KIPS Trans. Comp. and Comm. Sys.
Vol.7, No.9 pp.219~226 pISSN: 2287-5891



https://doi.org/10.3745/KTCCS.2018.7.9.219

# Efficient Collaboration Method Between CPU and GPU for Generating All Possible Cases in Combination

Ki-Bong Son† · Min-Young Son†† · Young-Hak Kim†††

## ABSTRACT

One of the systematic ways to generate the number of all cases is a combination to construct a combination tree, and its time complexity is $O(2^n)$. A combination tree is used for various purposes such as the graph homogeneity problem, the initial model for calculating frequent item sets, and so on. However, algorithms that must search the number of all cases of a combination are difficult to use realistically due to high time complexity. Nevertheless, as the amount of data becomes large and various studies are being carried out to utilize the data, the number of cases of searching all cases is increasing. Recently, as the GPU environment becomes popular and can be easily accessed, various attempts have been made to reduce time by parallelizing algorithms having high time complexity in a serial environment. Because the method of generating the number of all cases in combination is sequential and the size of sub-task is biased, it is not suitable for parallel implementation. The efficiency of parallel algorithms can be maximized when all threads have tasks with similar size. In this paper, we propose a method to efficiently collaborate between CPU and GPU to parallelize the problem of finding the number of all cases. In order to evaluate the performance of the proposed algorithm, we analyze the time complexity in the theoretical aspect, and compare the experimental time of the proposed algorithm with other algorithms in CPU and GPU environment. Experimental results show that the proposed CPU and GPU collaboration algorithm maintains a balance between the execution time of the CPU and GPU compared to the previous algorithms, and the execution time is improved remarkable as the number of elements increases.

Keywords : Combination, All Possible Cases, GPU, CPU, Collaboration

# 조합에서 모든 경우의 수를 만들기 위한 CPU와 GPU의 효율적 협업 방법

손 기 봉† · 손 민 영†† · 김 영 학†††

## 요 약

조합에서 모든 경우의 수를 생성하는 체계적인 방법 중 하나는 조합 트리를 구성 하는 것이며 조합 트리를 구성하는 시간 복잡도는 $O(2^n)$이다. 조합 트리는 그래프 동형 문제나 빈발 항목집합을 계산하는 초기 모델 등 다양한 목적으로 활용된다. 그러나 조합의 모든 경우의 수를 탐색해야 하는 알고리즘은 높은 시간 복잡도로 인해 현실적으로 활용되기 어렵다. 그럼에도 불구하고 데이터의 양이 방대해지고 이를 활용하기 위한 다양한 연구가 진행되면서 모든 경우의 수를 탐색해야만 하는 경우가 늘고 있다. 최근 GPU환경이 보급되고 쉽게 접할 수 있게 되면서 직렬 환경에서 높은 시간 복잡도를 가지는 알고리즘들을 병렬화 하여 시간을 줄이려는 다양한 시도가 이루어지고 있다. 조합에서 모든 경우의 수를 생성하는 방법은 순차적으로 진행되고 하부 작업의 크기가 편향되기 때문에 병렬 구현에 적합하지 않다. 병렬 알고리즘의 성능은 모든 스레드가 비슷한 크기의 작업을 가질 때 극대화될 수 있다. 본 논문에서는 모든 경우의 수를 구하는 문제를 병렬화 하기 위하여 CPU와 GPU가 효율적으로 협업하기 위한 방법을 제안한다. 제안한 알고리즘의 성능을 검증하기 위하여 이론적인 측면에서 시간 복잡도를 분석하고, CPU와 GPU환경에서 다른 알고리즘과 본 연구에서 제안한 알고리즘의 실험 시간을 비교한다. 실험 결과 본 연구에서 제안한 CPU와 GPU의 협업 알고리즘은 이전 알고리즘에 비하여 CPU의 수행시간과 GPU의 수행시간의 균형을 유지하였고 아이템의 개수가 커질수록 괄목할 만한 시간 개선을 보였다.

키워드 : 조합, 모든 경우의 수, GPU, CPU, 협업

※ 이 연구는 금오공과대학교 학술연구비로 지원되었음(2016-104-082).

† 비 회 원 : 금오공과대학교 컴퓨터공학과 박사과정
†† 비 회 원 : 금오공과대학교 컴퓨터공학과 박사
††† 종신회원 : 금오공과대학교 컴퓨터공학과 교수

Manuscript Received : February 7, 2018
First Revision : May 8, 2018
Second Revision : June 15, 2018
Accepted : June 26, 2018

* Corresponding Author : Young-Hak Kim(kimyh@kumoh.ac.kr)

## 1. 서 론

그래프 동형 알고리즘이나 Apriori 알고리즘의 초기 모델 등은 조합의 모든 경우의 수를 구해야 하는 알고리즘의 한 분야이다. 최근 데이터의 생성이 급격히 증가하고 이를 분석하기 위한 다양한 방법이 제시되면서 데이터 조합의 모든 경우의 수를 고려해야 하는 경우가 늘고 있다[1-4].

조합의 모든 경우의 수를 만들기 위한 체계적인 방법 중 하나는 트리를 이용해 모든 경우의 수를 생성하는 것이다. 그러나 이는 $O(2^n)$의 시간 복잡도를 가지고 있기 때문에 현실적으로 활용하는데 한계가 있다. 컴퓨터 기술의 발달에 따라 하드웨어와 소프트웨어가 괄목할 만하게 성장했음에도 불구하고, 데이터의 대형화 속도가 하드웨어나 소프트웨어의 발달 속도를 넘어서면서 모든 경우의 수를 고려해야 하는 알고리즘들은 비효율성으로 인해 실현되지 못하고 다른 접근 방식을 연구하게 되었다.

다른 방식의 알고리즘에 대한 연구에도 불구하고 모든 경우의 수를 고려하는 것은 문제 해결 알고리즘을 간단히 할 뿐만 아니라 모든 수의 경우를 고려하기 때문에 정확성을 높이는 방법이 된다.

한편 최근 그래픽 카드의 보급이 확산됨에 따라 GPU 환경에 대한 접근이 쉬워지고 있다. 기존의 병렬 알고리즘에 대한 연구와 구현은 슈퍼컴퓨터 등의 고가 장비를 요구하기 때문에 활발하지 못한 실정이었다. 그러나 최근 병렬 구조의 GPU 환경은 저렴하고 고성능의 연산 수행을 지원하여 병렬 알고리즘에 대한 연구가 활기를 띄고 있다.

GPU환경은 여러 스레드가 동시에 통계 등 수학적 유사한 연산을 계산하는데 적합한 환경으로 구성되어 있다. 이러한 GPU의 특성을 최대한 활용하기 위해서는 비슷한 크기의 연산들을 각각의 스레드에 분산 배정 하는 것이 중요하다. 기존 직렬 알고리즘 중에 GPU 환경에서 효율적으로 병렬화하기 위해서는 분할 정복이 적합하며 서브 작업들이 서로 독립적이고 비슷한 크기를 가져야 한다. 그러나 일부 직렬 알고리즘은 순차적인 처리를 필수적으로 요구하거나 서브 작업이 독립적이지 못하기 때문에 병렬화가 어렵다.

본 연구에서 대상으로 하는 조합의 모든 경우의 수를 생성하는 알고리즘이나 조합과 밀접하게 관련된 순열을 생성하는 알고리즘은 대부분 순차적인 방법으로 병렬화가 어려운 알고리즘 중 하나이다[5]. 이러한 어려움을 감안하여 경우에 따라 조합이나 순열을 생성하는 방법을 우회하거나 보완하는 병렬 알고리즘들이 제안되었다[6]. 그러나 조합의 모든 경우의 수 전체를 생성하는 병렬 알고리즘에 대한 연구는 부족한 실정이다.

조합의 모든 경우의 수를 트리를 이용하여 만드는 방법은 계층에 따라 순차적으로 진행되나 동일한 계층 내의 서브 작업은 독립적이기 때문에 병렬 처리가 가능하다. 이와 관련한 연구로 연산 수행 시간을 줄이기 위해 Apriori 알고리즘에 대한 Yanbin Ye 등의 병렬화 연구를 들 수 있다[7]. 하지만 Yanbin Ye 등의 방법은 병렬 수행을 위해 각 프로세서에 균등하게 항목들을 분배하지 못하고 한쪽으로 많은 항목이 분배되는 단점을 가지고 있다. 따라서 병렬 수행에 대한 효율성이 떨어지고 수행 시간이 길어지게 된다.

본 연구에서는 이러한 단점을 개선하고 GPU환경에서 조합의 모든 경우의 수를 효율적으로 병렬화 수행하기 위해 CPU에서 서브 작업의 크기를 균등하게 하는 과정과 CPU의 결과물을 GPU에서 병렬 처리하는 협업 알고리즘을 제안한다. 제안한 병렬화 방법의 성능을 검증하기 위하여 시간 복잡도를 분석하고, 제안된 방법을 구현하여 실험 결과를 비교 분석한다. 제한된 방법의 실험을 위해 각 스레드에 비슷한 크기의 연산을 할당하여 스레드에서 발생할 수 있는 연산 시간의 균형을 유지하고 대기 시간 등의 오버헤드를 최소하였다. 이러한 결과로 기존의 방법에 비해 전체 연산 시간이 크게 감소하였으며, 더 많은 개수의 원소를 이용한 조합으로 확장이 가능하였다.

2장에서는 관련 연구로서 GPU 환경 및 CUDA(Compute Unified Device Architecture)와 조합의 모든 경우의 수를 트리를 통해 구하는 방법에 대해 설명하고, 3장에서는 조합의 모든 경우의 수를 구하기 위하여 CPU와 GPU의 협업 기법을 제안하며 시간 복잡도를 분석한다. 4장에서는 제안한 알고리즘을 구현하여 다양한 변수에 대해 연산 속도를 비교 분석하고, 5장에서 결론과 향후 연구에 대해 기술한다.

## 2. 관련 연구

### 2.1 GPU와 CUDA

GPGPU(General-Purpose computing on Graphics Processing Units)는 GPU(Graphic Processing Units)의 빠른 계산력과 높은 메모리 대역, 병렬화, 멀티 스레드, 다중 코어를 이용하여 일반적인 목적을 달성하기 위해 활용할 수 있는 환경을 의미한다.

GPGPU는 CPU가 하던 그래픽처리 및 대용량의 데이터 계산을 대신 처리함으로서 빠른 처리 속도를 보이고 있다. GPGPU의 구조는 SIMD(Single Instruction Multiple Data)와 비슷하지만 GPGPU는 스레드를 기반으로 처리하기 때문에 SIMT(Single Instruction Multiple Threads)라고 부른다.

CUDA는 이러한 GPGPU에 작업 처리를 지시하기 위한 C 언어 기반 프로그래밍 언어이다. CUDA는 커널(Kernel) 함수가 준비되면 커널 호출 전 그리드를 구성하고, 각 그리드에서는 블록이 구성되며, 이 블록은 많은 스레드를 생성하면서 서로 데이터를 공유하고 병렬처리를 수행하게 된다[8].

### 2.2 조합의 모든 경우의 수

조합에서 항목이 4개일 때 모든 경우의 수를 생성하면 Fig. 1과 같이 생성될 수 있는 경우의 수는 16개가 된다. 즉, 전체 요소가 n개인 집합에서 모든 부분 k 조합을 만드는 경우의 수는 $O(2^n)$으로 나타낼 수 있다. 모든 조합을 생성하는 수식은 다음과 같이 나타낼 수 있으며, $\sum_{0 \le k \le n} \binom{n}{k} = 2^k$, 이는 파스칼 삼각형 이상 계수의 n번째 행의 합계로 계산할 수도 있다. 모든 조합의 결과(부분 집합)는 값을 포함하지 않는 집합인 0번째 집합부터 n개의 모든 원소를 포함하는 $2^n - 1$번째 집합까지 가질 수 있다. 항목의 개수 n이 증가함에 따라 전체 경우의 수는 지수 함수적으로 늘어난다.

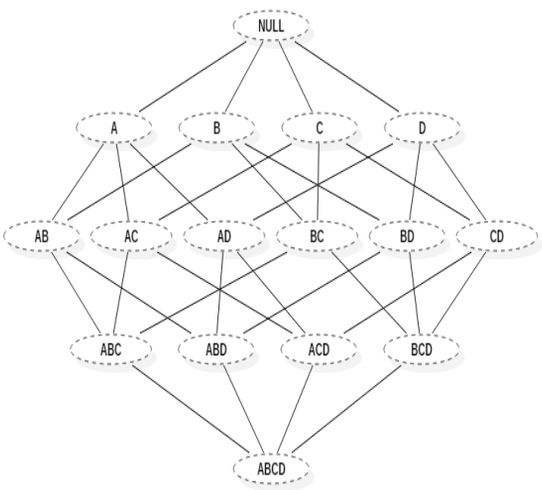


Fig. 1. The Number of All Possible Cases Generated by 4 Items

조합의 모든 경우의 수를 생성하는 알고리즘에는 크게 두 가지 방식이 사용되고 있다[5]. 그 중 하나는 동일한 재귀 구조를 반복 호출하여 모든 경우의 수를 생성하는 함수를 사용하는 방법이고, 또 다른 방법은 현재 경우의 수에 대한 개체를 생성한 이후 다음 개체를 호출하는 Next라는 함수를 사용하는 방법으로서 반복적 구조이다. Next함수는 C++를 위한 표준 템플릿 라이브러리(STL: Standard Template Library)에서 제공되며, Table 1과 같은 구조로 이루어져 있다. 여기서 done은 전역 변수로서 Next 함수가 다음 경우의 수에 대한 개체를 생성할 수 있을 때 True로 설정되는 값이다.

Table 1. Next Function Structure

```
Initialize; {includes done := false }
repeat
        Print It;
        Next;
until done;
```

모든 경우의 수를 생성하기 위하여 재귀 함수 호출 알고리즘과 Next를 이용하는 반복적 알고리즘 중 과거의 많은 연구에서는 Next를 이용하는 반복 알고리즘을 선호하였다[9]. 대표적인 이유 중 하나는 재귀 함수가 잦은 프로시저를 호출하고, 이 과정에서 오버 헤드가 발생하여 반복적 방법에 비해 재귀함수 호출이 느리다는 것 때문이었다. 그러나 최근 기술의 발달로 인하여 프로시저 호출이 오버 헤드를 발생 할 것이라고 가정할 필요성이 낮아졌다. 하드웨어에 따라 프로시저 호출의 속도가 높아졌기 때문에 단순히 재귀 함수가 오버 헤드를 발생시키는 이유로 배제할 필요가 없어진 것이다.

조합의 모든 경우의 수를 구하는 문제가 아닌 일부를 구하는 문제에서는 재귀 함수 방식보다 Next를 이용하는 방식이 좋은 선택이 된다. 그 이유는 재귀 함수가 조합을 생성하기 위해 반드시 순차적으로 진행되어야 하는 특징 때문이다. 조합의 일부분만을 필요로 하는 경우에도 전체를 구하는 경우와 동일하게 순차적으로 재귀 함수를 진행한다면 불필요한 추가 시간을 소요하게 된다. 이때는 Next 함수가 좋은 선택의 대안이 될 수 있다. 그러나 조합의 모든 경우의 수를 생성해야 하는 경우, 즉, 한 순간에 모든 경우의 수 중 하나만을 생성하는 경우나 ${}_nC_r$인 조합을 구하는 문제를 제외하고, 모든 경우의 수 조합세트를 완전히 필요로 하는 경우에는 재귀 함수 방식도 좋은 후보가 될 수 있다. 특히 모든 경우의 수를 자식 부모관계를 포함한 트리의 구조로 표현할 경우에는 재귀 함수 방식이 좋은 후보가 된다. 다만 모든 경우의 수의 트리를 구성하기 위해서는 높은 시간 복잡도가 요구되므로 이를 줄이기 위한 노력이 필요하다.

GPU의 발달과 병렬 환경의 보편화에 따라 이를 이용하여 기존 순차적 알고리즘의 시간 복잡도를 줄여보려는 시도가 늘고 있다. 따라서 본 연구에서는 GPU에서 CUDA를 활용하여 모든 경우의 수를 생성하기 위해 CPU와 GPU가 협력하는 병렬화 방법을 제안하고 실험을 통하여 성능을 검증한다. 이때 본 연구에서는 CPU환경과 GPU환경 모두에서 재귀적 방법을 통하여 조합의 모든 경우의 수를 생성한다.

조합의 모든 경우의 수를 생성하는 방법에 대한 연구는 다양한 분야에서 필요할 뿐만 아니라 빈도도 증가하고 있다. 그러나 대부분의 연구는 시간 복잡도를 줄이기 위해서 경우에 따라 주요한 일부 경우의 수를 생성하는 방향으로 진행되며, 순수하게 조합의 모든 경우의 수를 생성하는 알고리즘 자체에 대한 연구는 거의 없는 실정이다. 특히 GPU를 활용한 연구가 활발한 현재에도 순수한 조합의 모든 경우의 수를 생성하는 방법에 대한 연구는 온라인 커뮤니티 등에서 극히 제한적으로 연구되고 있다[9-11]. 병렬 하드웨어 중 GPU의 특징은 CPU와의 협업이 효율적이며, 각 스레드에서 같은 크기의 작업을 수행할 때 그 효율이 극대화 된다. 따라서 본 연구에서는 조합의 모든 경우의 수를 효율적으로 생성하기 위하여 GPU의 특징을 고려하고, GPU의 병렬 구조만을 활용한 방법이 아닌 CPU와 GPU가 협업하는 알고리즘을 제안한다.

## 3. CPU와 GPU의 협업 방법의 제안

조합의 모든 경우를 생성하는 알고리즘은 단순한 연산이 반복적으로 일어나는 구조로 연산 능력이 강한 GPGPU에 적합하다고 할 수 있다. 그러나 조합의 모든 경우의 수를 구하는 과정이 계층 순서로 순차적으로 진행되는 연산의 특성상 병렬화에 효율적이지 못하다. 따라서 본 논문에서는 조합의 모든 경우의 수를 구하는 과정을 효율적으로 병렬화 하고 전체 시간 복잡도를 줄이기 위해 CPU에서 병렬화를 위한 전처리 과정과 이에 뒤따르는 GPU 환경의 알고리즘을 제안한다.

본 연구에서 제안한 알고리즘은 Fig. 2와 같이 CPU와 GPU의 협업을 통해 진행된다. GPGPU의 SIMT 특징을 만족하기 위하여 동일한 명령어를 수행할 유사한 크기의 작업이 스레드에 할당되어야 효율적인 실행이 가능하다[12].

모든 경우의 수를 구하는 방법을 트리를 통해 진행하면 Fig. 3과 같이 체계적으로 만들 수 있다. 이 때, 트리를 구성하는 과정은 레벨별로 순차적으로 이루어지는데, 트리의 레

벨 중 어느 시점에서 병렬화를 수행할 지가 알고리즘의 성능을 결정하는 관건이 된다. 적은 개수의 항목에 대해 병렬화를 수행하면 GPU에서 하나의 스레드가 수행해야할 부담이 크기 때문에 병렬화를 진행하는 효율이 떨어지고, 많은 개수의 항목집합에 대해 병렬화를 수행하려면 많은 개수의 항목 집합을 생성해야 하는 CPU의 부담이 커지게 된다.

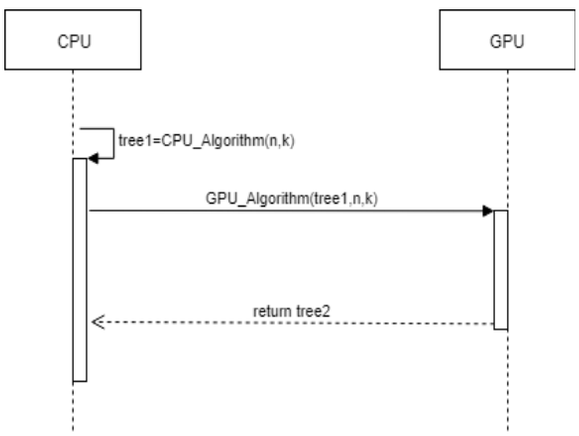


Fig. 2. The Proposed Generator for the Number of All Possible Cases

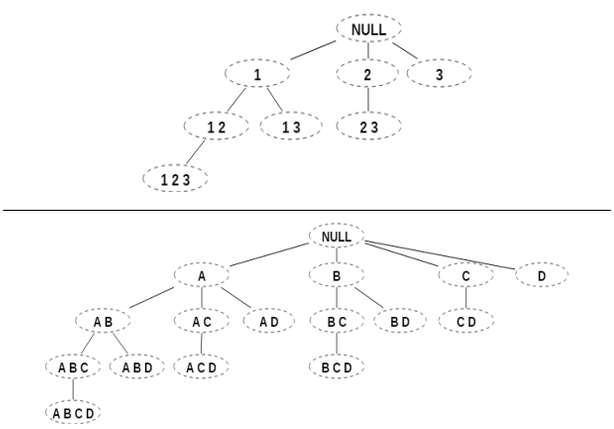


Fig. 3. A Systematic Way to Find the Number of All Possible Cases Using Tree
(above: 3 items, below: 4 items)

뿐만 아니라 Fig. 3과 같이 모든 경우의 항목집합은 좌측에 노드가 편중된 트리의 모습을 가지게 된다. 때문에 동일한 레벨의 노드에 대해 병렬화를 수행하면 스레드들의 수행시간이 각기 달라지기 때문에 효율성을 기대하기 어렵다. GPGPU의 병렬화는 각 코어가 독립성을 유지한 채 비슷한 크기의 연산을 할 수 있도록 메모리 할당을 해줘야 효율적이다.

본 논문에서는 GPU의 각 코어별 처리의 독립성을 유지하고 비슷한 크기의 작업을 할당하기 위해 Fig. 4와 같은 방법으로 CPU와 GPU의 작업을 분리 수행한다. Fig. 4의 구분 표시의 상단 부분(CPU part)은 CPU에서 생성될 조합의 결과물이며, 하단 부분(GPU part)은 GPU의 스레드들이 생성할 조합의 결과물이다.

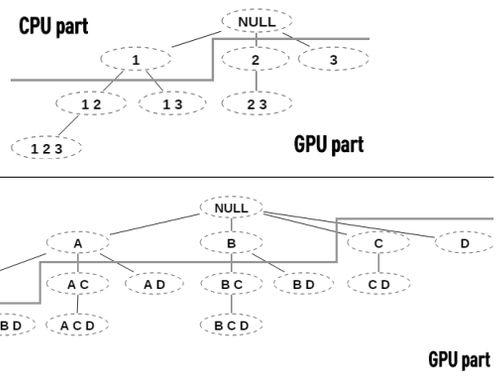


Fig. 4. CPU and GPU Task Separation
(above: 3 items, below: 4 items)

본 연구에서 제시한 예제 중 항목이 4개인 경우에는 CPU에서 절반인 2개의 항목 A와 B에 대하여 조합의 수를 생성하는 책임을 담당하였다. 이러한 원리를 적용하여 만약 n개의 항목에 대해 본 연구에서 제안한 알고리즘을 수행한다면 cpu가 생성해야 할 책임을 가지는 항목은 절반인 $\frac{n}{2}$개를 대상으로 만들게 되고, 그 결과는 $2^{\frac{n}{2}}$개가 된다. 이는 알고리즘을 적용할 환경에 따라 유동적으로 변경하여 적용 가능할 것이다. 하드웨어 등 실험 환경을 검토하였을 때 GPU와 CPU의 실험 환경의 효율성에 따라 유리한 비율로 조정하여 설정할 수 있다.

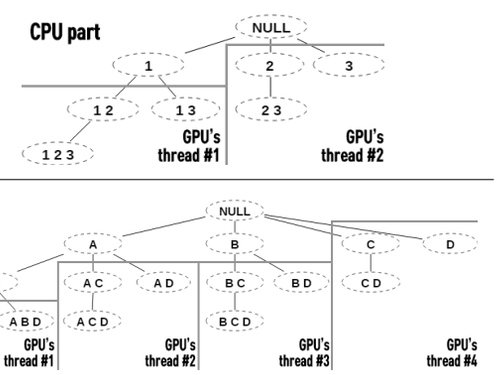


Fig. 5. The Division of Operations that a Single Thread in the GPU will be Responsible
(above: 3 items, below: 4 items)

Fig. 5는 하단 GPU에서 담당하게 될 대상 중 한 단위의 스레드가 담당하게 될 연산을 나타내기 위하여 구역을 나눈 모습이다. 본 연구에서 나타낸 예제 중 항목이 4개인 연산에서는 총 4개의 스레드가 역할을 나누어 각각 3개의 경우의 수를 생성한다. 최종적으로 4개의 스레드에서 생성한 총 12개의 결과물이 CPU에 반납되고, CPU에서 생성한 4개의 결과물과 합쳐서 16개의 경우의 수가 생성되게 된다. GPU에서 각각의 스레드는 작업이 독립적으로 수행될 수 있을 뿐만 아니라 같은 양의 결과물을 생성하게 됨으로 동일한 실행시간

이 걸릴 것으로 기대할 수 있다. CPU와 GPU에서 수행될 연산에 대해 좀 더 간단히 표현하면 Fig. 6과 같다.

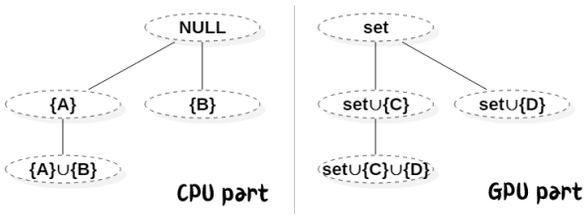


Fig. 6. (Left) For the Four Items, the Operation will Perform on the CPU; (Right) For the Four Items, the Operation that One Thread will Perform on the GPU

Fig. 6의 좌측 부분이 CPU에서 수행될 연산, 우측 부분이 GPU에서 하나의 스레드가 담당하게 될 연산이 된다. Fig. 6의 좌측 트리의 루트는 공집합이고, Fig. 6의 우측 트리의 루트는 CPU에서 주어진다. 우측 부분의 자식노드들은 루트 노드에서 주어진 경우 집합을 포함 하여 하부 집합을 구성한다. 예를 들어, GPU의 어떤 스레드가 CPU에서 전달받은 집합이 {A,B}라면 생성되는 자식 노드의 값은 {A,B,C}, {A,B,D}, {A,B,C,D} 가 된다.

즉, 본 연구에서는 각 스레드가 동일한 크기의 결과물을 생성할 수 있도록 작업 크기의 균형을 잡기 위하여 CPU를 활용한다. CPU에서 생성된 결과는 GPU의 각 스레드가 동일한 크기의 연산을 수행하기 위해 유도하는 역할을 한다.

**Algorithm 1:** CPU Algorithm

```
CPU_Algorithm(n, k):
Inputs:   Array of Elements(n), k
Outputs:  Result tree of CPU algorithm
Steps:
1.  *tree=maketree();
2.  tree.insert_node(null);
3.  *queue=init();
4.  for (i=0;i<k;i++) {
5.     result=root.insert_child(i);
6.     queue.in(result);
7.  }
8.  while(node=queue.out()){
9.     for(j=node.lastvalue()+1 ; j<k ; j++){
10.       result=node.insert_child(j);
11.       queue.in(result);
12.    }
13. }
```

이는 스레드의 작업 불균형으로 인해 발생 가능한 대기 시간 등의 오버헤드를 최소화 한다.

조합의 모든 경우의 수를 생성하는 과정의 시간 복잡도는 $O(2^n)$인데 반해 본 연구에서 제안한 CPU와 GPU가 협업하여 진행하는 병렬화 기법의 시간 복잡도는 Fig. 6의 좌측 CPU 부분이 $O(2^k)$라면 GPU의 시간 복잡도는 $O(2^{n-k})$이 된다. 이때 k는 n보다 작은 양수로 전체 시간 복잡도 $O(2^k+2^{n-k})$로 감소하게 된다. k의 값에 따라 시간 복잡도는 최악의 경우 $O(2^n)$ 최선의 경우$O(2^{\frac{n}{2}})$로 표현이 가능하며, 이는 조합의 모든 경우의 수를 생성하는 알고리즘의 시간 복잡도인 $O(2^n)$보다 작거나 같은 크기를 가진다.

**Algorithm 2:** GPU Algorithm

```
GPU_Algorithm(set, n, k):
Inputs:   Result of CPU algorithm(set), k, n
Outputs:  Result tree of GPU algorithm
Steps:
1. *queue=init();
2. for (i=k;i<n;i++) {
3.    result=set.insert_child(i);
4.    queue.in(result);
5. }
6. while(node=queue.out()){
7.    for(j=node.lastvalue()+1 ; j<n ; j++){
8.       result=node.insert_child(j);
9.       queue.in(result);
10.   }
11. }
```

이 때 k는 n보다 작은 값으로 사용자에 의해 주어질 수 있다. 이는 하드웨어의 제약인 GPU의 스레드 개수와 알고리즘에 의해 생성될 결과물과 저장소의 크기에 따라 고려되어 질 수 있다. 하드웨어의 제약이 없다면 가장 이상적인 k의 값은 $\frac{n}{2}$ 이 되고, 시간 복잡도는 $O(2^{\frac{n}{2}})$이 된다.

본 논문에서 제안한 방법으로 조합의 모든 경우의 수를 생성하는 방법을 CPU와 GPU와 협업해서 병렬화하기 위한 알고리즘은 <Algorithm 1,2>와 같다. <Algorithm 1>에서 먼저 CPU 작업을 통하여 GPU의 각 스레드별 할 일을 분리시키기 위해 전처리 작업을 수행한다. <Algorithm 1>에서 4-7번째 줄은 트리에 CPU가 담당하는 두 번째 레벨의 자식 노드를 추가하는 과정이다. Fig. 4 상단에서는 “1”노드가 아래 트리에서는 “A”, “B” 노드가 추가되고 큐에 저장된다. <Algorithm 1>에서 6번째 줄과 같이 큐에 저장된 노드는 큐가 비어 있을 때까지 순서대로 빼내어진다. 이 때 꺼내어진 노드의 마지막 값 인덱스보다 크고 k보다 작은 추가 값이 더해진 노드들이 꺼내어진 노드의 자식 노드가 되고 큐에 삽입된다. 예를 들어, Fig. 6의 좌단 트리에서 k는 2이고 큐에서 빠진 노드가 “A”일 때, “A”의 인덱스보다 크고 k의 값보다 작은 “B”가 더해져 “AB”노드가 “A”의 자식 노드가 된다.

<Algorithm 2>는 CPU 전처리 과정에서 생성된 집합 set을 루트로 하여, 이에 추가적으로 GPU의 스레드 별로 경우의 수가 추가할 수 있도록 연산이 수행된다. CPU에서 전처리 과정을 통해 생성된 경우의 수들은 GPU의 메모리로 복사한다. 각 스레드는 CPU에서 생성된 경우의 수 중 하나의 set를 배정받고 별도의 큐를 통해 k보다 같거나 크고 n보다 작은 인덱스의 값에 대해 CPU와 동일한 과정을 수행한다. <Algorithm 2>에서 2-5번째 줄은 Fig. 6 우측 트리에서 2번째 레벨에 해당하는 자식을 추가하는 과정이다. 전체 항목이

4개이고 k가 2일 경우 GPU는 3, 4번째 항목에 대해서 경우의 수를 생성한다. 트리의 2번째 레벨에 추가된 노드는 스레드의 큐에 삽입되고, <Algorithm 2>의 6-11번째 줄에서 큐가 빌 때까지 반복되며 CPU 과정과 같이 자식 노드를 생성한다.

GPU는 CPU에서 넘어 온 항목집합의 개수만큼 스레드를 할당한다. 각 스레드는 알고리즘을 수행하여 생성된 정보를 CPU에 반환해 CPU에서 GPU의 결과를 합쳐 최종 결과를 반납한다. CPU에서 병렬화 이전에 생성될 경우의 수의 개수가 $2^k$개라면, GPU에 할당될 스레드 또한 $2^k$개다. 이 때 하나의 스레드가 생성해야하는 아이템의 개수 CPU에서 전달받은 경우와 합산하여 $2^{n-k}$개가 된다. $2^k$개의 스레드가 모두 $2^{n-k}$의 아이템을 최종 생성하므로 전체 생성되는 아이템의 수를 합산하면 Equation (1)과 같이 된다. 이는 조합의 모든 경우의 수인 $2^n$과 일치한다.

$$2^{n-k} \times 2^k = 2^{n-k+k} = 2^n \quad (1)$$

## 4. 실험적 비교

본 논문에서 제안한 알고리즘의 성능을 비교 평가하기 위해 조합의 모든 경우의 수를 CPU에서 수행한 알고리즘 2개(반복적 방법과 재귀적 방법), Ye and Chiang이 제안한 병렬 알고리즘[7]과 본 연구에서 제안한 CPU와 GPU가 협업하여 수행한 알고리즘을 CPU와 GPU에서 각각 구현하였다. 실험은 4GB RAM의 인텔코어 i5-2500K 3.3GHz의 PC와 GeForce GTX 670 GPU 수행되었다. 해당 GPU는 2GB의 글로벌 메모리를 가지고 있다.

실험 데이터 n은 1에서 40까지를 대상으로 하여 CPU와 GPU에서의 수행 시간을 측정하고 모든 경우의 수에 대한 조합이 정상적으로 생성되었는지 확인한 뒤, 실행 시간을 비교하였다. 모든 경우의 수를 생성하는 알고리즘은 C++를 통해 CPU에서 구현하거나, GPU환경에서는 CUDA를 활용하여 구현하여 시간을 측정하였다. 실험에 대한 비교는 동일한 데이터에 대해 50회를 시도하고 그 평균을 사용하여 실험 결과를 구성하였다.

Table 2. Experimental Results (Unit: ms)

| # of Element | # of Combination | Next Function In CPU | Recursive method in CPU | Ye & Chiang Algorithm | Proposed Parallel Algorithm |
|---|---|---|---|---|---|
| 10 | 1,024 | 5 | 1 | 2 | 2 |
| 15 | 32,768 | 117 | 1 | 2 | 2 |
| 20 | 1,048,576 | 3,388 | 36 | 2 | 2 |
| 25 | 3.4E+07 | 96,173 | 1,332 | 985 | 4 |
| 30 | 1.1E+09 | excess | 47,963 | 35,111 | 6 |
| 35 | 3.4E+10 | excess | excess | excess | 25 |
| 40 | 1.1E+12 | excess | excess | excess | 52 |

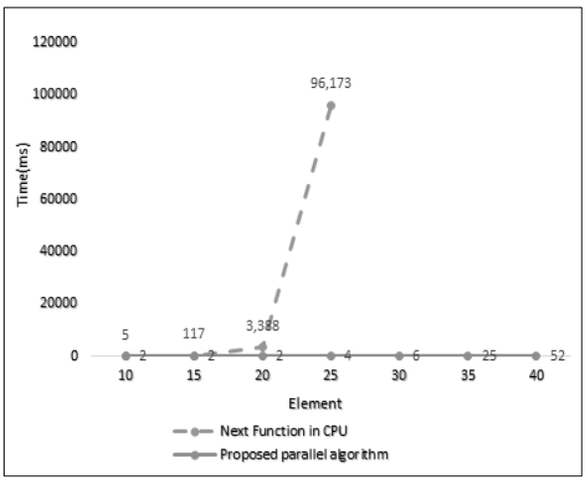


Fig. 7. Comparison of Experimental Results Between NEXT Function and Proposed Algorithm

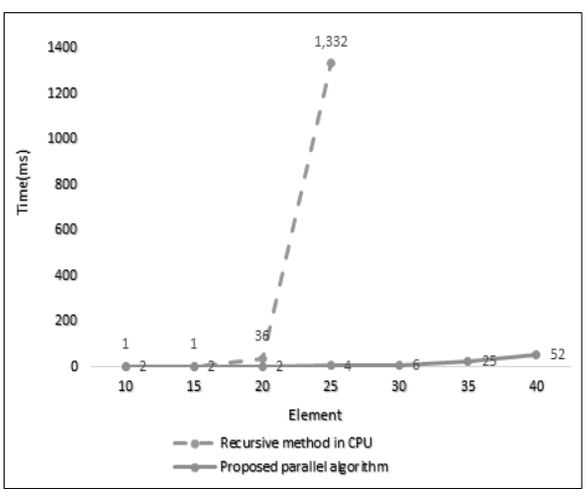


Fig. 8. Comparison of Experimental Results Between Recursive Method and Proposed Algorithm

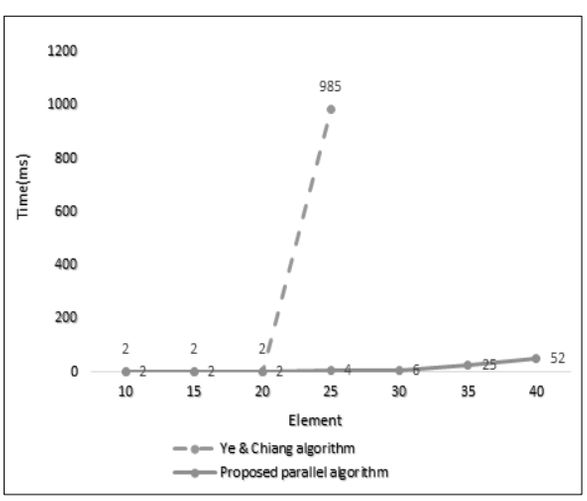


Fig. 9. Comparison of Experimental Results Between Ye & Chiang Algorithm and Proposed Algorithm

Table 2와 Figs. 7-9는 CPU와 GPU 상에서 수행된 알고리즘들의 실험 결과를 보여준다. 각 그림에서 X축은 Y축은 아이템의 개수와 실행 시간을 의미한다. 실험 결과를 관찰하면 아이템의 개수가 늘어날수록 실행 시간이 늘어나고 본 연구에서 제안한 방법과 비교 방법들 간의 시간 차이가 많이 나고 있어, 아이템의 개수가 늘어날수록 제안한 협업 알고리즘이 타 알고리즘들에 비해 우수함을 보였다.

실험 결과에 포함된 비교 대상의 방법들이 특정 아이템 개수가 되기까지 기하급수적으로 증가하는 실험 시간을 갖다가 각각의 방법마다 차이는 있으나 특정 개수를 지나면 측정할 수 있는 실행 시간을 초과(Excess)하는 결과를 보였다. Ye and Chiang이 제안한 병렬 알고리즘 또한 35개의 Element에서 과도한 실행 시간이 발생하였다. Ye and Chiang이 제안한 병렬 알고리즘이 본 연구에서 제안한 기법과 유사한 CPU & GPU 협업 알고리즘임에도 불구하고 본 연구에서 제안한 협업 기법과 큰 차이가 나는 이유는 본 연구와는 달리 GPU에서 실행하는 서브 작업들의 크기가 다르기 때문이다. Ye and Chiang이 제안한 병렬 알고리즘에서는 가장 많은 작업을 할당받는 특정 스레드가 존재하며, 이는 특정 스레드의 작업이 끝날 때까지 다른 스레드가 대기하는 현상을 초래하게 된다.

Table 3. CPU & GPU Collaboration Algorithms'S Experimental Results (Unit: ms)

| # of Element | | 10 | 15 | 20 | 25 | 30 | 35 | 40 |
|---|---|---|---|---|---|---|---|---|
| Ye & Chiang Algorithm | CPU | 0.2 | 0.3 | 0.4 | 1.9 | 3.3 | 5.8 | 10.8 |
| | GPU | 1.5 | 1.8 | 2.1 | 983 | 35,118.1 | excess | excess |
| Proposed Parallel Algorithm | CPU | 0.7 | 0.9 | 1.2 | 1.9 | 2.4 | 10.1 | 23.0 |
| | GPU | 0.8 | 0.9 | 1.2 | 2.0 | 3.8 | 14.8 | 28.6 |

CPU와 GPU가 협업하는 기법인 Ye and Chiang이 제안한 병렬 알고리즘과 본 연구에서 제안한 협업 알고리즘의 실행 시간을 더 자세히 비교하면 Table 3과 같다. 이 때 실험시간은 CPU와 GPU간의 데이터를 이동하는 등의 전처리 시간은 제외하였다. 그 결과 본 연구에서 제안한 기법의 CPU와 GPU에서 실행시간은 아이템 개수의 크기에 관계없이 비슷한 크기를 가지는 것으로 측정되었다. 그러나 Ye and Chiang이 제안한 병렬 알고리즘은 GPU에서 수행해야 하는 연산의 부담이 급격하게 증가하여 이에 따른 수행 시간의 불균형이 크게 나타났다. 35개의 Element에서 GPU의 실행시간은 Excess 결과를 보였다. Ye and Chiang이 제안한 병렬 알고리즘은 CPU에서만 수행한 알고리즘에 비하여 효율적이나 CPU와 GPU간의 Work Balance가 맞지 않아 본 연구에서 제안한 알고리즘에 비해 성능이 떨어진 결과를 보인 것으로 분석된다.

30개의 입력 값(Elements)은 약 10억 개의 조합 결과가 생성되고, 40개를 대상으로 하는 경우에는 약 1조 개, 50개의 경우는 약 1,100조 개 이상의 조합 결과가 생성된다. 1개의 조합 결과가 4바이트의 Integer라고 가정한다면 50개의 입력 값의 모든 조합결과는 4페타바이트(PB) 만큼 증가한다. 이는 매우 방대한 메모리 사용 문제를 초래 할 수 있기 때문에 실제 응용에서는 메모리 용량을 고려하여 알고리즘을 수행하여야 한다.

본 실험에서는 조합의 생성 CPU와 GPU간의 메모리 이동 시간은 실험 시간에 포함 시키지 않았다. 큰 개수의 입력 값을 수행할 경우에는 GPU 메모리 한계 특성으로 인해 분할하여 반복 수행해야 하고, 조합의 생성시간 이외로 추가적인 CPU와 GPU간에 메모리 이동 시간이 발생될 수 있다. 추후 GPU의 메모리의 제약과 CPU와 GPU간의 데이터 이동 속도의 문제가 개선된다면 이 문제 또한 완화될 것이다.

## 5. 결론 및 향후 과제

조합에서 모든 경우의 수를 생성하는 알고리즘은 기하급수적인 시간 복잡도를 가지는 단점이 있다. GPU가 보급되면서 모든 경우의 수를 생성하는 알고리즘의 병렬화를 통해 시간 복잡도를 줄이려는 시도가 있어왔다. 그러나 조합의 모든 경우의 수를 생성하는 알고리즘의 순차적 특징으로 인해 대부분 이를 우회하거나 조합의 일부 경우의 수를 생성하는 알고리즘의 병렬화에 대해 연구가 수행되었다.

본 연구에서는 모든 경우의 수를 생성하기 위하여 CPU와 GPU의 협업 알고리즘을 제안하였다. 모든 경우의 수를 트리를 통해 체계적으로 생성할 경우 알고리즘은 트리의 레벨별로 순차적으로 진행되나 각 서브트리가 독립적이기 때문에 병렬 처리가 가능하다. 그러나 조합을 위한 트리는 편중되는 모습을 이루기 때문에 균형 있는 작업 크기의 병렬화에 문제가 있다. 본 연구에서는 GPU 병렬환경의 특징에 따라 스레드가 비슷한 크기의 작업을 할당 받기 위하여 CPU와 GPU의 협업 작업을 제안한다. 우선 CPU 작업이 수행되고 그 결과물로서 GPU의 각 스레드는 동일한 크기의 작업을 할당받을 수 있다. GPU 작업이 끝나면 모든 경우의 수가 생성된다. 제안된 알고리즘은 실험을 통하여 다른 알고리즘들과 성능을 비교하였다. 실험 결과 제안한 CPU와 GPU의 협업 알고리즘이 아이템의 개수가 커질수록 괄목할 만한 시간 개선을 보였다.

본 연구에서 제안한 알고리즘은 GPU의 메모리와 스레드 개수의 제한은 고려하지 않았다. GPU에서 하나의 스레드가 담당할 연산과 메모리의 크기는 CPU가 수행해야 할 단위의 크기에 비례한다. 조합에 포함된 항목의 개수가 커질수록 사용하는 스레드의 개수가 커지게 되고 이는 CPU가 담당할 연산과 GPU 스레드 전체가 수행해야 할 총 연산은 상당한 차이를 초래하게 된다. 이는 GPU의 스레드 개수의 제약과 메모리 제약 등의 문제를 고려하여 추후에 연구 보완 되어야 할 것이다. 이를 토대로 더 다양한 아이템의 개수와 GPU의 전체 스레드 개수를 고려한 경우에 대해 CPU와 GPU에서의 성능 비교 검토가 필요하다.

## References


[1] R. Agrawal and R. Srikant, “Fast Algorithms for Mining Association Rules,” *20th International Conference on Very Large Data Bases*, pp.487-499, 1994.

[2] Y. Jiao, “Research of an improved apriori algorithm in data mining association rules,” *International Journal of Computer and Communication Engineering*, Vol.2, No.1, 2013.

[3] H. Kang, K. Yang, C. Kim, Y. Rhee and B. Lee, “A Time-based Apriori Algorithm,” *The transactions of The Korean Institute of Electrical Engineers*, Vol.59, No.7, pp.1327-1331, 2010.

[4] S. Ko and J. Lee, “Weighted Bayesian Automatic Document Categorization Based on Association Word Knowledge Base by Apriori Algorithm,” *Journal of Korea Multimedia Society*, Vol.4, No.2, pp.171-181, 2001.

[5] F. Ruskey, “Combinatorial generation.” Preliminary working draft, University of Victoria, Victoria, BC, Canada, 2003.

[6] K. C. Wei, X. Sun, H. Chu, and C. C. Wu, “Reconstructing permutation table to improve the Tabu Search for the PFSP on GPU.” *The Journal of Supercomputing*, Vol.73, No.11, pp.4711-4738, 2017.

[7] Y. Ye and C. C. Chiang, “A Parallel Apriori Algorithm for Frequent Itemsets Mining,” *Proceedings of the Fourth International Conference on Software Engineering Research*, 2006.

[8] NVIDIA. “CUDA C Programming Guide version 5.0,” NVIDIA, 2012.

[9] Permutations with CUDA and OpenCL [Internet], https://www.codeproject.com/Articles/380399/Permutations-with-CUDA-and-OpenCL, 2016.

[10] F. Bodon, “A Fast Apriori Implementation,” IEEE ICDM Workshop on Frequent Itemset Mining Implementations, 2003.

[11] F. Bodon, “Surprising Results of Trie-based FIM Algorithm,” IEEE ICDM Workshop on Frequent Itemset Mining Implementations, 2004.

[12] W. Ahn, C. Lee, and C. Yoo, “Changes in the performance of the CUDA simple code according to the configuration of blocks and threads,” *Proc. of the Korea Information Science Society Fall Conference*, Vol.39, No.2(A), pp.19-21, 2012.



### 손 기 봉

https://orcid.org/0000-0001-5720-546X

e-mail : gukb@kumoh.ac.kr

2012년 가톨릭대학교 컴퓨터정보공학부 (학사)

2015년 조선대학교 전기·전자·통신교육 (석사)

2015년~현 재 금오공과대학교 컴퓨터공학과 박사과정

관심분야 : 분산처리, 데이터 마이닝



### 손 민 영

https://orcid.org/0000-0001-9099-0180

e-mail : son0804@kumoh.ac.kr

2008년 고려대학교 컴퓨터정보학과(학사)

2010년 고려대학교 정보경영공학과(석사)

2017년 금오공과대학교 컴퓨터공학과 (박사)

관심분야 : 네트워크, 분산처리, 그래프, 데이터마이닝



### 김 영 학

https://orcid.org/0000-0003-4232-4612

e-mail : kimyh@kumoh.ac.kr

1984년 금오공과대학교 전자공학과(학사)

1989년 서강대학교 전자계산학과(석사)

1997년 서강대학교 전자계산학과(박사)

1999년~현 재 금오공과대학교 컴퓨터공학과 교수

관심분야 : 병렬알고리즘, 분산처리, 임베디드시스템