Journal of KIIT. Vol. 16, No. 1, pp. 19-26 Jan. 31, 2018. pISSN 1598-8619, eISSN 2093-7571
http://dx.doi.org/10.14801/jkiit.2018.16.1.19

최소 좁은 세상 셀룰러 유전알고리즘의 CUDA 병렬화

김영태*, 강태원**

CUDA Parallelization of the Smallest-Small-World Cellular Genetic Algorithms

Youngtae Kim*, Taewon Kang**

요 약

본 논문에서는 범용 계산을 위한 NVIDIA GPU(Graphic Processing Unit) 플랫폼인 CUDA를 사용하여 최소 좁은 세상 셀룰러 유전알고리즘(SSWCGA)를 병렬화하고 성능을 평가한다. SSWCGA는 모집단이 격자형 네트워크로 구성되어 병렬화하기에 적합하다. 병렬 프로그램에서는 난수 발생의 효율을 높이기 위하여 난수 도메인을 그리드 상에서 병렬로 생성하여 도메인 계산을 병렬로 실행하도록 하였다. 또한 모집단 개체의 최대값을 계산하기 위하여 캐시메모리 역할을 하는 공유메모리를 사용하여 성능을 높였다. 평가 결과 동일한 크기의 모집단에 대해 동일한 비율로 최적해를 찾는 것을 확인 할 수 있었다. 수행시간 측면에서 단일 CPU 프로세스에 비해 대략 10% 내외의 시간에 최적해를 찾을 수 있음을 알 수 있었다.

Abstract

In this paper, we implement and evaluate a CUDA parallel program of the Smallest-Small-World Cellular Genetic Algorithms for the NVIDIA general GPU(Graphic Processing Unit) platform. The population set of the SSWCGAs is a grid network, therefore it is suitable for parallelization. In the parallel program, random number domains are generated in parallel on each grid, and therefore domains are calculated in parallel as well. We used a shared memory(cache memory on GPU) to calculate the domain reduction efficiently. As a result of the evaluation, the optimal solution was not found at the same ratio for the same size population, but also efficiently within 10% of the time compared to the single CPU processor.

Keywords
smallest-small-world CGA, CUDA, GPU, random number

* 강릉원주대학교 컴퓨터공학과 교수
- ORCID: http://orcid.org/0000-0002-8125-6686
** 강릉원주대학교 컴퓨터공학과 교수 (교신저자)
- ORCID: http://orcid.org/0000-0003-4343-5517

· Received: Oct. 23, 2017, Revised: Nov. 17, 2017, Accepted: Nov. 20, 2017
· Corresponding Author: Taewon Kang
Dept. of Comuuter Science & Engineering, Gangneung-Wonju National Univ.
150, Namwon-ro, Heungeop-myeon, Wonju-si, Gangwon-do, 26403
Tel.: +82-33-760-8666, Email: twkang@gwnu.ac.kr

Ⅰ. 서 론

자연과 사회의 문제를 다루는 많은 모델이 병렬처리가 가능하고, 인간의 뇌 역시 병렬처리 장치로 볼 수 있다. 2006년 NVIDIA의 CUDA(Compute Unified Device Architecture)가 발표되면서 병렬처리 기술은 새로운 전환을 맞이하였다. CUDA는 고가의 슈퍼컴퓨터가 아니라 보통의 개인용 PC에서 병렬처리를 가능하게 한다. CUDA는 병렬처리를 위한 복잡한 하드웨어 및 프로그래밍 지식 없이 C 언어 등 고급 언어에 대한 지식만으로 쉽게 구현할 수 있다. 이러한 이유로 컴퓨터가 사용된 이후 오랫동안 여러 가지 환경에서 개발되어온 많은 문제들이 최근 수년 동안 CUDA를 사용하여 병렬화되고 있다.

복잡계 네트워크는 자연과 사회현상을 다루는 중요한 도구로 과학과 공학뿐 아니라 인문학과 사회과학의 분야에서 많은 연구가 진행되고 있다[1][2]. 복잡계 네트워크의 특성을 반영한 최소 좁은 세상 셀룰러유전알고리즘(SSWCGAs, Smallest-Small-World Cellular Genetic Algorithms)[3]은 대표적으로 병렬화가 가능한 알고리즘이다. 본 논문에서는 범용 GPU 플랫폼인 NVIDIA의 CUDA를 사용하여 SSWCGAs을 병렬화하였다. 2장에서는 SSWCGAs와 CUDA 병렬 프로그램의 구현을, 3장에서는 성능 분석을 하고 4장에서 결론으로 맺는다.

Ⅱ. CUDA 병렬 프로그램의 구현

이 장에서는 SSWCGAs의 CUDA 병렬 프로그램 구현에 대하여 설명한다.

2.1 최소 좁은 세상 셀룰러 유전알고리즘

좁은 세상 네트워크는 임의 네트워크의 일종이다[2]. 임의 네트워크는 두 노드 사이의 경로 길이가 짧고 클러스터링 계수 역시 매우 작다. 하지만 좁은 세상 네트워크는 대부분의 이웃 노드들이 서로 연결되어 있으며, 임의의 두 노드가 짧은 단계 만에 도달할 수 있는 경로를 갖는 네트워크이다. 즉, 클러스터링 계수는 크면서, 상대적으로 임의의 두 노드 사이의 평균 경로 길이가 매우 짧은 것이다[2][4][5]. 최소 좁은 세상 셀룰러 유전알고리즘[3][6]은 셀룰러 유전알고리즘[7]의 모집단을 좁은 세상 네트워크 형태로 구성하여 높은 클러스터링 계수를 유지하면서 짧은 평균 경로 길이를 갖는 위상을 갖도록 한 것이다. 이것은 격자형 네트워크에 최소의 링크를 추가하고도, 높은 클러스터링 계수를 갖도록 하여 세부탐색 능력은 보통의 셀룰러 유전 알고리즘과 같지만, 상대적으로 짧은 평균경로 길이를 가지므로 멀리 떨어진 개체들의 유전적 상호작용이 활발하여 빠르게 전역 탐색이 이루어진다.

그림 1은 SSWCGAs의 수행 절차를 나타낸다[3].

```
//main loop
create_population();
do {
    eval_population();
    if(check_converge()) break;
    selection();
    crossover();
    mutation();
} while(!Max_Gen);
//selection process
selection() {
    for i=1 to width
    for j=1 to height
        neigh_ij = get_neighbor(node_ij);
        winners_ij = select(neigh_ij);
        add winners_ij to selected_list;
}
get_neighbor() {
    if center return {8 neighborhoods, shortcuts}
    if shortcut return {8 neighborhoods, center}
    otherwise return {8 neighborhoods}
}
//crossover process
crossover() {
    for i=1 to width
    for j=1 to height
        if (Pcross)
            make child1, child2 from selected_list;
            new_ij = better{child1,child2} | better{current_ij,
                                child1,child2}
}
//mutation process
mutation() {
    for i=1 to width
    for j=1 to height
        if (Pmut)
            mutate new_ij;
}
```

그림 1. SSWCGAs의 수행 절차
Fig. 1. Algorithm of the SSWCGAs

2.2 CUDA 계산 환경 및 병렬 프로그램의 인덱스 전환

병렬 프로그램은 순차 프로그램에서 반복을 병렬로 처리하여 속도를 개선하는 것이 가장 기본이다. 따라서 반복문의 인덱스를 전환하는 것이 효율적인 병렬 프로그램의 구현에 있어서 중요한 역할을 한다. 여기서는 CUDA 프로그램에서의 인덱스 전환에 대하여 설명한다.

GPU는 배열 형태의 쓰레드로 구성이 되어 있는 쓰레드 블록 단위로 병렬 계산을 하며 CUDA 환경에서는 하나의 명령으로 쓰레드 블록을 제어하는 SIMT(Single Instruction Multiple Threads)을 사용한다[8]. 일반적으로 데이터 도메인은 쓰레드 블록의 크기보다 크기 때문에 도메인을 쓰레드 블록 크기만큼의 서브 도메인으로 나눈다. 그림 2는 하나의 도메인과 쓰레드 블록의 관계를 나타낸다. 그림에서는 12×12 크기의 도메인을 크기가 3×4인 12(4×3)개의 쓰레드 블록으로 나누었으며, 각 쓰레드 블록에 할당된 서브 도메인을 병렬로 계산하여 쓰레드 블록의 크기인 12배의 이론적인 성능 향상을 보일 수 있다.

다음은 CUDA 병렬 프로그램의 인덱스 계산 방식을 이용한 병렬 프로그램의 구현에 대해 설명한다. 그림 2와 같은 크기의 도메인과 쓰레드 블록을 사용할 경우, 쓰레드 블록 변수 blockDim.x blockDim,y의 값은 각각 4와 3이고 blockIdx.x는 0, 1, 2이고 blockIdx.y 는 0, 1, 2, 3이다. 쓰레드 블록 내에서의 각 쓰레드의 인덱스 값 threadIdx,x는 0, 1, 2, 3이고 threadIdx,y는 0, 1, 2이다. 따라서 도메인의 모든 요소들이 인덱스 계산을 통하여 사용될 수 있다.

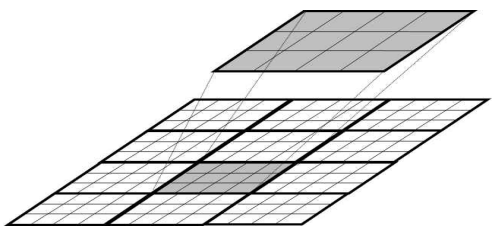


그림 2. 그리드(도메인)과 쓰레드 블록(서브 도메인)
Fig. 2. Grid(domain) and thread blocks(sub-domain)

그림 3은 돌연변이 연산을 병렬화한 것을 나타낸다. blockIdx와 threadIdx의 값들은 동시에 제공되기 때문에 프로그램에는 반복문 대신 도메인의 경계값을 비교하는 비교문으로 대체된다. k 반복문은 하나의 후보해에 대해 유전자 단위로 돌연변이 여부를 판정하는 것으로 도메인과는 상관이 없기 때문에 계속 존재하게 된다.

2.3 최대값 및 배열의 합 계산 - 리덕션

모집단 전체의 적합도 합이나 최적 개체를 찾기 위해서는 모집단 전체 즉 도메인 전체를 대상으로 처리가 이루어져야한다. 도메인의 모든 값들에서 최대값을 찾거나 합계를 구하는 경우에, 여러 쓰레드 블록에 값들이 분산되어 있기 때문에 다른 쓰레드에서 계산한 값을 처리해야 한다.

```
void Mutation() {
    int i, j, k;
    for (i=NB_DEPTH; i<P_SIZE_NB; i++){
        for (j=NB_DEPTH; j<P_SIZE_NB; j++) {
            for (k=0; k<CHROM_LEN; ++k) {
                if (flip((float)PMUT))
                pool[i][j].string[k] = ~pool[i][j].string[k] & 0x01;
            }
        }
    }
}
```

⟹

```
__global__ void Mutation(struct population *pool) {
    int i = blockIdx.x*blockDim.x+threadIdx.x;
    int j = blockIdx.y*blockDim.y+threadIdx.y;
    int k;
    if ((NB_DEPTH<=i) && (i<P_SIZE_NB)) {
        if ((NB_DEPTH<=j) && (j<P_SIZE_NB)) {
            for (k=0; k<CHROM_LEN; ++k) {
                if (flip((float)PMUT))
                    pool(i,j).string[k] = ~pool(i,j).string[k] & 0x01
            }
        }
    }
}
```

그림 3. 돌연변이 연산의 병렬화
Fig. 3. Parallelizing the mutation operation

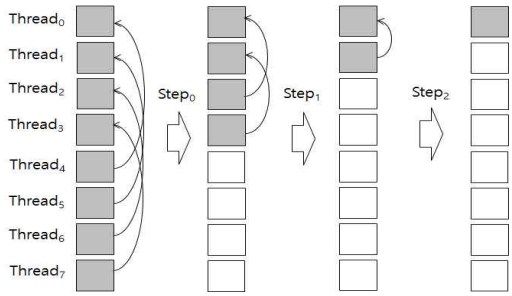


그림 4. 리덕션 방식
Fig. 4. Reduction method

이를 위하여 본 논문에서는 리덕션 방식[8]을 적용하였다. NVIDIA GPU는 캐시메모리의 성능을 가진 쓰레드 블록 공유 메모리를 제공한다. 따라서 최적의 성능을 위하여 계산하고자 하는 도메인을 공유 메모리에 복사한 후 계산을 수행한다. 그림 4는 8개의 쓰레드를 가진 블록에서 처리한 값을 3단계 리덕션으로 처리하는 것을 나타낸다. 각 단계 별로 (N/2#step-1)만큼 떨어진 쓰레드의 값을 사용하여 계산하고 log(N) 단계 이후에 최종적으로 Thread0에 원하는 결과를 저장한다. 각 쓰레드 블록의 계산 결과는 배열에 저장한 후에 CPU에서 최종적으로 계산한다.

그림 5는 리덕션 방식을 사용한 프로그램의 구현 사례다. (a)는 적합도 합과 최적해를 구하는 순차 프로그램이고 (b)는 3단계 리덕션을 수행하는 CUDA 프로그램이다: (i) 먼저 계산의 대상을 공유 메모리에 복사한 후에 (ii) 리덕션 방식의 계산을 진행하고 (iii) CPU에서의 계산을 위하여 해당 값을 해당 배열에 복사한다.

2.4 병렬 난수 프로그램

SSWCGAs은 난수를 주로 사용한다. 본 논문에서는 전체 도메인 단위로 임의의 수를 제공하는 난수 프로그램 방식[9]을 사용하였다. 이 병렬 난수 방식은 그림 6과 같이 최초의 도메인을 위한 난수는 순차적으로 계산하고 이후에는 도메인 단위로 난수를 병렬로 계산한다. 즉, 최초의 난수 도메인은 CPU 프로그램을 사용하여 순차적으로 계산하며 다음 도메인부터의 난수는 GPU에서 병렬로 계산하는 방식이다.

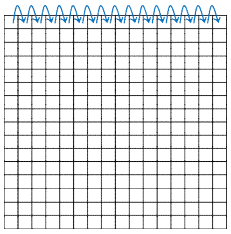
```
for (i=NB_DEPTH; i<P_SIZE_NB; i++) {
    for (j=NB_DEPTH; j<P_SIZE_NB; j++) {
        pool[i][j].fitness = ObjectValue(i,j);
        sum_fit += pool[i][j].fitness;
        if (pool[i][j].fitness > (*best_p).fitness) {
            for (k=0; k<CHROM_LEN; k++)
                (*best_p).string[k] = pool[i][j].string[k];
                (*best_p).fitness = pool[i][j].fitness;
        }
    }
}
```

(a) 순차 프로그램
(a) sequential program

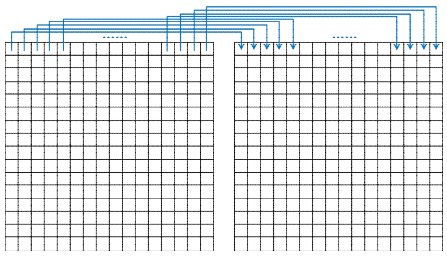
```
reduction(tid_x,tid_y).sum_fit = 0.0;
reduction(tid_x,tid_y).best_fit = 0.0;

// copy into shared memory
if ((NB_DEPTH<=i) && (i<P_SIZE_NB)) {
    if ((NB_DEPTH<=j) && (j<P_SIZE_NB)) {
        reduction(tid_x,tid_y).sum_fit = pool(i,j).fitness;
        reduction(tid_x,tid_y).best_fit = pool(i,j).fitness;
    }
}
// do reduction in shared memory
for(unsigned int s=blockdim/2; s>0; s>>=1) {
    if (idx2(tid_x,tid_y) < s) {
        reduction(tid_x,tid_y).sum_fit += reduction(tid_x,tid_y+s).sum_fit;
        if (reduction(tid_x,tid_y).best_fit < reduction(tid_x,tid_y+s).best_fit)
{
            reduction(tid_x,tid_y).best_fit =
reduction(tid_x,tid_y+s).best_fit;
            reduction(tid_x,tid_y).i = i;
            reduction(tid_x,tid_y).j = j;
        }
    }
    __syncthreads();
}
// write result for this block to global memory
if (idx2(tid_x,tid_y) == 0) {
    for (k=0; k<CHROM_LEN; k++)
        best(blockIdx.x,blockIdx.y).string[k] =
            pool(reduction(0,0).i,reduction(0,0).j).string[k];
    best(blockIdx.x,blockIdx.y).fitness = reduction(0,0).best_fit;
    sum_fit(blockIdx.x,blockIdx.y) = reduction(0,0).sum_fit;
}
```

(b) CUDA 병렬 프로그램
(b) CUDA parallel program

그림 5. 리덕션에 의한 적합도 합과 최적해 계산
Fig. 5. Fitness sum and best sln. with reduction method



(a) CPU 에서의 최초 난수 도메인의 순차 계산
(a) Sequential Calc. of Initial random # domain in CPU



(b) GPU에서의 병렬 난수 계산
(b) Parallel Calc. of random # in GPU

그림 6. 난수 도메인의 계산 순서
Fig. 6. Calculation sequence of data domain for random #

```
for (i=NB_DEPTH; i<P_SIZE_NB; i++){
    for (j=NB_DEPTH; j<P_SIZE_NB; j++) {
        for (k=0; k<CHROM_LEN; ++k) {
            f = ((float)rand())/RAND_MAX;
            if ((prob == 1.0) || (f < prob))
                pool[i][j].string[k] = ~pool[i][j].string[k] &
0x01;
}}}

if ((NB_DEPTH<=i) && (i<P_SIZE_NB)) {
        if ((NB_DEPTH<=j) && (j<P_SIZE_NB)) {
            for (k=0; k<CHROM_LEN; ++k) {
                f = ((float)randnum(i,j))/RAND_MAX;
                if ((prob == 1.0) || (f < prob))
                    pool[i][j].string[k] = ~pool[i][j].string[k] &
0x01;
}}}
```

그림 7. 리덕션에 의한 적합도
Fig. 7. Fitness by reduction method

그림 7은 난수를 계산하는 프로그램을 나타낸다. 순차 프로그램에서는 프로그램을 수행하면서 난수를 순차적으로 생성하지만 병렬 프로그램에서는 미리 쓰레드 블록 단위로 난수를 생성한 후, 병렬로 계산에 사용한다.

단순 유전 알고리즘을 병렬로 구현한 국내 외 사례는[10]-[13], 셀룰라 유전 알고리즘을 병렬로 구현한 사례는 [14]에 있다. 확률적 알고리즘을 병렬로 구현할 때는 난수를 어떻게 발생시키는지가 성능에 영향을 준다. 이 연구에서는 난수 발생의 효율을 높이기 위하여 난수 도메인을 그리드 상에서 병렬로 생성하여 도메인 계산을 병렬로 실행하도록 한 점에서 가존의 구현과 다르다. 또한 2.3절에 나타난 것과 같이 모집단을 분할하여 계산한 후, 개체의 전역 최대값을 계산하기 위하여 캐시메모리 역할을 하는 공유메모리를 사용하여 성능을 높였다.

III. 성능 평가 및 분석

이 장에서는 본 연구에서 병렬로 구현한 최소 좁은 세상 셀룰러 유전알고리즘의 정확도와 수행시간을 평가한다.

평가를 위해 다중 6차 바이폴라 디셉티브 문제(MD)를 사용한다. 6비트가 모두 0이거나 1일 때 적합도 값 1을 갖는 6차 바이폴라 디셉티브 함수로 구성된 이 문제는 골드버그, 뎁 및 혼 등이 제안한 것으로 GA-hard 문제에 속한다[15]. 디셉티브 함수는 6비트로 표현된 개체에 대하여 스키마 경쟁에서 승자가 열성이 되도록 고안한 것이다. 문제에 포함된 디셉티브 함수의 개수가 n일 때 탐색공간의 크기는 2^{6n}이고, $(\binom{6}{3}+2)^n$개의 지역 최적해를 갖는다. 그 중에서 2^n개만이 전역 최적해가 되므로 다루기가 매우 어려운 문제다. 예를 들어, 5개의 디셉티브 함수로 구성된 5중 6차 바이폴라 디셉티브 문제(MD5)는 개체의 크기가 30비트 스트링이며, 따라서 탐색공간의 크기는 2^{30}이고, 5,153,632개의 지역 최적해를 갖는다. 그 중에서 32개만이 전역 최적해이다. MD20은 20개의 디셉티브 함수를 부분 함수로 갖는 것으로 복잡도는 크게 증가한다.

3.1 최적해 성공률 평가 및 분석

본 논문은 SSWCGAs를 병렬로 구현하는 것이 주목적이므로 SSWCGAs 자체의 최적화 성능을 검증할 필요는 없다(SSWCGAs의 성능은 [3] 참조). 다만

병렬로 구현한 경우에도 최적해를 잘 찾는지를 확인할 필요가 있다. 여기서는 모집단의 크기를 달리해 가면서 100세대 내에 최적해에 도달하는 비율을 측정하여 병렬로 구현한 경우에도 최적해를 잘 찾는지를 평가한다. 이를 위해 MD20 문제에 대해 모집단의 크기를 달리해가면서 50회 실험하였으며, 교차율과 돌연변이율은 각각 0.85, 0.01로 하였다. 두 매개변수가 유전 알고리즘의 수행에 영향을 주지만 서로 다른 50개 모집단에 의한 평균 성능을 측정하므로 매개변수를 달리한 효과를 내포한다. SSWCGAs의 모집단은 논리적으로 격자형태이므로 모집단의 크기는 $n \times n$으로 나타낸다. 표 1은 실험 결과를 나타낸다. 실험에 사용된 GPU는 엔버디아 텔사 C2075(코어당 1.15Hz, 448개의 코어로 구성)이다.

표 1. 병렬 SSWCGAs의 최적화 성공률
Table 1. Success rate of parallel SSWCGAs

size of pop.	SSWCGAs		parallel SSWCGAs	
	success rate	avg generation	success rate	avg generation
40x40	0.78	30.6	0.32	25.0
50x50	0.96	29.5	0.92	59.8
60x60	0.96	28.5	0.92	41.8
70x70	0.98	28.5	0.92	36.5
80x80	1.0	28.2	0.98	33.5
90x90	1.0	27.2	0.96	6.15
100x100	1.0	26.6	1.0	3.4

단일 프로세스에서 구현한 경우 모집단의 크기가 80x80을 넘으면 언제나 최적해에 도달하며 그때까지의 평균 세대수는 30.6~26.6임을 알 수 있다. 반면에 병렬로 구현한 경우 성공률은 다소 떨어지지만 모집단의 크기가 80x80을 넘으면서 96%이상의 비율로 최적해를 찾았고, 평균 세대수는 59.8~3.4임을 알 수 있다. 따라서 동일한 크기의 모집단에 대해 동일한 비율로 최적해를 찾을 수 있으므로 구현은 잘 되었다고 할 수 있다. 평균 세대수 편차에 대해서는 결론에서 다시 언급한다.

3.2 수행시간 분석

병렬로 구현한 SSWCGAs의 수행시간을 평가하기 위해 3.1과 같은 조건에서 모집단의 크기를 달리해 가면서 100세대 수행시간의 50회 평균을 계산하였다. 실험 결과는 표 2와 같다. 실험에 사용된 쓰레드는 256(=16×16)개로서 이는 리덕션 연산을 수행하기 위한 최대의 개수이다. 비교를 위하여 사용된 CPU는 인텔 i7-4770(3.4GHz)이다.

표 2. 100세대 평균 수행시간(50회) (밀리초)
Table 2. Average time for 100 generations(50 iterations, ms)

size of pop.	40	50	60	70	80	90	100
single	1362.1	2123.4	3057.6	4171.6	5446.9	6898.9	8553.4
parallel	185.7	254.5	328.3	439.5	543.6	704.1	800.0
rate	14%	12%	11%	11%	10%	10%	9%

*모집단 크기 n은 $n \times n$을 줄여서 나타낸 것이다.

병렬로 구현하는 경우 100세대 수행시간 평균이 14%~9% 수준임을 알 수 있다. 그림 8은 수행시간 변화 추이를 나타낸다.

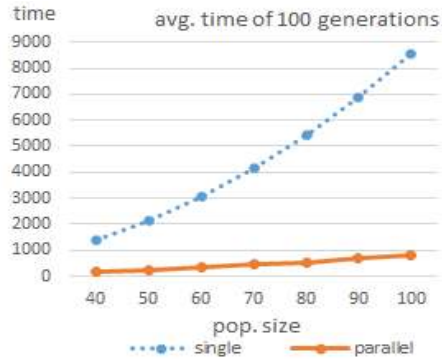


그림 8. 100세대 평균 수행 시간
Fig. 8. Average time for 100 generations

다음은 같은 조건에서 모집단의 크기를 달리해 가면서 최적해에 도달하는 평균 수행시간을 계산하였다. 실험 결과는 표 3과 같다.

표 3. 최적해에 도달하는 평균 수행시간(50회) (밀리초)
Table 3. Average time to optimum(50 iterations, ms)

size of pop.	40	50	60	70	80	90	100
single	416.8	626.4	870.2	1188.5	1533.8	1876.5	2275.2
parallel	47.5	152.6	139.8	160.4	182.1	43.3	27.2
rate	11%	24%	16%	13%	12%	2%	1%

*모집단 크기 n은 $n \times n$을 줄여서 나타낸 것이다.

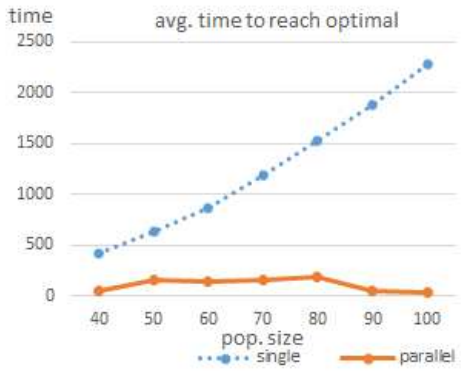


그림 9. 최적해에 도달하는 평균 수행 시간
Fig. 9. Average run time for the best solution)

그림 9는 최적해에 도달하는 수행시간 변화 추이를 나타낸다. 모집단의 크기가 50x50인 경우를 제외하면 대략 10% 수준임을 알 수 있다. 이 경우 수행시간 비율의 편차가 큰 것은 앞과는 다르게 100세대 전에 최적해를 찾는 세대수에 편차가 크기 때문이다. 따라서 병렬로 구현하는 경우가 단일 프로세서로 구현하는 것보다 상대적으로 불안정하다고 할 수 있다.

Ⅳ. 결 론

최소 좁은 세상 셀룰러 유전알고리즘은 격자 즉 그래프 형태의 모집단을 좁은 세상 네트워크로 구성하여 높은 클러스터링 계수를 유지하면서 짧은 평균 경로 길이를 갖는 위상을 갖도록 한 것이다 [16]. 이렇게 하면 세부탐색 능력은 보통의 셀룰러 유전 알고리즘과 같으면서 상대적으로 짧은 평균경로 길이 때문에 빠르게 전역 탐색이 이루어진다. 이 연구에서는 이를 CUDA를 사용하여 병렬화 하였다.

성능을 평가하기 위해 20중 6차 바이폴라 디셉티브 문제 MD20을 사용하였다. 이 문제의 탐색공간 크기는 $2^{20\times 6}$으로 지역 최적해가 많은 매우 어려운 문제다.

평가 결과 동일한 크기의 모집단에 대해 동일한 비율로 최적해를 찾는 것을 확인 할 수 있었다. 또한 수행시간 측면에서도 대략 10% 내외의 시간에 최적해를 찾을 수 있음을 확인하였다.

모집단이 충분히 큰 90x90 이상의 경우 매우 빠르게(적은 세대수 만에) 최적해를 찾는데, 이는 지역 최적해가 지름길을 통해 중앙으로 이동하여 지역적으로 경쟁하는데 영향을 주기 때문으로 판단된다. 단일 프로세서에서 구현하는 경우 먼저 모든 후보해들이 지역적으로 경쟁한 후에 지름길을 통한 유전자 교환이 일어나는 반면, 병렬로 구현하는 경우 쓰레드 블록이 완료되는 시점에 유전자 교환이 일어나고 이것이 다른 블록 처리시 교환된다. 따라서 더 빠르게 좋은 후보해가 모집단에 퍼지게 된다. 이것은 병렬로 구현하는 경우 조기 수렴이라는 부작용이 발생할 수 있음을 의미한다. 실제로 병렬 SSWCGAs가 최적해를 찾는 비율이 2~4% 정도 낮은 것, 최적해에 도달하는 평균 수행시간의 편차가 큰 것과 관계될 것으로 판단되며, 추후 이에 대한 분석과 대응 방안에 대한 연구가 따라야 할 것이다.

References

[1] Reka Albert and Albert-Laszlo Barabasi, "Statistical mechanics of complex networks", Rev. Mod. Phys., Vol. 74, No. 1, pp. 47-97, Jan. 2002.

[2] Ted G. Lewis, "Network Science Theory and Practice", Wiley, Apr. 2009.

[3] Taewon Kang, "Smallest-Small-World Cellular Genetic Algorithms", Journal of KIISE B: Vol. 34, No. 11, pp. 971-983, Nov. 2007.

[4] D. Watts and S. H. Strogatz, "Collective dynamics of small world networks", Nature(London), Vol. 393, No. 6684, pp. 440-442, Jun. 1998.

[5] D. Watts, "Small world: the dynamics of networks between order and randomness", Princeton, 1999.

[6] T. Nishikawa, A. E. Motter, Y. C. Lai, and F. C. Hoppensteadt, "Smallest small-world networks", Physical Review E, Vol. 66, No. 4, 046139, Oct. 2002.

[7] D. Whitley, "Cellular Genetic Algorithms", Proc. 5th International Conference on Genetic Algorithms, pp. 658, 1993.

[8] NVIDIA, CUDA, Compute Unified Device Architecture Reference Manual Version 5.0, NVIDIA Corporation, 2012.

[9] Youngtae Kim and Gyuhyun Hwang, "Efficient Parallel CUDA Random Number Generator on NVIDIA GPUs", Journal of KIISE, Vol. 42, No. 12, pp. 1467-1473, Dec. 2015.

[10] Stefano Debattisti, Nicola Marlat, Luca Mussi, Stefano Cagnoni, Implementation of a Simple Genetic Algorithm within the CUDA Architecture, http://www.gpgpgpu.com/gecco2009/3.pdf.

[11] Il Jun Ahn and In Kyu Park, "Design of Omok AI using Genetic Algorithm and Game Trees and Their Parallel Processing on the GPU", Journal of KIISE Vol. 37, No. 2, pp. 66-75, Apr. 2010.

[12] Hyunsoo Park and Kyungjoong Kim, "Systematic Evaluation of Island based Real-Valued Genetic Algorithm with Graphics Processing Unit", Proc. of Korea Computer Congress, Vol. 37, No. 1(C), pp. 328-333, Jul. 2010.

[13] Byeongyong Hyeon, Ohsung Gwon, Soohwan Hyun, and Kisung Seo, "GPU Implementation of Genetic Algorithm", Proc. of Korean Institute of Intelligent Systems, Vol. 20, No. 1, pp. 3-6, Apr. 2010.

[14] Pablo Vidal and Enrique Alba, "A Multi-GPU Implementation of a Cellular Genetic Algorithm, Evolutionary Computation (CEC)", 2010 IEEE Congress on Date of Conference: pp. 1-7, Jul. 2010.

[15] D. E. Goldberg, K. Deb, J. Horn, Massive Multimodality, Deception, and Genetic Algorithms, IlliGAL Report No. 92005, Apr. 1992.

[16] TaeWon Kang, "Fault Tolerance Analysis of the Neural Networks of Scale-Free Network Architecture", Journal of KIIT, Vol. 11, No. 7, Jul. 31, 2013.

저자소개

김 영 태 (Youngtae Kim)

1986년 : 연세대학교 수학과 (이학사)
1992년 : 미국 Iowa State Univ. (MS.)
1996년 : 미국 Iowa State Univ. (Ph. D.).
1998년 ~ 현재 : 강릉원주대학교 컴퓨터공학과 교수
관심분야 : 초고속 컴퓨팅, 컴퓨터 성능 분석

강 태 원 (Taewon Kang)

1985년 : 연세대학교 수학과 (이학사)
1988년 : 고려대학교 전산과학과 (이학사)
1991년 : 고려대학교 수학과 (이학석사)
1996년 : 고려대학교 컴퓨터학과 (이학박사)
1997년 ~ 현재 : 강릉원주대학교 컴퓨터공학과 교수
관심분야 : 복잡계, 인공생명, 인공지능, 소프트컴퓨팅