4-러시안 알고리즘의 CUDA 구현

김영호*, 정주희*, 강대웅*, 심정섭*, 김민호**, 박수준**, 임명은**, 정호열**
*인하대학교 컴퓨터정보공학부
**한국전자통신연구원
yhkim8505@gmail.com jngjuhe@naver.com kdw8219@naver.com jssim@inha.ac.kr
kimmh@etri.re.kr psj@etri.re.kr melim@etri.re.kr hoyoul.jung@etri.re.kr

CUDA Implementation for the Four-Russian Algorithm

Young Ho Kim*, Ju-Hui Jeong*, Dae Woong Kang*, Jeong Seop Sim*,
Minho Kim**, Soo-jun Park**, Myungeun Lim**, Ho-Youl Jung**
*Dept. of Computer Science and Information Technology, Inha University
**Electronics and Telecommunications Research Institute

요 약

상수 크기의 알파벳 Σ에 대해 길이가 각각 m, n인 두 문자열 X와 Y의 편집거리는 X를 Y로 변환하기 위해 필요한 최소 편집연산의 수로 정의된다. 두 문자열의 편집거리는 잘 알려진 동적프로그래밍을 이용하여 $O(mn)$ 시간과 공간에 계산할 수 있으며, 4-러시안 알고리즘을 이용해도 계산할 수 있다. 4-러시안 알고리즘은 블록 크기를 상수 t라 할 때, 전처리 단계에서 $O((3|\Sigma|)^{2t}t^2)$ 시간과 $O((3|\Sigma|)^{2t}t)$ 공간이 필요하며, 계산 단계에서 $O(mn/t)$ 시간과 $O(mn)$ 공간을 이용하여 편집거리를 계산하는 알고리즘이다. 본 논문에서는 4-러시안 알고리즘의 계산 단계를 CUDA를 이용하여 구현하고 실험을 통해 CPU 기반의 순차적인 수행시간과 GPU 기반의 병렬적인 수행시간의 비교결과를 제시한다. 본 논문의 병렬알고리즘은 m/t개의 쓰레드를 사용하여 $O(m+n)$ 시간에 편집거리를 계산한다. GPU 기반의 알고리즘이 CPU 기반의 알고리즘 보다 $t=1$일 때 약 10배 빠르고, $t=2$일 때 약 3배 빠른 결과를 보였다.

1. 서론

문자열들의 불일치를 어느 정도 허용하는 근사문자열매칭 문제는 생물정보학, 검색엔진, 컴퓨터보안 등 많은 분야에서 활발히 연구되고 있다[1-3]. 최근에는 차세대염기서열분석(next-generation sequencing)에서 레퍼런스 매핑(reference mapping)의 비용과 시간을 줄이기 위해 빠른 근사문자열매칭 알고리즘들이 이용되고 있다[4,5]. 또한, 편집거리(edit distance)도 근사문자열매칭 알고리즘을 이용하여 계산할 수 있다.

편집거리는 두 문자열의 유사도(similarity)를 판별하는 척도로서 염기서열 분석에도 활용될 수 있다. 상수 크기의 알파벳 Σ에 대해 길이가 각각 m, n인 두 문자열 X와 Y의 편집거리는 X를 Y로 변환하기 위해 필요한 최소 편집연산(edit operation)의 수로 정의된다[6]. 이때 편집연산은 삽입(insertion) 연산, 삭제(deletion) 연산, 교체(change) 연산으로 구성된다. 두 문자열의 편집거리는 잘 알려진 동적프로그래밍(dynamic programming)을 이용하여 $O(mn)$ 시간과 공간에 계산할 수 있으며[7], 4-러시안 알고리즘을 이용해도 계산할 수 있다[8,9]. 4-러시안 알고리즘은 블록 크기를 상수 t라 할 때, 전처리 단계에서 $O((3|\Sigma|)^{2t}t^2)$ 시간과 $O((3|\Sigma|)^{2t}t)$ 공간이 필요하며, 계산 단계에서 $O(mn/t)$ 시간과 $O(mn)$ 공간을 이용하여 편집거리를 계산하는 알고리즘이다.

최근 GPU(graphic processing unit)의 성능이 향상되면서 문자열 관련 문제에서도 GPU를 활용하여 해결하려는 연구가 진행되고 있다. [10]에서는 CPU에서 접미사트리(suffix tree)를 생성하고 GPU에서 매칭을 구현하여 레퍼런스 매핑 속도를 향상시켰다. [11]에서는 CPU에서 접미사배열(suffix array)을 생성하고 GPU에서 매칭을 구현하여 레퍼런스 매핑 속도를 향상시켰다. [12]에서는 Aho-Corasick 오토마타를 이용한 그래프 모델을 CUDA로 생성하여 최장공통비상위문자열(longest common non-superstring) 문제를 해결하였다. [13]에서는 Smith-Waterman 알고리즘을 GPU 기반으로 구현하여 편집거리를 계산하였다.

본 논문에서는 4-러시안 알고리즘을 병렬화하고 GPU를 활용하여 구현한다. 그리고 실험을 통해 CPU 기반의 순차알고리즘과 CUDA로 구현한 GPU 기반의 병렬알고리즘의 성능을 비교한다.

본 논문의 구성은 다음과 같다. 2장에서는 본 논문의 알고리즘을 위한 몇 가지 용어 정의와 관련 연구의 결과들을 제시한다. 3장에서는 4-러시안 알고리즘으로 편집거리를 계산하는 병렬알고리즘을 설명한다. 4장에서는 각각 CPU와 GPU로 구현한 4-러시안 알고리즘의 성능비교를 하고, 5장에서 결론과 향후 연구 방향을 제시한다.

• 이 논문은 지식경제부 및 한국산업기술평가관리원의 IT산업원천 기술개발산업의 일환으로 수행하였음.[10038768, 유전체 분석용 슈퍼컴퓨팅 시스템 개발]
• 이 논문은 2011년도 정부(교육과학기술부)의 재원으로 한국연구재단의 지원을 받아 수행된 기초연구사업임(No. 2011-0027343)

2. 관련 연구

상수 크기의 알파벳 Σ에 대한 길이가 n인 문자열 S $(1 \le i \le j \le n)$에 대해 $S[i]$는 i번째 문자를 나타낸다. $S[1..i]$는 S의 길이가 i인 접두사(prefix)를 나타내고, $S[j..n]$는 S의 길이가 $n-j+1$인 접미사(suffix)를 나타낸다. 그리고 $S[i..j]$는 S의 i번째부터 시작하고 길이가 $j-i+1$인 부분문자열(substring)을 나타낸다. 그리고 Σ^t는 Σ에 대해 길이 t인 문자열로 정의한다.

2.1 D-테이블

	Δ	a	b	c	a	c	d	c	a	c
Δ	0	1	2	3	4	5	6	7	8	9
a	1	0	1	2	3	4	5	6	7	8
b	2	1	0	1	2	3	4	5	6	7
a	3	2	1	1	1	2	3	4	5	6
b	4	3	2	2	2	2	3	4	5	6
c	5	4	3	2	3	2	3	3	4	5
a	6	5	4	3	2	3	3	4	3	4

(그림 1) X= *ababca* , Y= *abcacdcac* 의 D-테이블

길이가 각각 m, n인 두 문자열 X, Y의 편집거리는 잘 알려진 동적프로그래밍을 이용하여 계산할 수 있다. 이때 계산된 $(m+1)\times(n+1)$ 크기의 테이블을 D-테이블이라고 하자(그림 1 참조). D-테이블의 $D[i,j]$는 $X[1..i]$와 $Y[1..j]$의 편집거리를 저장한다. D-테이블의 첫 번째 행과 열은 다음과 같이 초기화된다.

$$\begin{pmatrix} D[0,0]=0\ , \\ D[i,0]=i \quad (1 \le i \le m), \\ D[0,j]=j \quad (1 \le j \le n) \end{pmatrix} \qquad (1)$$

이후, 각 $D[i,j]$는 다음의 점화식으로 구한다.

$$D[i,j]=\min\begin{pmatrix} D[i,j-1] & + & 1, \\ D[i-1,j] & + & 1, \\ D[i-1,j-1] & + & \delta(X[i],\,Y[j]) \end{pmatrix} \qquad (2)$$

만약 $X[i]=Y[j]$ 이면, $\delta(X[i],\,Y[j])=0$ 이고, $X[i] \ne Y[j]$ 이면, $\delta(X[i],\,Y[j])=1$ 이 된다. 각 $D[i,j]$는 식(2)를 이용하여 $O(1)$시간에 구할 수 있고, 테이블의 크기가 $(m+1)\times(n+1)$이므로 $O(mn)$시간에 D-테이블을 구할 수 있다. 이때 $D[m,n]$이 두 문자열 X, Y의 편집거리이다. 그림 1은 X= *ababca* , Y= *abcacdcac* 일 때, 식 (1)과 (2)를 이용하여 생성된 D-테이블이다.

2.2 4-러시안 알고리즘

4-러시안 알고리즘의 기본 아이디어는 D-테이블을 블록 단위로 나눠서 계산하는 것이다. 크기가 $(t+1)\times(t+1)$인 블록을 t-블록이라 하자. 그리고 (i,j,t) 블록을 왼쪽 상단이 (i,j)인 t-블록이라 하겠다. 예를 들어 그림 2를 보면, (0,0,3), (0,3,3), (0,6,3), (3,0,3), (3,3,3), (3,6,3) 블록이 있다.

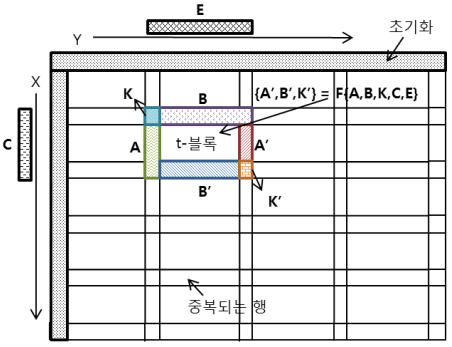
	Δ	a	b	c	a	c	d	c	a	c
Δ	0	1	2	3	4	5	6	7	8	9
a	1			2			5			8
b	2			1			4			7
a	3	2	1	1	1	2	3	4	5	6
b	4			2			3			6
c	5			2			3			5
a	6	5	4	3	2	3	3	4	3	4

(그림 2) 4-러시안 알고리즘으로 생성한 D-테이블

D-테이블은 임의의 행 또는 열에서 이전의 값과 차이가 최대 1이기 때문에 t-블록의 행과 열의 값을 {-1,0,1}로 변환한다[9]. 예를 들어 그림 2의 블록(3,3,3)의 4행 [3, 2, 3, 3]은 벡터 [0, -1, 1, 0]으로 변환될 수 있다.

각 (i,j,t) 블록은 두 벡터 A, B, 변수 K, 그리고 두 문자열 C, E인 $\{A,B,K,C,E\}$에 따라 편집거리가 결정된다(그림 3 참조). A는 각 블록의 첫 번째 열을 나타내는 벡터 $[a_0..a_p]$ $(a_p \in \{-1,0,1\},\ 0 \le p \le t)$이고, B는 첫 번째 행을 나타내는 벡터 $[b_0..b_q]$ $(b_q \in \{-1,0,1\},\ 0 \le q \le t)$이다. K는 a_0와 b_0에 대응되는 $D[i,j]$이다. C와 E는 각각 해당 블록에 대응하는 X와 Y의 부분문자열인 $X[i..i+t]$와 $Y[j..j+t]$이다. 만약 K 값이 상수 k라면, A와 B를 {-1,0,1}의 조합으로 변환할 수 있다. 예를 들어 그림 2의 (3,3,3) 블록은 $\{A,B,K,C,E\}$를 다음과 같이 정의할 수 있다. K = D[3,3] = 1, A = [0,1,0,1], B = [0,0,1,1], C = "*abca*", 그리고 E = "*cacd*". 그러고 나서 $\{A,B,K,C,E\}$을 입력받아 $\{A',B',K'\}$을 K' = $D[i+t,j+t]$, A' = $[D[i,j+t],D[i+1,j+t],..,D[i+t,j+t]]$, B' = $[D[i+t,j],D[i+t,j+1],..,D[i+t,j+t]]$으로 구한다. 여기서 A와 B를 {-1,0,1}을 원소로 갖는 벡터로 변환한 이유는 (i,j,t)블록에서 출력한 K'이 $(i+t,j+t,t)$블록의 입력 K로 사용되기 때문에 $(i+t,j+t,t)$블록의 A와 B에 K값을 더하면 원래 계산하려는 값을 복원할 수 있기 때문이다.

4-러시안 알고리즘은 크게 전처리 단계와 실제 계산 단계로 나눌 수 있다.



(그림 3) 전처리된 룩업테이블을 사용한 4-러시안 계산[9]

(1) 전처리 단계

전처리 단계에서는 계산 단계에서 사용할 룩업테이블(lookup table)을 생성한다. 길이 $t+1$인 문자열($\sum^{t+1}$)과 $\{-1,0,1\}^{t+1}$를 원소로 갖는 벡터의 모든 조합에 대해 t-블록 편집거리를 계산한다. 그러고 나서 각 t-블록의 $\{A,B,C,E\}$에 대한 값을 유일한 인덱스로 하는 룩업테이블을 생성한다. 이 룩업테이블은 각 조합에 대한 t-블록의 편집거리를 계산한 마지막 행과 열에 대한 벡터 값만 저장한다. 이것은 계산 단계에서 $O(1)$ 시간에 해당 조합인 t-블록의 편집거리 정보를 찾을 수 있다.

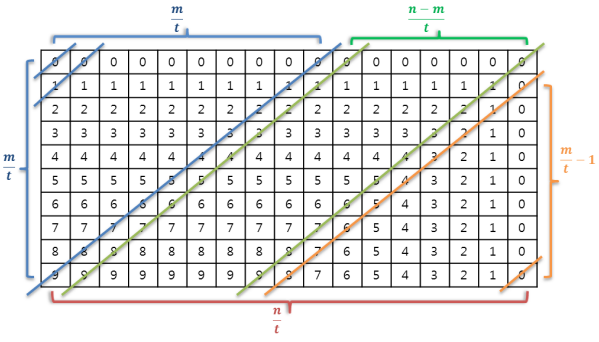
$\{-1,0,1\}^{t+1}$에 대한 모든 조합을 구하기 위해서는 3^{t+1}개의 조합이 필요하다. 그리고 $\sum^{t+1}$에 대한 모든 조합을 구하기 위해서는 $|\sum|^{t+1}$개의 조합이 필요하다. 그러므로 모든 조합의 t-블록을 전처리하는데 $O((3|\sum|)^{2t}t^2)$의 시간이 필요하다. 룩업테이블은 마지막 행과 열만 저장하기 때문에 $O((3|\sum|)^{2t}t)$의 공간이 필요하다.

(2) 계산 단계

실제 계산을 하기 전에 D-테이블의 첫 번째 행과 열은 초기화 되어야 한다. 이는 식(1)의 D-테이블을 초기화하는 과정과 같다.

계산 단계에서는 해당 블록의 $\{A,B,K,C,E\}$에 대해 전처리 단계에서 구한 룩업테이블을 이용하여 $\{A',B',K'\}$를 구한다. 룩업테이블에서 해당 조합의 정보를 찾는데 $O(1)$ 시간이 필요하고, 변환된 $\{A',B',K'\}$ 값을 복원하기 위해서 $O(t)$ 시간이 필요하다. $\{A',B',K'\}$는 인접한 블록을 구하기 위해 다시 입력으로 사용된다(그림 3 참조). 그러므로 D-테이블 전체의 편집거리를 구하기 위해서는 $O(mn/t)$ 시간이 필요하다. 만약 $t=\Theta(\log m)$ 이면, 4-러시안 알고리즘은 $O(mn/\log m)$ 시간에 수행된다.

3. 4-러시안 알고리즘의 CUDA 구현



(그림 4) 4-러시안 병렬알고리즘을 이용한 D-테이블 계산

본 논문에 제시된 알고리즘은 4-러시안 알고리즘의 계산 단계를 CUDA를 이용하여 병렬적으로 계산한다. 병렬알고리즘을 설명하기에 앞서 그림 4의 D-테이블의 각 셀을 t-블록으로 가정하자. 4-러시안 병렬알고리즘은 한 회 당 대각선에 있는 t-블록들을 병렬적으로 계산한다. 한 회 당 최대 m/t개의 쓰레드를 사용하여 대각선에 있는 t-블록마다 하나의 쓰레드를 할당하면, 각 t-블록에 쓰레드 번호를 매길 수 있다. 그림 4에서 대각선에 있는 각 셀은 한 회에 병렬적으로 계산되는 t-블록들을 보여주고, 각 t-블록에 있는 번호는 쓰레드 번호를 의미한다.

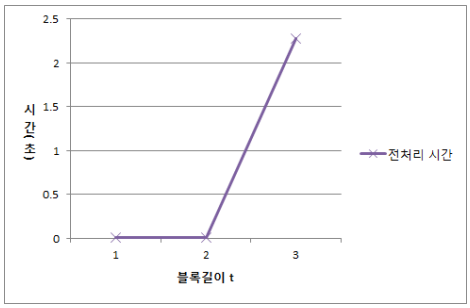
병렬알고리즘은 총 3 부분으로 설명할 수 있다. 첫 번째 부분(그림 4의 파란색 대각선이 있는 부분)은 각 회당 쓰레드 개수가 1씩 증가하고, m/t회 수행한다. 두 번째 부분(녹색 대각선이 있는 부분)은 m/t개의 쓰레드를 가지고 $(n-m)/t$회 수행한다. 마지막 부분(주황색 대각선이 있는 부분)은 쓰레드 개수가 1씩 감소하고, $(m/t)-1$회 수행한다. 그러면 총 $O((m+n)/t)$회 수행하게 된다. 각 t-블록 당 룩업테이블에서 A'과 B'을 읽어오는데 $O(t)$시간이 필요하므로 4-러시안 병렬알고리즘의 시간복잡도는 $O(m+n)$이 된다.

4. 실험 결과 및 분석

실험 환경은 다음과 같다. CPU는 Intel Core i7 970, RAM은 6GB, GPU는 NVIDIA tesla c2070, 그리고 OS는 Linux Fedora(64bit)를 사용하였다.

$\sum$ = {A,C,G,T}로 고정하고, 시간 함수는 [4]에서 제공하는 SOAP2 프로그램의 함수를 이용하였다. 그리고 실험은 전처리 단계와 계산 단계로 구분하여 실험하였다. 전처리 단계는 t에 대해 룩업테이블을 생성하는 수행시간을 실험하였고, 계산 단계는 두 문자열의 길이를 1,000부터 10,000까지 1,000씩 증가시키면서 4-러시안 알고리즘으로 D-테이블을 생성하는 시간을 실험하였다. 그리고 CPU 코드와 GPU 코드의 수행시간을 비교하였다. GPU로 구현한 4-러시안의 수행시간은 CUDA의 특성상 호스트 메모리(main memory)와 디바이스 메모리(GPU memory) 사이의 데이터를 복사하는 cudaMemcpy() 함수의 수행시간을 포함하였다. 앞으로 GPU 기반의 4-러시안 프로그램을 G4R이라 하고, CPU 기반의 4-러시안 프로그램을 C4R이라 하겠다.

4.1 전처리 단계의 수행 결과



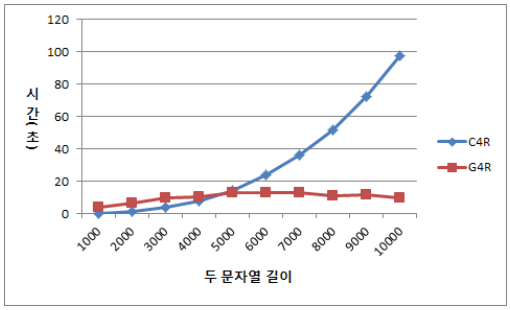
(그림 5) 블록길이에 따른 수행시간

그림 5는 블록의 길이 t가 1에서 3일 때, 전처리 단계의 수행시간이다. 시간복잡도인 $O((3|\sum|)^{2t}t^2)$에서 알 수 있듯이 t의 크기에 따라 급격하게 수행시간이 증가한다. t = 4일 때

는 공간복잡도인 $O((3|\Sigma|)^{2t}t)$에서 알 수 있듯이 메모리 부족으로 메모리 스왑이 20GB 가까이 일어나고, 오랜 수행시간으로 실험이 불가능했다.

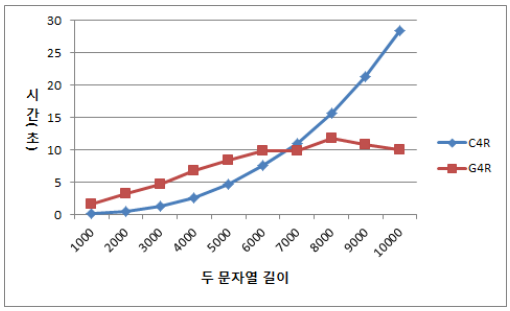
4.2 계산 단계의 수행 결과

계산 단계에서 C4R와 G4R의 성능비교는 GPU 메모리 문제로 t를 2까지 실험하였다.



(그림 6) C4R과 G4R의 성능 비교
($t=1$, $1{,}000 \le m,n \le 10{,}000$)

그림 6은 $t=1$ 일 때, 두 문자열의 길이를 동일하게 증가시키면서 실험한 결과이다. C4R는 수행시간이 선형적으로 증가하는 것을 보이는 반면 G4R는 거의 일정한 수행시간을 보였다. 두 문자열의 길이가 각각 10,000 일 때, G4R의 수행시간은 약 9.78초이고, C4R는 약 97.42초로 약 10배 빠른 실험결과를 보였다.



(그림 7) C4R과 G4R의 성능 비교
($t=2$, $1{,}000 \le m,n \le 10{,}000$)

그림 7은 $t=2$ 이고, 나머지 입력은 그림 6과 동일하게 실험하였다. 그림 6과 비교하면 G4R의 수행시간은 거의 동일하였지만, C4R의 성능이 약 3배 증가하였다. 그래도 여전히 입력 크기에 따라 C4R의 수행시간은 선형적으로 증가하였고, G4R의 수행시간은 거의 일정하였다. 두 문자열의 길이가 각각 10,000일 때, G4R은 약 9.98초의 수행시간을 가졌고, C4R은 약 28.39초의 수행시간을 가졌다. 즉, G4R이 C4R보다 약 3배 빠른 결과를 보였다.

5. 결론

본 논문에서는 4-러시안 알고리즘을 병렬화하여 CUDA로 구현하였다. 4-러시안 알고리즘 특성상, 블록 길이에 따라 기하급수적인 시간 및 공간이 필요하기 때문에 다양한 블록 길이에 대해 실험하지 못했다. 그러나 작은 t에 대해 G4R은 좋은 수행속도를 가진다. 이론상으로도 m/t개의 쓰레드를 사용했을 때, 계산 단계의 시간복잡도가 $O(mn/t)$에서 $O(m+n)$으로 향상되는 것을 보였다. 앞으로 GPU의 메모리가 커지고 코어 개수가 더 늘어나면, 전처리 단계도 GPU를 이용해 병렬적으로 처리할 수 있을 것이다.

참고문헌

[1] A. V. Aho and M. J. Corasick, “Efficient String Matching: An Aid to Bibliographic Search”, Communications of the ACM, vol.18, no.6, 1975.

[2] S. Forrest, A. S. Perelson, L. Allen, and R. Cherukuri, “Self-nonself discrimination in a computer”,in Proc.IEEE Symp. Res. Security Privacy, pp.202-212, 1994.

[3] T. F. Smith and M. S. Waterman, “Identification of Common Molecular Subsequences”, J. Mol. Biol. 147, pp.195-197, 1981.

[4] R. Li, C. Yu, Y. Li, T. W. Lam, S. M. Yui, K. Kristiansen and J. Wang, “SOAP2: an improved ultrafast tool for short read alignment”bioinformatics, vol.25, no.15, pp.1966-1967, 2009.

[5] S. Bao, R. Jiang, W. K. Kwan, B. B. Wang, X. Ma and Y. Q. Song, “Evaluation of next-generation sequencing software in mapping and assembly”, Journal of Human Genetics, vol.56, pp.406-414, 2011.

[6] D. Gusfield, Algorithms on Strings, Trees, and Sequences, Cambridge university press, 1997.

[7] V. I. Levenstein, Binary codes capable of correcting deletions, insertions, and reversals, Sov. Phys. Dokl., vol.10, pp.707-710, 1966.

[8] W. J. Masek and M. S. Paterson, “A Faster Algorithm Computing String Edit Distance”, Journal of computer and system science, vol.20, no.1, pp.18-31, 1980.

[9] V. Kundeti and S. Rajasekaran, “Extending the Four Russian Algorithm to Compute the Edit Script in Linear Space” ICCS 2008, pp.893-902, 2008.

[10] C. Trapnell and M. C. Schatz, “Optimizing data intensive GPGPU computations for DNA sequence alignment”, Parallel Computing, vol.35, no.8-9, pp.429 - 440, 2009.

[11] A. Gharaibeh and M. Ripeanu. “Size Matters: Space/Time Tradeoffs to Improve GPGPU Applications Performance”, In IEEE/ACM Supercomputing (SC 2010), 2010.

[12] 윤현철, 심정섭, “최장공통비상위문자열 그래프 모델의 CUDA 기반 구현”, 한국정보과학회 학술발표논문집, vol.38, no.2, 2011.

[13] L. Ligowski, W. Rudnicki, “An efficient implementation of Smith Waterman algorithm on GPU using CUDA, for massively parallel scanning of sequence databases”, in Parallel & Distributed Processing (IPDPS). IEEE Int. Symp., pp.1-8, 2009.