CS 508 Project Report

Muchen Xu (muchenx2), Lan Zhang (lz31), Jiaai Xu (jiaaixu2), Yanlong Yao (yy48)

May 9, 2023

1 Motivation

The importance of multi-pattern matching algorithms spans over various applications, such as email spam filtering, bio-informatics, and academic plagiarism detection.

In email spam filtering, these algorithms enable more sophisticated spam detection by identifying patterns in email content, enhancing communication security and reducing the burden on users. In bio-informatics, multi-pattern matching algorithms facilitate the analysis of vast and complex datasets, such as DNA, RNA, and protein sequences, improving the insights of gene functions, regulation, and the molecular basis of diseases. Additionally, in academic plagiarism detection, these algorithms play a crucial role in maintaining academic integrity by efficiently comparing large text collections to identify potential plagiarism.

Overall, multi-pattern matching algorithms contribute significantly to advancing knowledge and addressing critical challenges across various fields by enabling the efficient and accurate identification of patterns within large datasets.

2 Related Works

There are two related works in this field we are going to compare our implementation with. One is a Hybrid Parallel Implementation of the Aho-Corasick and Wu-Manber Algorithms [2], published in 2014, where in this paper, it utilized Aho-Corasick and Wu-Manber algorithms on GPU to accelerate the computation of multi-pattern matching. However, its implementation of finding a pattern in the Aho-Corasick data structure is not efficient.

The other is A Rabin-Karp Implementation [3], published in 2020. This paper utilizes Rabin-Karp implementation for multi-pattern matching problem. They proposed a Prefix-Sum Based Parallel Rabin-Karp algorithm, tailored for parallel execution on the GPU. At its core, it used a fast-parallel prefix-sums algorithm to maximize parallelization together with a look-up table to accelerate the task of matching on multiple patterns. Although this paper presented a better performance than the previous experiments, our implementation can still outperform. We focused on the time spent during the computation on GPU, as other time spent on preprocessing and postprocessing on CPU is negligible. The detail measurement is presented in Section 6 Evaluation and Results.

3 Problem Definition

The problem, multi-pattern matching, can be formally defined as follows,
Given:

An **Alphabet** of size C,

A target **Text** of length L, only contain characters from the alphabet,

A set of **Patterns** of size N, each pattern consisting of M characters,

Report:

Occurrences of each pattern in the target text string.

Example:

Alphabet = ['a', 'b', 'c', 'd'], where $C = 4$

Text = "abcacababc", where $L = 10$

Patterns = ["ab", "ca", "da", "bc"], where $N = 4, M = 2$

Output:

Occurrences = {"ab": 3, "ca": 2, "da": 0, "bc": 2}

4 Baseline

4.1 Parallelization

Given a text of length L, N patterns with each pattern of length M, and the tile size T, there are total $\lceil \frac{L}{T} \rceil$ threads, each thread handles the substring of text of length $T + M - 1$. Each thread counts the occurrence of the patterns in this subtext of length $T + M - 1$. The reason why each thread computes an extra $M - 1$ inputs is illustrated in the figure below. Figure 1 shows an example with $L = 30$, $M = 4$ and $T = 5$. If a thread does not take the extra $M - 1$ inputs and a pattern is located from position 3 to 6, then neither thread 0 and thread 1 can recognize this pattern as thread 0 only knows `Text[0:4]` and thread 1 only knows `Text[5:9]`.

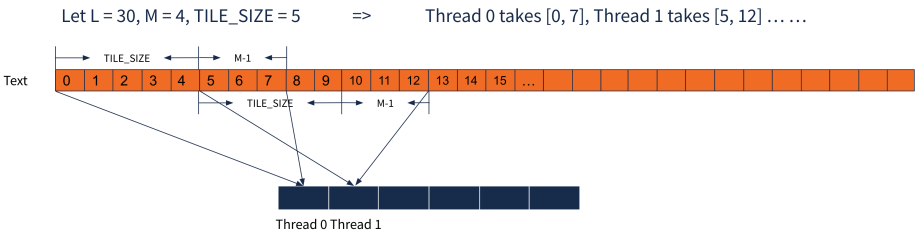


Figure 1: Parallelization

4.2 Aho-Corasick Finite State Machine

We implemented a Aho-Corasick finite state machine to efficiently find a pattern. Each state represents a prefix of a pattern. Each operation is to append a character to the string of the current state. On receiving an character, the transition will move the current state to the state presenting longest suffix of the current one. For example, for the previous patterns in section 3, patterns = ["ab", "ca", "da", "bc"], Figure 2 below shows an incomplete finite state machine, where for state 5:

1. If append operation 'a' to state 5, "ab" become "aba", the longest suffix in the FSM is 'a' (by discarding "ab"), so it points to state 1.

2. If append operation 'b' to state 5, "ab" become "abb", the longest suffix in the FSM is 'b' (by discarding "ab"), so it points to state 2.

3. If append operation 'c' to state 5, "ab" become "abc", the longest suffix in the FSM is "bc" (by discarding 'a'), so it points to state 6.

4. If append operation `'d'` to state 5, `"ab"` become `"abd"`, the longest suffix in the FSM is `'d'` (by discarding `"ab"`), so it points to state 4.

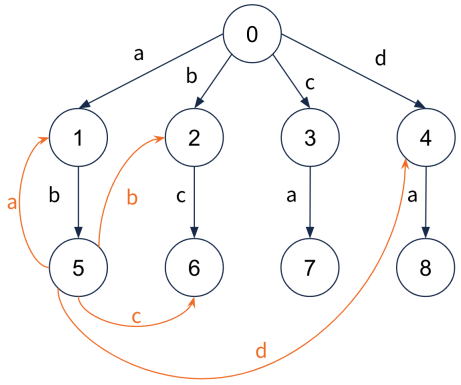


Figure 2: Incomplete Aho-Corasick Finite State Machine

The final complete finite state machine is shown in table 1. A CPU implementation of building this data structure is shown in Appendix A.

State	add 'a'	add 'b'	add 'c'	add 'd'
State 0	1	2	3	4
State 1	1	5	3	4
State 2	1	2	6	4
State 3	7	2	3	4
State 4	8	2	3	4
State 5	1	2	6	4
State 6	7	2	3	4
State 7	1	5	3	4
State 8	1	5	3	4

Table 1: Complete Aho-Corasick Finite State Machine

4.3 Baseline GPU Kernel

```
template <int charSetSize, int TILE_SIZE, int BLOCK_SIZE>
__global__ void ACGPUSimple(const int* tr, const unsigned char* text,
                            int* occur, const int M, const int L) {
  int idx         = blockIdx.x * blockDim.x + threadIdx.x;
  int threadStart = idx * TILE_SIZE;
  int threadEnd   = threadStart + TILE_SIZE + M - 1;

  int state = 0;
  for (int i = threadStart; i < threadEnd && i < L; i++) {
    state = tr[state * charSetSize + text[i]];
    atomicAdd(&occur[state], 1);
  }
}
```

Listing 1: Baseline Kernel

Listing 1 is the baseline GPU kernel, the input `cont int* tr` is the Aho-Corasick finite state machine, `const unsigned char* text` is the input text, `int* occur` is the output occurrence, `const int M` is the length of a pattern, `const int L` is the length of the text. First, it computes the index of the thread and based on the index, compute the starting point and end point. Then, it loops over the input text. In each loop, it is given a current state and an operation `text[i]`, then look up the next state from the finite state machine `tr`. Then, atomically add one to the occurrence of the state.

5 Optimizations

5.1 Shared Memory for Output

Given the atomic addition in the kernel, a natural thought to optimize it will be using the shared memory to alleviate memory access latency.

```
template <int charSetSize, int TILE_SIZE, int BLOCK_SIZE, int GPUbinSize>
__global__ void ACGPUSharedMemBranchFree(const int* tr, const unsigned char* text,
                                         int* occur, const int M, const int L,
                                         const int trieNodeNumber) {
  // ...

  extern __shared__ int localOccur[];
  for (int i = threadIdx.x; i < trieNodeNumber; i += blockDim.x)
    localOccur[i] = 0;
  __syncthreads();
  int state = 0;
  for (int i = threadStart; i < threadEnd && i < L; i++) {
    state = tr[state * charSetSize + text[i]];
    atomicAdd(&localOccur[state], 1);
  }
  __syncthreads();
  for (int i = threadIdx.x; i < trieNodeNumber; i += blockDim.x)
    atomicAdd(&occur[i], localOccur[i]);
}
```

Listing 2: Optimization 1. Assume `occur` fits into shared memory and thus be branch-free.

Assume the `occur` array fits into the shared memory. We can store all occurrences of a block first to the shared memory, and then write them back to global memory at one time, as illustrated in Listing 2. Such organization leverages on hierarchical memory speed and hence give better performance.

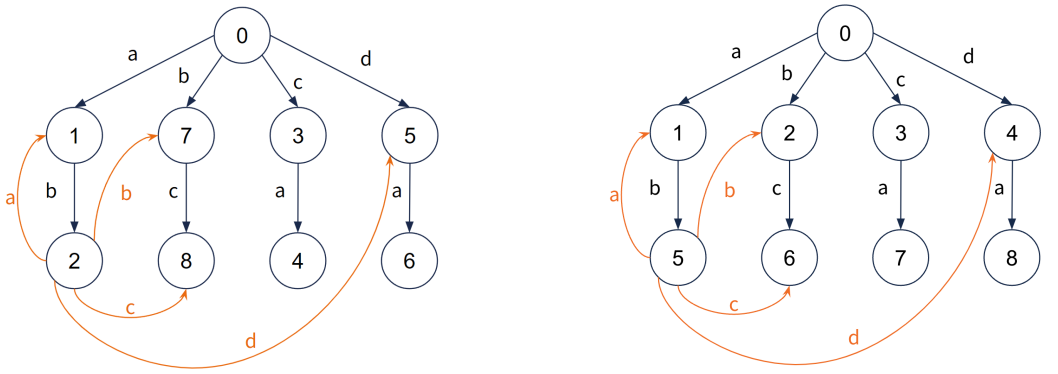
In case of `occur` exceeds the maximum shared memory size, we make the shared memory to handle the first `GPUbinSize` slot in `occur`, and for the rest we directly add to global memory, as illustrated in Listing 3.

```
// ...
int state = 0;
for (int i = threadStart; i < threadEnd && i < L; i++) {
  state = tr[state * charSetSize + text[i]];
  if (state < GPUbinSize) // Introduces control divergence here
    atomicAdd(&localOccur[state], 1);
  else
    atomicAdd(&occur[state], 1);
}
// ...
```

Listing 3: Optimization 1. If `occur` exceeds the maximum shared memory size, use GPU bin to handle the first `GPUbinSize` slot.

5.1.1 Reorder State Machine for Cache-friendly Memory Access Pattern

For the program shown in Listing 3, in order to improve the usage rate of shared memory as well as to decrease the hurt from control divergence. We tried to reorder the node index in the Trie (dictionary tree, the underlying data structure of Aho-Corasick state machine), based on the intuition that the node in shallow layer will more likely be accessed, regarding the nature of Aho-Corasick finite state machine. A comparison between the Trie before and after the reordering is visualized in Figure 3.



(a) A typical Trie whose node is labeled in order of added patterns (`ab`, `ca`, `da`, `bc`).

(b) Reorder the Trie nodes that shallower node has smaller index.

Figure 3: Visualized Trie (dictionary tree) before and after the reordering

We expect the shallower node to be accessed more frequently, and hence increase the hit rate inside GPU bin. We believe this approach also alleviates the control divergence at some degree, because the first branch are more likely to be taken.

5.2 Shared Memory for Coalesced Memory Access

```
template <int charSetSize, int TILE_SIZE, int BLOCK_SIZE>
__global__ void ACGPUCoalecedMemRead(const int* tr, const unsigned char* text,
                                     int* occur, const int M, const int L) {
  // ...

  extern __shared__ unsigned char localText[];
  for (int i = threadIdx.x; i < blockDim.x * TILE_SIZE + M - 1; i += blockDim.x)
    localText[i] = text[idx + i];
  __syncthreads();

  int state = 0;
  for (int i = threadStart; i < threadEnd && idx + i < L; i++) {
    state = tr[state * charSetSize + localText[i]];
    atomicAdd(&occur[state], 1);
  }
}
```

Listing 4: Optimization 2. Shared Memory for Coalesced Memory Access.

We implement this optimization because the memory access for the global text is not coalesced in running the state machine. In Listing 4, we try to use all the threads to copy the text from the global text to local shared memory text for this basic block. The total number of memory reads of this optimization is also reduced due to the overlapping $M-1$ characters between tiles. The memory read now is coalesced, it performs better than baseline.

5.3 Compact Memory

```
struct int2x4_t {
  unsigned char q0 : 2;
  unsigned char q1 : 2;
  unsigned char q2 : 2;
  unsigned char q3 : 2;
};
```

Listing 5: Compact Text Struct.

```
template <int charSetSize, int TILE_SIZE>
__global__ void ACGPUCompactMem(const int* tr, const int2x4_t* text_conpact,
                                int* occur, const int M, const int L) {
  int idx         = blockIdx.x * blockDim.x + threadIdx.x;
  int threadStart = idx * TILE_SIZE;
  int threadEnd   = threadStart + TILE_SIZE + (M - 1) / 4;

  int state = 0;
  for (int i = threadStart; i < threadEnd && i < L / 4; i++) {
    state = tr[state * charSetSize + text_conpact[i].q0];
    atomicAdd(&occur[state], 1);
    state = tr[state * charSetSize + text_conpact[i].q1];
    atomicAdd(&occur[state], 1);
    state = tr[state * charSetSize + text_conpact[i].q2];
    atomicAdd(&occur[state], 1);
    state = tr[state * charSetSize + text_conpact[i].q3];
    atomicAdd(&occur[state], 1);
  }
  // If [threadStart, threadEnd - 1] is not enough to process the last 4 characters
  if (threadEnd < L / 4) {
    if ((M - 1) % 4 >= 1) {
      state = tr[state * charSetSize + text_conpact[threadEnd].q0];
      atomicAdd(&occur[state], 1);
    }
    if ((M - 1) % 4 >= 2) {
      state = tr[state * charSetSize + text_conpact[threadEnd].q1];
      atomicAdd(&occur[state], 1);
    }
    if ((M - 1) % 4 >= 3) {
      state = tr[state * charSetSize + text_conpact[threadEnd].q2];
      atomicAdd(&occur[state], 1);
    }
  }
}
```

Listing 6: Optimization 3. Compact Memory Kernel.

We notice that sometimes the size of the character set is small. For example, if the size of the character set is 4, we only need 2 bit to store one bit character, but one C++ char has 8 bit. The solution is to use bit-field struct (see Listing 5). The size of struct `int2x4_t` is 1 byte, 8 bit. It has 4 2-bit elements. We also need to modify the kernel (see Listing 6) where `text_conpact` is an array of element type `int2x4_t`. The threadEnd and for loop condition changes. In one iteration, we read one element from `text_compact` and then travel four states and store the output. We need to be careful about the boundary conditions. When $(M-1)$ is not divided by 4, we need special handing to deal with the last element. The run time of the kernel is dominated by IO (read global trie and text, use atomicAdd to store the output occurrence). Therefore, this optimization could improve the performance because it reduces global memory access to text to one quarter compared to the baseline. We use `int2x4_t` as

an example here. In practice, we use `int1x8_t`, `int2x4_t` and `int4x2_t` to match different character set size.

5.4 Only Count Leaf Nodes

```
template <int charSetSize, int TILE_SIZE, int BLOCK_SIZE>
__global__ void ACGPUEqLength(const int* tr, const unsigned char* text, int* occur,
                 const int N, const int M, const int L, const int trieNodeNumber) {
  ...
  int leafStart   = trieNodeNumber - N;

  extern __shared__ int localOccur[];
  for (int i = threadIdx.x; i < N; i += blockDim.x)
    localOccur[i] = 0;
  __syncthreads();
  int state = 0;
  for (int i = threadStart; i < threadEnd && i < L; i++) {
    state = tr[state * charSetSize + text[i]];
    if (state >= leafStart)
      atomicAdd(&localOccur[state - leafStart], 1);
  }
  __syncthreads();
  for (int i = threadIdx.x; i < N; i += blockDim.x)
    atomicAdd(&occur[i], localOccur[i]);
}
```

Listing 7: Optimization 4. Only Count Leaf Nodes when Pattern Length is the Same.

Another important point in our problem setting is that all of the patterns have the same length `M`. In the Aho-Corasick algorithm, we need to store the occurrence of all states and add that number in the topology order of the failing tree to get the final occurrence of the patterns. However, when all patterns have the same length, these patterns are exactly the leaf nodes in the trie. Therefore, we use a similar reordering process to make all the leaf nodes at the end of state machine. Because states that are near the root of the trie are more likely to be accessed, this modification can reduce a large fraction of global memory writes. This kernel (Listing 7) is based on Optimization 1 (Listing 2). That is because by only counting the leaf nodes, we can reduce the length of the output occurrence array, and therefore we can assume that the occurrence fits into the shared memory.

6 Evaluation and Results

6.1 Comparison with the State-of-the-Art

We also compare our results with the state-of-the-art repositories and papers. After searching for papers of multiple pattern matching problem, we find one paper [2] with released source code. They use a hybrid parallel implementation of the Aho-Corasick and Wu-Manber algorithms. We downloads its source code and performs the evaluation on the same machine (RAI). The experiment settings are $C = 4, M = 8, L = 2^{30}$. The result (see Table 2) shows that our implementation runs 4.00x (Opt 1) and 7.90x (Opt 4) faster than their algorithm (geometric average). We notice that when `N` gets larger, the run time of our algorithm is not dominated by `N`. That is because when `N` is small, we have small number of states and hence guarantees compact memory access. When `N` gets larger, we get sparse memory access. When reaching the limit of the memory bandwidth, the run time converges to some number.

	N = 1000	N = 4000	N = 8000	N = 12000	N = 16000
Aho-Corasick and Wu-Manber	141.254	159.973	167.325	OOM	OOM
Opt 1 (Listing 3)	28.5521	38.9457	53.1077	51.8932	50.0971
Opt 1 improvement	4.948x	4.108x	3.151x	/	/
Opt 4	8.37014	28.3115	32.3461	30.6248	OOM
Opt 4 improvement	16.88x	5.650x	5.173x	/	/

Table 2: The performance evaluation of Aho-Corasick and Wu-Manber algorithm and our implementation (both running on RAI).

Lucas Nunes et al. propose a pattern matching algorithm based on Rabin-Karp [3] on 2020. They don't publish the source code, so we compare with the performance they mentioned in the paper. They use Tesla V100 GPU while we use TITAN V (RAI). Considering that the Tesla V100 is more powerful than TITAN V (see Table 3), this comparison is fair. The experiment settings are $C = 2, L = 2^{27}$, `N` is from 4 to 256 and `M` is from 10 to 30. Table 4 shows that our implementation gets 5.18x improvement (geometric algorithm among all configurations) against Rabin-Karp implementation.

	Global Memory	Shared Memory / SM	Registers / SM
Tesla V100	16 GB	96 KB	256 KB
TITAN V	12 GB	48 KB	64 KB
	GPU Boost Clock Rate	Peak FP32 TFLOPS	Peak FP64 TFLOPS
Tesla V100	1530 MHz	15.7	7.8
TITAN V	1455 MHz	12.3	6.1

Table 3: Tesla V100 vs. TITAN V.

Runtime (ms)	Rabin-Karp			Our Implementation (Opt 4)		
	M = 10	M = 20	M = 30	M = 10	M = 20	M = 30
N = 4	3.73	5.07	5.08	0.601	0.760	0.891
N = 16	4.27	5.12	5.08	0.692	0.860	1.02
N = 64	4.90	5.14	5.12	0.850	1.04	1.29
N = 256	5.43	5.22	5.18	1.01	1.25	1.51

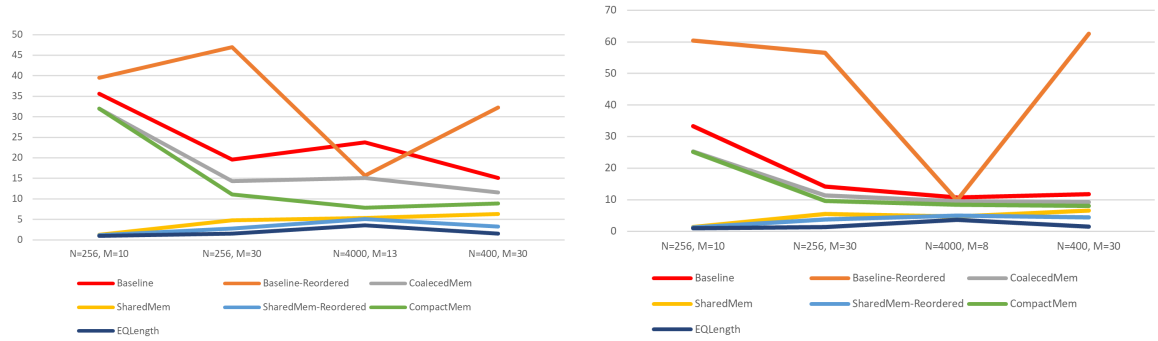
Table 4: The performance evaluation of Rabin-Karp algorithm (based on data provided in the paper) and our implementation (running on RAI).

6.2 Ablation Study

We ran variants of kernels on different circumstances as shown in Table 5, and try to reason about why some optimization are successful. The test cases are designed in two divisions, $C = 2$ and $C = 4$. And first two column of each means AC finite state machine is small. And another two column means AC finite state machine is large, i.e. close to the maximum size of shared memory. Each group is then divided into two test cases: first for shallow tree, where the tree is nearly complete, and the second for deep tree, this means tree is sparse regarding the depth. At the mean time, we keep the number of node on Trie relatively close for big cases. And we use $L = 2^{27}$ for text length. Note that red line is Baseline, instead of above orange line. The "EQLength" kernel refers to the optimization described in Section 5.4, because it's a special case where all patterns assumed to have equal length.

C	2				4			
N	256	256	4000	400	256	256	4000	1500
M	10	30	13	30	10	30	8	12
TrieNodeNumber	824	5946	9438	9013	1763	6855	11289	11307
Baseline	35.626	19.583	23.758	15.068	33.324	14.151	10.653	14.156
Baseline-Reordered	39.520	46.945	15.714	32.247	60.356	56.525	9.803	16.873
CoalecedMem	31.968	14.307	15.044	11.606	25.278	11.374	9.471	11.437
SharedMem	1.231	4.739	5.333	6.308	1.334	5.442	4.650	5.166
SharedMem-Reordered	1.183	2.749	5.038	3.226	1.242	3.780	4.931	5.786
CompactMem	31.992	11.041	7.826	8.868	25.110	9.629	8.392	9.680
EQLength	1.018	1.502	3.576	1.552	0.943	1.403	3.573	1.490

Table 5: Comparison of kernels' performance under different circumstance, measured in millisecond.



(a) Kernel Performance when $C = 2$.

(b) Kernel Performance when $C = 4$.

Figure 4: Visualized kernel performance comparison.

Reordering: Comparing "Baseline" and "Baseline-Reordered", we observe that reordering the Trie will not always give a improvement. We attribute this to the fact that reordering destructs the original spatial locality: in a typical Trie, for a pattern who shares little common prefix with others, the nodes representing suffix will have successive index. But after reordering, such spatial locality is destroyed and thus give non-coalesced memory access. However, reordering doesn't give performance hurt when applied to "SharedMem" kernel, comparing "SharedMem" and "SharedMem-Reordered".

Coalesced Memory Read: In 'CoalescedMem' kernel, text string is coalescedly fetched to shared memory, and reduced the memory fetch by sharing the overlapping characters. Thus we believe this optimization can safely be combined with any kernel, as long as shared memory is not exhausted.

Shared Memory for occurrence counting: It is the optimization that gives us major improvement. It leverages on hierarchical memory speed by gathering all result first to shared memory and than feed back to global memory. We observed that reordering seems always to give improvement to this optimization.

Compact Memory: At first we did not expect this to be a useful optimization, as we are introducing so many extra instructions as overhead. But the result seems that the compacted memory decreases the memory latency, which is far more expensive than the introduced instruction time. We believe this optimization can also safely be combined with any kernel.

Only Count Leaf Nodes: This optimization gives extra improvements, due to less `atomicAdd` usage. There's no doubt that it will prune about $(1 - \frac{1}{M}) \times 100\%$ unimportant memory access, though

such pruning may be hidden by the control divergence.

Considering all analysis above, we can expect the optimal kernel may be given by combining coalesced and compact memory access, and using shared memory to count the occurrence of the leaf nodes.

7 Conclusions

In this project, we implemented and parallelized a multiple pattern matching algorithm, Aho-Corasick, on GPU, and applied plenty of optimizations to it. Our best kernel achieved a geometric average speedup of 7.90x compared to the algorithm [2] proposed in 2014 by Kouzinopoulos et al., and 5.18x compared to the algorithm [3] proposed in 2020 by Nunes et al.. These results demonstrate the effectiveness of our approach in accelerating pattern matching on GPUs. We release the source code on Github [1].

References

[1] Cs508-project-ac-gpu: Aho-corasick algorithm running on gpu. https://github.com/Scarlet1ssimo/CS508-Project-AC-GPU.

[2] Kouzinopoulos, C. S., Assael, J.-A. M., Pyrgiotis, T. K., and Margaritis, K. G. A hybrid parallel implementation of the aho-corasick and wu-manber algorithms using nvidia cuda and mpi evaluated on a biological sequence database, 2014.

[3] Nunes, L. S. N., Bordim, J. L., Ito, Y., and Nakano, K. A rabin-karp implementation for handling multiple pattern-matching on the gpu. *IEICE Trans. Inf. Syst. 103-D* (2020), 2412–2420.

A CPU Code for Building Aho-Corasick State Machine

```
int TrieBuildCPU(unsigned char* const* patterns, int* tr, int* idx,
                 const int M, const int N, const int charSetSize) {
  int trieNodeNumber = 1;
  for (int i = 0; i < N; i++) {
    int state = 0;
    for (int j = 0; j < M; j++) {
      int c = patterns[i][j];
      if (!tr[state * charSetSize + c])
        tr[state * charSetSize + c] = trieNodeNumber++;
      state = tr[state * charSetSize + c];
    }
    idx[i] = state;
  }
  return trieNodeNumber;
}

void ACBuildCPU(int* tr, int* fail, const int charSetSize) {
  queue<int> q;
  for (int c = 0; c < charSetSize; c++) {
    int state = tr[0 * charSetSize + c];
    if (state) {
      fail[state] = 0;
      q.push(state);
    }
  }
  while (!q.empty()) {
    int u = q.front();
    q.pop();
    for (int i = 0; i < charSetSize; i++) {
      auto& v = tr[u * charSetSize + i];
      if (v) {
        fail[v] = tr[fail[u] * charSetSize + i];
        q.push(v);
      } else
        v = tr[fail[u] * charSetSize + i];
    }
  }
}
```