The CPU then performs the refinement step, verifying these candidates via exact character comparisons. This separation ensures that the GPU processing power is not wasted on verifying and checking for potential hash collisions.

5 Results

5.1 Program Outputs

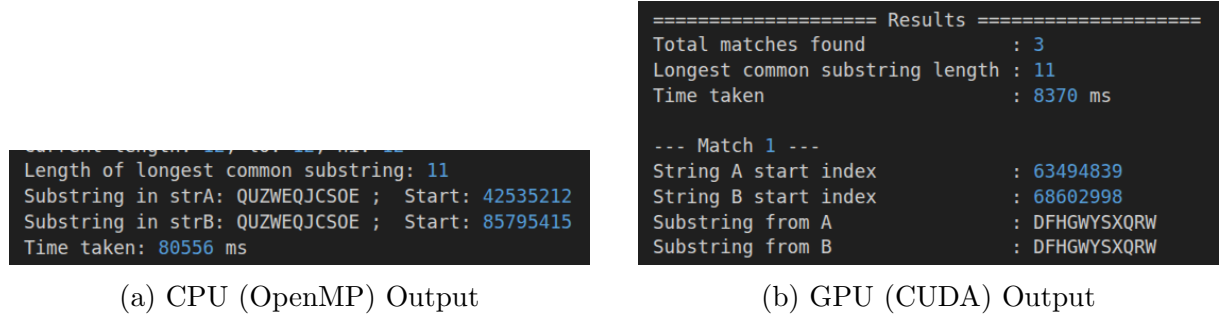


Figure 1: CPU and GPU implementation outputs

For most randomly constructed strings, our programs found that the LCS is of length 11.

5.2 CPU

The average execution times on the CPU over 5 runs for 6 different configurations are presented below:

Implementation	Time (s)	Speedup vs OpenMP (1-core)
OpenMP (1 threads)	1282	1.00x
OpenMP (2 threads)	665	1.93x
OpenMP (4 threads)	343	3.74x
OpenMP (8 threads)	181	7.08x
OpenMP (16 threads)	108	11.87x
OpenMP (32 threads)	80	16.03x

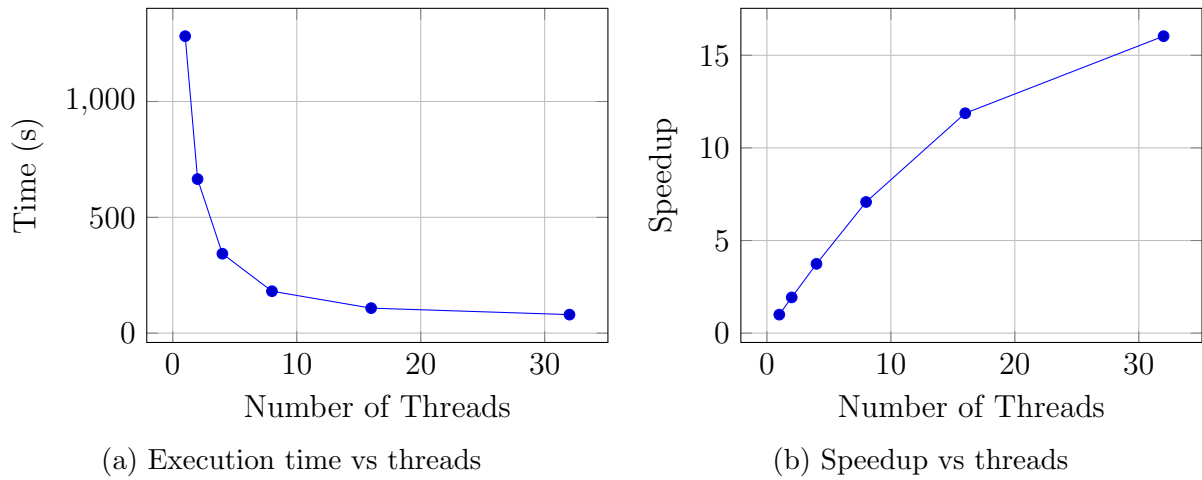


Figure 2: Performance scaling of OpenMP implementation for problem size of 100M

5.3 GPU

The runtime of the CUDA based LCS approach was 7 seconds for 100M problem size.

5.4 Speedup

The CUDA implementation completed the computation in 7 seconds(calculated average for 5 runs) for the 100M problem size. The best CPU execution time was obtained using OpenMP with 32 threads, which required 80 seconds.

The speedup of the GPU implementation over the best OpenMP execution is computed as:

$$\text{Speedup}_{\text{GPU vs best OpenMP}} = \frac{T_{\text{best OpenMP}}}{T_{\text{GPU}}} = \frac{80}{7} = 11.43\times$$

Thus, the GPU implementation achieves an approximate **11.43** \times speedup over the best-performing OpenMP configuration.

All CUDA experiments were performed using a kernel launch configuration of 256 threads per block. The number of blocks was computed dynamically based on the total number of substrings processed in the kernel:

$$\text{blocks} = \left\lceil \frac{\text{totalSubstrings} + 256 - 1}{256} \right\rceil$$

6 CUDA Profiling & Observations

Broadly, our Program consists of 3 custom kernels:

- `parallelSearch`
- `computeRollingHashes`: Just a wrapper for `substringHasher` device function

6.1 Memory Transfers v/s Computation

First, we include the overall split of execution time of the kernels. Note that the `DeviceRadixSortDownsw` and `DeviceRadixSortUpsweepKernel` are the built-in kernels for sorting in the thrust library.

▶ CPU (72)
▼ CUDA HW (0000:18:00.0 - NVIDIA RTX A5000)
▼ 92.5% Kernels
▶ 74.3% DeviceRadixSortDownsweepKernel
▶ 19.0% DeviceRadixSortUpsweepKernel
▶ 4.3% computeRollingHashes
▶ 1.8% parallelSearch
▶ 0.4% _kernel_agent
▶ 0.2% RadixSortScanBinsKernel
▼ 7.5% Memory
<0.1% Memset
>99.9% HtoD memcpy
<0.1% DtoH memcpy
▶ Threads (9)

Figure 3: Overall Kernel Time Split

Memory Transfers is only around 7.5% of the total execution time. While majority(92.5%) of the time is spent on actual computation.

Among the 7.5%, as expected, most of it is used up for host to device transfer(transfer of prefix hashes and powers array) and a very miniscule amount of memory is transferred back from the device to the host. This indicates that the memory transfers triggered by hash collisions are very less in number, as was predicted.

This demonstrates efficient use of the GPU's time and minimization of Memory transfers.

6.2 Kernel wise Time Split

Now, we observe the split of the time spent on each kernel:

- **DeviceRadixSortDownsweepKernel**: 74.3% of GPU time
- **DeviceRadixSortUpsweepKernel**: 19.0% of GPU time
- **computeRollingHashes**: 4.3% of GPU time
- **parallelSearch**: 1.8% of GPU time
- **_kernel_agent**: 0.4% of GPU time
- **RadixSortScanBinsKernel**: 0.2% of GPU time

Observe that the biggest bottleneck by far is the sorting kernels. This indicates that we have successfully reduced the problem to essentially a sorting problem. And we used the fastest tool available to us for this task(`thrust::sort_by_key`).

6.3 computeRollingHashes Kernel

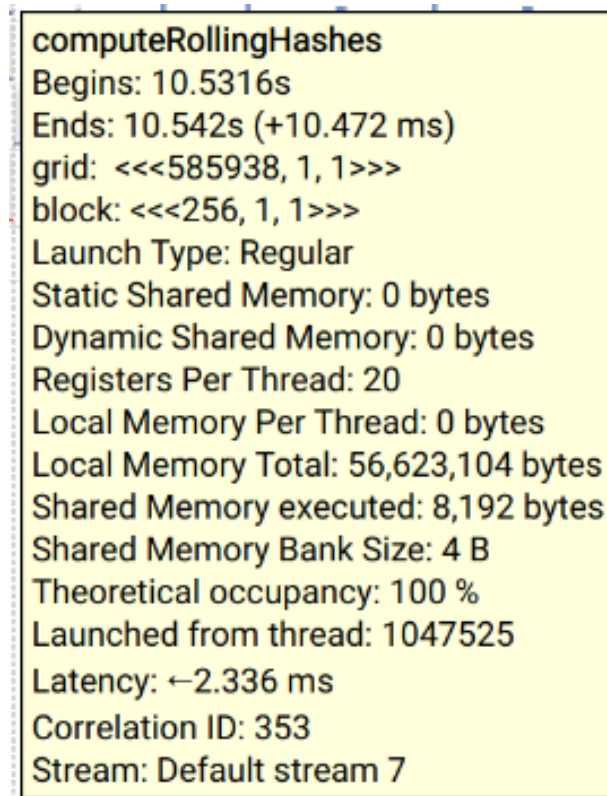


Figure 4: computeRollingHashes Details

The profiling data for `computeRollingHashes` confirms a highly optimized implementation, achieving 100% theoretical occupancy.

Additionally, 0 bytes of Local Memory usage and 0 bytes of shared memory means that memory operations are minimal and each thread don't maintain any memory for itself, and cross thread communication is minimal. This is expected since this kernel and the function that it is a wrapper for (`substringHasher`) only do simple arithmetic operations.

6.4 parallelSearch Kernel

```
parallelSearch
Begins: 11.7721s
Ends: 11.7769s (+4.818 ms)
grid: <<<769043, 1, 1>>>
block: <<<256, 1, 1>>>
Launch Type: Regular
Static Shared Memory: 0 bytes
Dynamic Shared Memory: 0 bytes
Registers Per Thread: 16
Local Memory Per Thread: 0 bytes
Local Memory Total: 56,623,104 bytes
Shared Memory executed: 8,192 bytes
Shared Memory Bank Size: 4 B
Theoretical occupancy: 100 %
Launched from thread: 1047525
Latency: ←654.234 µs
Correlation ID: 1400
Stream: Default stream 7
```

Figure 5: parallelSearch kernel details

The profiling data for parallelSearch confirms the efficiency of the "Sort-and-Scan" strategy, executing in just 4.8 ms.

The profiling data for parallelSearch is also highly optimized implementation, achieving 100% theoretical occupancy.

Similarly, just like the last kernel, 0 bytes of Local Memory usage and 0 bytes of shared memory means that memory operations are minimal and each thread don't maintain any memory for itself, and cross thread communication is minimal. This is also expected since this kernel just performs matching between consecutive elements of an array. It does no computation most of the time and only on a very few occasions, on finding some match, it performs some extra computation.

6.5 Overall Run Pattern

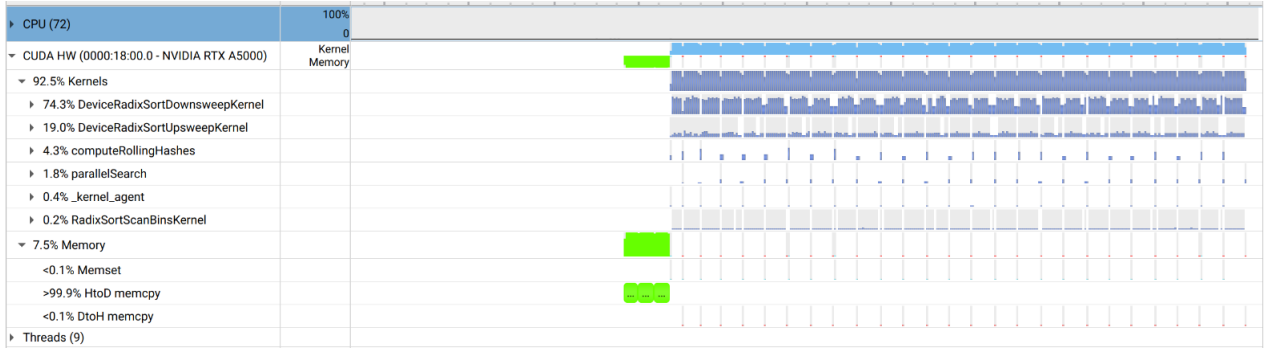


Figure 6: Overall Run Pattern

As expected, our two kernels run only once in every iteration of the binary search operation. And the memory transfers from Host to Device, occur only once at initialization.

7 Conclusion

By using the Rabin-Karp rolling hash algorithm with a "Sort-and-Scan" strategy, we achieved an $11.43\times$ speedup over the best CPU implementation (32-thread OpenMP).

Profiling shows that memory transfers constituted only 7.5% of execution time, with 92.5% spent on computation, primarily on sorting operations. Both custom kernels (`computeRollingHashes` and `parallelSearch`) achieved 100% theoretical occupancy with zero local memory usage per thread, demonstrating effective utilization of GPU architecture through memory coalescing and divergence minimization.

The implementation has essentially reduced the LCS Problem to an sorting task, using Thrust's radix sort to handle the computational bottleneck.

Notice that we relied heavily on the structure of the data(which is random). Had this been a different kind of string like DNA Sequence, it would not have been possible to use this particular structure of the program. For example for DNA Sequences where patterns repeat many times, the threads in our program will diverge many times since hashes will match often. This would slow down our program significantly. Our particular approach only does as well as it does, because it was designed keeping the random structure of the data in mind.