Faster Multiple Pattern Matching System on GPU based on Bit-Parallelism

Hirohito Sasakawa

Graduate School of IST, Hokkaido University
N14, W9, Sapporo, 062-0033, Japan
sasakawa@ist.hokudai.ac.jp

Hiroki Arimura

Graduate School of IST, Hokkaido University
N14, W9, Sapporo, 062-0033, Japan
arim@ist.hokuda.ac.jp

Abstract— In this paper, we propose fast string matching system using GPU for large scale string matching. The key of our proposed system is the use of bit-parallel pattern matching approach for compact NFA representation and fast simulation of NFA transition on GPU. In the experiments, we show the usefulness of our proposed pattern matching system.

I. Introduction

By rapid growth of network infrastructure and sensor technologies, massive amount data have emerged in various fields. The **large-scale pattern matching problem** is one of the most important problems, where a pattern matching system has to work with a large number of input pattern against the huge amount of data. These problems are, however, quite CPU-intensive tasks and it is difficult for a software on CPU to efficiently process massive data streams real time. Therefore, researches on programmable hardware such as GPU have attracted increasing attention for recent years.

GPU (Graphics Processing Unit) is a special hardware for graphics processing, has several hundreds of processing units and high memory band width.

In this paper, we propose fast pattern matching system, **RunHNFA** using GPU for very long and a large number of extended pattern. To handle very long patten whose NFA does not fit into a computer word, we use the special NFA called a **hierarchical NFA (HNFA)**. HNFA consists of a collection of small NFAs, each of which is encoded by a bit-vector stored in a computer word, and organized into a hierarchical structure. The key of this system is the use of fast bit-parallel simulation [6] of transition of HNFA. Finally, we ran experiments on the amino-acid sequence data to evaluate efficiency of the proposed system. In the experiments, our proposed pattern matching system achieves 2 times faster than pattern matching algorithm on CPU.

Related works: We explain a recent research of large-scale string matching. Lin and Tsai [3] proposed an extension of Aho-Corasick algorithm [1] for GPU. This algorithm supports exact pattern matching for multiple patterns. Wu, Diao, and Rizvi [10] gave pattern matching algorithm for complex event processing. Margara and Cugola [5] proposed complex event processing engine on GPU.

II. Preliminaries

A. Extended pattern matching

Let Σ be a finite alphabet of **letters**. A **text** of length n on Σ is a sequence $T = t_1 \cdots t_n \in \Sigma^*$. In what follows, $\equiv$ means the notational equivalence. The target pattern that our proposed system deals with is the class of **extended patterns** defined as follows.

Definition 1 (Extended pattern) *The class of* **extended pattern** *is a subclass of regular expressions defined as follows: an extended pattern $P = a_1 \cdots a_m$ $(m \geq 0)$, where for each $1 \leq i \leq m$, a_i is an expression, called a* **component***, with one of the following forms:*

1. *A* **letter** $a_i = a \in \Sigma$ *is a component with the language by $L(a) = a$.*
2. *A* **don't care** $a_i = .$ *is a component with the the language by $L(a) = \Sigma$. This matches any letter in Σ.*
3. *A* **class of letters** $a_i = \beta$ *is a component, where $\beta \subseteq \Sigma$, with the language $L(\beta) = \beta$. As notation, we write $[ab\cdots]$ for $\beta = \{a, b, \ldots\}$. Note that a letter $a \in \Sigma$ and a don't care symbol '.' are a class of letters.*
4. *An* **optional letter** $a_i = \beta?$ *is a component, where $\beta \subseteq \Sigma$ is a class of letters, and $\beta? \equiv (\beta|\varepsilon)$*
5. **Bounded repeats** $a_i = \beta\{x, y\}$, $a_i = \beta\{, y\}$, $a_i = \beta\{x\}$ *are components with equivalence $\beta\{x, y\} \equiv (\beta?)^{y-x}\beta^x$, $\beta\{y\} \equiv (\beta?)^y$, $\beta\{x\} \equiv (\beta)^x$, respectively, where $\beta \in \Sigma$ and $x \leq y$.*
6. **Unbounded repeats** $a_i = \beta^*$ *and* $a_i = \beta^+$ *are components, where $\beta \in \Sigma$ is a class of letters, and $\beta^+ = \beta\beta^*$.*

For $P = a_1 \cdots a_m$, we define its language by $L(P) = L(a_1) \cdots L(a_m)$ and size of P is defined as m.

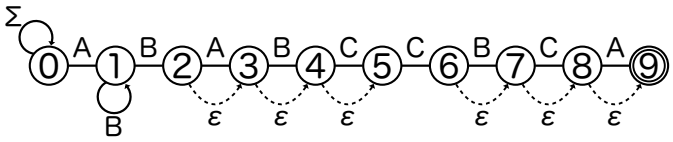

Fig. 1. The NFA of P_1 on Example 1

Example 1 *For finite alphabet* $\Sigma = \{A, B, C\}$, $P_1 = AB^+A?B?C?CB?C?A?$ *is an example of the extended pattern.*

Finally, we define multiple extended pattern matching problem as below.

Definition 2 (Multiple extended pattern matching problem) *The* **multiple extended pattern matching problem** *is, given an extended pattern set* $\mathcal{P} = \{P_1, \cdots, P_k\}$ *and an input text* T *of length* n*, to output the set of all end position of a pattern* P_i *in* T *for* $i = 1, \ldots k$

B. Extended SHIFT-AND algorithm

Bit-parallelism is a faster calculation method using inresister parallelism, which takes advantage of the intrinsic parallelism of the bitwise and arithmetic operations inside a computer word. It is also called SWAR (SIMD Within A Register), or broadword computation [8]. The extended SHIFT-AND algorithm, proposed by Navarro and Raffinot [6, 7], is a pattern matching algorithm for an extended pattern. In the preprocessing phase of this algorithm, we construct **non-deterministic finite automaton** (NFA for short) corresponding to input extended pattern P, and then, in the runtime phase, we search for all the occurrence of P in an input text T using bit parallel simulation of NFA transition. Figure 1 is an NFA of pattern P_1on Example 1. The extended SHIFT-AND algorithm runs in $O(m+|\Sigma|)$ time for preprocessing phase, $O(n\lceil \ell/w \rceil)$ time for runtime phase, and it consumes $(2|\Sigma| + 4)\lceil \ell/w \rceil$ word for memory space, where w is the bit length of registers and ℓ is the number of states of NFA corresponding to input pattern P.

C. Model of computation

As the model of computation, we assume a **unit-cost RAM** with word length w. For any bitmask length s, a **bitmask** is a bit sequence $X = b_s \cdots b_1 \in \{0, 1\}$ of *ell* bits. For bitmask with $\ell \leq w$, we assume that the following Boolean and arithmetic operations are performed in $O(1)$ time: **bitwise AND** "&", **bitwise OR** "|", **bitwise NOT** "∼", **bitwise XOR** "∧", **right shift** "<<", **left shift** ">>", **integer addition** "+", **integer subtraction** "−". The space complexity is measured in the number of words.

D. GPU

Graphics processing unit (GPU) is a special hardware for graphics processing. It has several sets of processing unit, called **stream multi-processor** (SM), and each SM has Q SIMD processing core and **shared memory** of size M which can access in low-latency. For example, a high-end GPU, Nvidia GeForce GTX 480 released in 2011, has 15 SMs with $Q = 32$ cores and shared memory of $M = 64$ KB, totally equipped with 480 cores.

A GPU achieves fast graphics processing by running hundreds of threads using SMs with shared memory. The use of a GPU instead of a CPU to perform general purpose computation is known as **GPGPU (General Purpose computing on GPUs, or GPU)** and its researches have attracted much attention in recent years, especially in high performance computing fields [2, 4, 9].

III. Pattern Matching System on GPUs

In this section, we present our pattern matching system on GPU based on simulation of NFAs using **RunHNFA** algorithm.

A. Architecture of proposed system

The architecture of the proposed system consists of two part: **host** side computing using CPU and **device** side computing using GPU. Our system works as below:

1. Construct a set of bitmasks for each pattern $P_i \in Pat$ on the host side.
2. Transport sets of bitmasks to the device side.
3. Run pattern matching algorithm using sets of bitmasks parallely on the device side.
4. Return the results of pattern matching to the host side.

B. RunHNFA algorithm

We show our pattern matching algorithm for extended pattern matching problem called **RunHNFA**. As preprocessing in this algorithm, we first transform input pattern P into **hierarchical NFA** (HNFA for short). The hierarchical NFA is a NFA specially designed for parallel calculation of NFA simulation, consisting of **upper module** UM and **lower module** LM. Figure 2 shows the hierarchical NFA of Example 1 for $w = 4$. Next, we constructs a set of bitmasks corresponding the hierarchical NFA. We use five types of bitmasks defined as follows:

- $Ch[c]$ is a w bit mask that indicates all bit-positions of letter move labeled with a letter c in input pattern P. That is, $Ch[c][i] = 1$ iff the state i has a letter move edge labeled with $c \in \Sigma$.

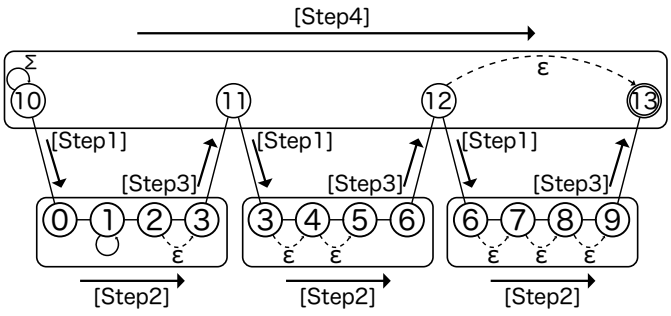

Fig. 2. Transition of hierarchical NFA .

- *Rep*[*c*] is the w bit mask that indicates all bit-positions of self-loops labeled with a letter c in input pattern P. That is, $Rep[c][i] = 1$ iff the state i has a self-loop labeled with $c \in \Sigma$.
- *Ebeg* is the w bit mask that sets 1 at the previous position of the lowest bit-position of every ε-block.
- *Eblk* is the w bit mask that sets 1's at all positions of in every ε-block.
- *Eend* is the w bit mask that sets 1 at the highest bit-position of every ε-block.

After the calculation of a set of bitmasks, we run runtime algorithm (Fig. 3) to find all the occurrence of input extended pattern on the text. In runtime algorithm RunHNFA, first we initialize the hierarchical NFA. Next, we simulate hierarchical NFA transition as below for each letter on input text:

Step1 Transport the active state from UM to LM (Line 6).

Step2 Simulate NFA transition on LM (Line 7).

Step3 Transport the active state from LM to UM (Line 8).

Step4 Simulate NFA transition on UM (Line 10).

Step5 Check the acceptance and report occurrence (Line 11).

IV. Time and Space Complexity

In this section, we analyze space and time complexity of our GPU pattern matching system. Let $L = \lceil \ell/w \rceil$ be the number of computer words necessary to store the state set of an NFA with ℓ states.

A. Space complexity

In RunHNFA algorithm, hierarchical NFA has one upper module and k lower modules, where $k = L = \lceil \ell/w \rceil$. For a upper module, it consists of state set mask S and three ε-transition masks of length w. For a lower module, it consists of state set mask S, three ε-transition masks, and $|\Sigma|$ arrays of bitmasks for character transition

Algorithm 1 RunHNFA($UM, (LM_j)_{j=0}^{k-1}$: module, T: text)

```
1:  procedure SEARCH
2:      n ← length(T);
3:      UM.I ← 0^{w-1}1;
4:      EpsClo(UM);
5:      for i = 0 to n do
6:          for j = 0 to k − 1 do
            UpperToLower(UM, LM_j);
7:          par j = 0 to k − 1 do
            RunExShiftAnd(LM_j, T[i]);
8:          for j = 0 to k − 1 do
            LowerToUpper(LM_j, UM);
9:          Syncthreads();
10:         EpsClo(UM)
11:         if UM.S & Bit(w) ≠ 0 then
12:             report an occurrence at i;
```

Fig. 3. The pattern matching algorithm using hierarchical NFA.

Algorithm 2 EpsClo(M: module)

```
1: M.S ← M.S | M.I
2: Z ← M.S | M.Eend
3: M.S ← M.S | (M.Eblk & ((∼ (Z − M.Ebeg)) ∧ Z))
```

Fig. 4. The procedure EpsClo for computing ε-closure.

Algorithm 3 RunExShiftAnd(LM_j: module, $T[i]$: letter)

```
1: LM.S ← LM.S | LM.I;
2: EpsClo(LM);
3: LM.S ← (((LM.S << 1) | LM.I) & LM.Ch[T[i]]) |
   (LM.S & LM.Rep[T[i]]);
4: EpsClo(LM);
```

Fig. 5. The procedure RunExShiftAnd for simulating NFA transition on the lower module.

Algorithm 4 UpperToLower(UM, LM: modules)

```
1: if UM.S & Bit(j) ≠ 0
2: then LM.I ← 1;
3: else LM.I ← 0;
```

Fig. 6. The procedure UpperToLower for transporting active state from upper module to lower module of hierarchical NFA.

Algorithm 5 LowerToUpper(LM, UM: module)

1: **if** $LM.S \,\&\, Bit(w) \neq 0$
2: **then** $UM.S \leftarrow UM.S \mid Bit(j)$;
3: **else** $UM.S \leftarrow UM.S \,\&\, (\sim Bit(j))$;

Fig. 7. The procedure LowerToUpper for transporting active state from upper module to lower module of hierarchical NFA.

TABLE I
SPACE COMPLEXITY OF RUNHNFA ALGORITHM.

module	S	ε-move	chr-move	#module	total
Upper	1	3	0	1	4
Lower	1	3	$2\|\Sigma\|$	$\lceil \ell/w \rceil$	$(4+2\|\Sigma\|)\lceil \ell/w \rceil$

and self-loop transition. Therefore, the memory usage of RunHNFA is,

$$\begin{aligned} S_{all} &= (2|\Sigma|+4)\lceil \ell/w \rceil + 4 \\ &= O(|\Sigma|\lceil \ell/w \rceil) \,(\text{words}) \end{aligned}$$

The details are given in TableI.

For **multi-stream pattern matching** which is single pattern matching on s streams, our algorithm can deal the task using one set of bitmasks (same as signal stream pattern matching) and s state set mask. In this situation, we consume

$$\begin{aligned} S_{all} &= s(\lceil \ell/w \rceil + 1) + (2|\Sigma|+3)\lceil \ell/w \rceil + 3 \\ &= O((s+|\Sigma|)\lceil \ell/w \rceil) \,(\text{words}) \end{aligned}$$

memory space.

B. Time complexity

If the NFA size is ℓ, our algorithm runs in $O(\ell)$ time for preprocessing phase. The RunHNFA shown in Fig. 3 simulates NFA transition for each letter $T[i]$ for $i = 1, \ldots n$. In the calculation of NFA transition (line 5 to 12), the procedure UpperToLower (Fig. 6), RunExShiftAnd (Fig. 5), and LowerToUpper (Fig. 7) are performed on every lower module $LM_j (0 \leq j \leq k-1)$, each procedure runs in $O(1)$ time, the total time on lower modules is $O(k) = O(\lceil \ell/w \rceil)$. For the upper module, the procedure EpsClo is performed on upper module runs in $O(1)$ time. Therefore, the total time of NFA simulation of each character in the input text is

$$O(\lceil \ell/w \rceil) + O(1) = O(\lceil \ell/w \rceil) \,(\text{time}). \quad (1)$$

Theorem 1 *Our proposed pattern matching algorithm RunHNFA solves extended pattern matching problem in $O(n\lceil \ell/w \rceil)$ time using $O(\ell)$ time for preprocessing and $O(|\Sigma|\lceil \ell/w \rceil)$ space, where n is the length of the input text, ℓ is the size of NFA of input pattern, w is the computer word length.*

TABLE II
RUNNING TIME OF GPU ALGORITHMS (SEC), WHERE t REPRESENTS THE NUMBER OF PATTERNS THAT CAN BE ALLOCATED TO EACH SM.

Algorithm	t	Number of Patterns p				
		20	40	60	80	100
ArrayNFA	4	568.2	569.1	570.1	571.6	628.5
RunHNFA	14	20.9	23.8	29.3	32.8	32.6
SpeedUp	–	x26	x24	x19	x19	x19

V. EXPERIMENTS

To evaluate the performance of our proposed system, we implemented following algorithms:

- **RunHNFA**: the GPU implementation of the proposed bit-parallel matching algorithm **RunHNFA** of Sec. III.B in CUDA language[1], which is a hierarchical version of **Extended SHIFT-AND** algorithm by Navarro and Raffinot [6].
- **RunHNFA-CPU**: the CPU implementation of **RunHNFA** algorithm in C++ language.
- **ArrayNFA**: the GPU implementation of the original non-hierarchical version of **Extended SHIFT-AND** algorithm [6], using integer array of length ℓ for representing a state set.

The experiments were performed on workstation equipped with a 2.8GHz Intel Core i7 processor with 12GB RAM and Nvidia GTX 480, operating on Debian GNU/Linux 6.0.5. We used g++ 4.6.3 and CUDA 5.0 as the compiler.

In what follows, let ℓ be a size of NFA of input pattern. In the experiments, we used 10 MB of amino-acid sequence data whose alphabet size is $|\Sigma| = 20$ from ExPASy : SIB Bioinformatics Resource Portal[2] as the input text and random generated extended string as the input pattern.

A. The comparison of algorithms on GPU

In this experiment, we measured running time of algorithms on GPU by varying the number of input pattern from 20 to 100. We fix the input extended pattern size as $\ell = 130$. Thus, the number of modules is $L = \lceil 130/32 \rceil = 4$. Table. II shows the result of this experiment. The column t on table. II represents the number of executable thread on each SM on GPU. In the table,

[1] https://developer.nvidia.com/category/zone/cuda-zone
[2] http://expasy.org/

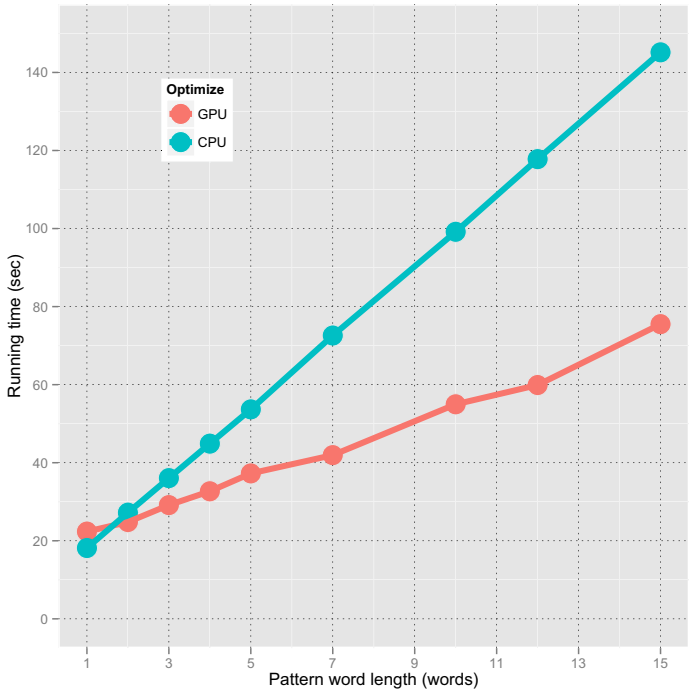

Fig. 8. The running time of the GPU version of pattern matcher (RunHNFA) and its CPU version (RunHNFA-CPU) by varying input pattern length.

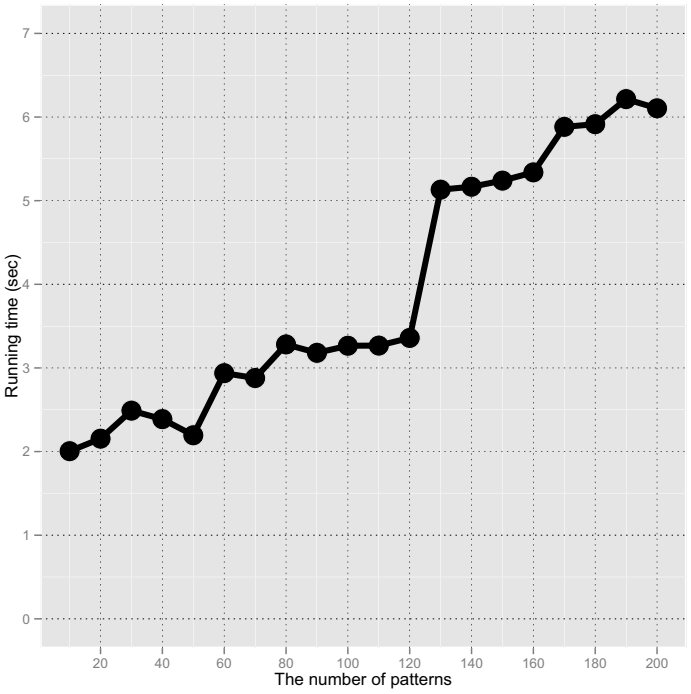

Fig. 9. The running time of GPU pattern matcher (RunHNFA) against the number of input patterns.

ArrayNFA ran 4 pattern matchers per SM with 64KB of shared memory and totally 4 logical threads. RunHNFA ran 14 pattern matchers per SM with the same amount of shared memory and totally $14 \times 4 = 56$ logical threads. The table shows RunHNFA is about from 19 to 26 times faster than ArrayNFA for the number of patterns $p = 20$ to 100.

B. *Comparison of input pattern length*

We compared the running time of RunHNFA and RunHNFA-CPU by varying the word length of input pattern NFA $L = \lceil \ell/w \rceil$. In Fig. 8, we observe the running time of GPU implementation is faster than CPU except the word length of the input pattern $L = 1$. We also observe GPU implementation is two times faster when $L = 12$.

C. *Running time against the number of patterns*

We measure the running time of our GPU system by varying the number of patterns from 10 to 200. We fix the pattern length $\ell = 100$. Fig. 9 shows the result of this experiment. In Fig. 9, the running time significantly changes when the number of input patterns is 140.

VI. Conclusion

In this paper we propose fast extended pattern matching system on GPU. In the proposed system, we give the GPU extension of bit-parallel pattern matching algorithm RunHNFA based on module decomposition of NFA. In the experiment, our algorithm achieved two times faster than CPU version of RunHNFA. The key of fast pattern matching is the compact representation of NFA using bit-parallel technique and module decomposition for GPU parallel execution.

As the future work, it is an interesting problem to extend our system to real world stream data such as GPS trajectories.

References

[1] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18(6):333–340, June 1975.

[2] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 18. ACM, 2009.

[3] C.-H. Lin, S.-Y. Tsai, C.-H. Liu, S.-C. Chang, and J.-M. Shyu. Accelerating string matching us-

ing multi-threaded algorithm on GPU. In *the 2010 IEEE Global Telecommunications Conference (GLOBECOM'10)*, pages 1 –5, dec. 2010.

[4] S. Manavski and G. Valle. Cuda compatible gpu cards as efficient hardware accelerators for smith-waterman sequence alignment. *BMC bioinformatics*, 9(Suppl 2):S10, 2008.

[5] A. Margara and G. Cugola. High performance content-based matching using gpus. In *Proceedings of the 5th ACM international conference on Distributed event-based system*, pages 183–194. ACM, 2011.

[6] G. Navarro and M. Raffinot. Fast and simple character classes and bounded gaps pattern matching, with application to protein searching. In *Proceedings of the fifth annual international conference on Computational biology*, RECOMB '01, pages 231–240, New York, NY, USA, 2001. ACM.

[7] G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings: Practical on-line search algorithms for texts and biological sequences*. Cambridge University Press, 2002. ISBN 0-521-81307-7. 280 pages.

[8] S. Vigna. Broadword implementation of parenthesis queries. *CoRR*, abs/1301.5468, 2013.

[9] P. Vouzis and N. Sahinidis. GPU-BLAST: using graphics processors to accelerate protein sequence alignment. *Bioinformatics*, 27(2):182–188, 2011.

[10] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'06)*, pages 407–418. ACM, 2006.