GPU를 이용한 무리 짓기에서 이웃 에이전트 찾기의 병렬 처리※

이재문
한성대학교 멀티미디어공학과
jmlee@hansung.ac.kr

A Parallel Processing of Finding Neighbor Agents in Flocking Behaviors Using GPU

Jae Moon Lee
Dept. of Multimedia Engineering, Hansung University

요 약

논문은 GPU를 이용한 무리 짓기에 대한 병렬 알고리즘을 제안한다. 이를 위하여 GPU의 병렬처리 구조로 CUDA를 사용하였으며, 그것의 특성 및 제한 요소들을 분석하였다. 이의 특성 및 제한 요소를 기초로 무리 짓기에서 가장 많은 비용을 요구하는 이웃 에이전트들을 찾는 것을 병렬화 함으로써 성능을 개선하였다. 제안된 알고리즘을 GTX 285상에서 구현하였고, 그것의 성능을 실험적으로 기존의 공간분할 알고리즘과 비교하였다. 비교의 결과는 제안된 알고리즘이 실행 시간 관점에서 최대 9배 정도 우수하다는 것을 보였다.

ABSTRACT

This paper proposes a parallel algorithm of the flocking behaviors using GPU. To do this, we used CUDA as the parallel processing architecture of GPU and then analyzed its characteristics and constraints. Based on them, the paper improved the performance by parallelizing to find the neighbors for an agent which requires the largest cost in the flocking behaviors. We implemented the proposed algorithm on GTX 285 GPU and compared experimentally its performance with the original spatial partitioning method. The results of the comparison showed that the proposed algorithm outperformed the original method up to 9 times with respect to the execution time.

Keywords : GPU, CUDA, Parallel Processing, Flocking Behavior(GPU, 쿠다, 병렬처리, 무리 짓기)

접수일자 : 2010년 08월 30일 심사완료 : 2010년 09월 24일
※ 본 연구는 2009년도 한성대학교 연구년 지원과제 임.

1. 서 론

최근 GPU(Graphic Process Unit)의 컴퓨팅 파워를 범용 계산에 활용하는 GPGPU(General Purpose computations on GUP)에 대한 연구가 수학, 과학 분야에 급속히 확산되어 가고 있다. 특히, GPU는 작은 프로세서들의 큰 배열로 구성되어 있기 때문에 단순한 병렬 연산에서 좋은 성능을 보여 준다[3,4,5,6,7,8,9,10,11]. 특히, 행렬 연산, 데이터 정렬 등 기본 연산에서 이미 CPU보다 고속처리가 가능한 것으로 알려져 있으며, 바이오 정보처리 등의 응용에서도 많은 연구가 진행되고 있다[3,4,5,9,10].

GPGPU 기술은 이미 오래전부터 개발되어 왔지만 최근 PC에 본격적으로 적용되고 있다. NVIDIA 사는 C언어 기반 개발 도구인 CUDA(Compute Unified Device Architecture)를 발표 하였으며[7], AMD 사도 ATI Stream을 개발하여 발표 하였다. 인텔 사의 경우 x86 기반 다수의 코어로 구성된 라라비(Larrabee)를 준비 중이다. Microsoft 사의 DirectCompute는 Windows VISTA 및 Windows 7 환경에서 NVIDIA 사의 최신 CUDA 아키텍처를 실행할 수 있도록 해주는 새로운 GPU 컴퓨팅 API 제공한다[7]. 또한 GPU를 이용한 다양한 가속화 연구가 진행되고 있는데 [6,8]에서는 GPU의 고유의 기능인 효율적인 렌더링에 대한 연구를 하였고, [3,4,5,9,10,11]에서는 GPU를 사용하여 대용량 데이터를 처리하는 연구를 하였다.

무리 짓기는 다수의 에이전트들이 자율적으로 움직일 때 나타나는 행동을 말한다[1,2,11,12]. 이러한 집단행동은 영화나 게임 등에서 장면의 사실성을 증대시키거나 현실적으로 제작하기 어려운 영상을 시뮬레이션 하는데 사용된다. 무리 짓기는 무리에 속한 에이전트들이 상호 작용을 하면서 집단행동을 하고, 실시간으로 처리되어야 하기 때문에 대부분의 경우 에이전트의 수를 제한한다. 본 논문은 보다 많은 에이전트에 대하여 실시간으로 시뮬레이션이 가능하도록 CUDA를 이용하여 무리 짓기를 병렬처리 하여 성능을 개선하는 것이다.

2장에서는 CUDA의 구조 및 무리 짓기 알고리즘에 대하여 소개하며, 3장에서는 CUDA를 사용한 병렬 무리 짓기 알고리즘을 소개한다. 4장에서는 기존의 CPU 기반 알고리즘과 성능 비교를 하며, 5장에서 결론을 논한다.

2. 관련 연구

2.1 CUDA 소개

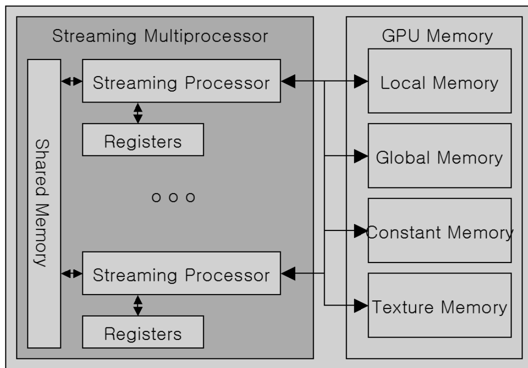
2.1.1 CUDA 지원 GPU 구조

GPGPU를 위한 개발 환경 및 아키텍처로 대표적인 것이 NVIDIA 사의 CUDA이다[7]. CUDA는 NVIDIA 사의 GPU를 위한 통합된 병렬 데이터 연산을 제공하는 소프트웨어 개발 도구이다. CUDA의 가장 큰 장점은 개발자가 보다 쉽고 직관적으로 GPU를 프로그래밍 할 수 있도록 확장된 형태의 C언어를 지원한다는 것이다. 따라서 C언어에 익숙한 개발자라면 GPGPU 프로그래밍을 위해 별도의 병렬처리 프로그래밍 언어 등을 익힐 필요가 없다.

CUDA는 소수의 고성능 프로세서로 구성된 구조가 아니라 다수의 저성능 프로세서의 배열로 구성된 구조이다. 따라서 복잡한 구조의 프로그램에 대한 병렬처리 보다는 단순한 연산의 반복적인 처리에 적합한 모델이다. 예를 들어 행렬 연산이 대표적인 예이다. CUDA에서 프로그램 실행 단위는 쓰레드(Thread)이고, 다수의 쓰레드를 효율적으로 관리하고, 실행하기 위하여 블록(Block), 그리드(Grid)의 기능을 제공한다. 블록은 동일한 프로그램을 동시에 실행하는 쓰레드의 집합이고, 그리드는 이러한 블록을 논리적으로 1차원, 2차원 및 3차원으로 분류시키는 기능이다. 특히, 블록내의 쓰레드들은 빠른 속도로 통신이 가능하도록 다양한 제한 요소들을 갖는다.

CUDA에서 쓰레드가 빠른 통신을 보장하기 위하여 하나의 블록에 속한 모든 쓰레드는 하나의 스트리

밍 멀티프로세서에 의하여 실행된다. 따라서 CUDA에서 효율적인 병렬처리를 하기 위해서는 사용 가능한 스트리밍 멀티프로세서 수와 블록의 수를 적절히 조절하여야 한다. [그림 1]은 CUDA에서 하나의 스트리밍 멀티프로세서에 대한 스트리밍 프로세서와 다양한 메모리의 관계를 나타내고 있다. 하나의 스트리밍 멀티프로세서는 8개의 스트리밍 프로세서로 구성하고 이들은 16K바이트의 메모리를 공유한다. 각 스트리밍 프로세서는 독립적으로 사용하는 레지스터를 가지고 있으며, 공유 메모리외 GPU 메모리(지역, 전역, 상수 및 텍스처 메모리)를 액세스 할수 있다. 공유 메모리는 스트리밍 멀티프로세서 내부에 존재하여 액세스 속도가 매우 빠르나 용량이 제한적이고, GPU 메모리는 스트리밍 멀티프로세서 외부에 있어 액세스 속도는 느리나 용량은 매우 크다. 공유 메모리의 경우 대부분 16K바이트이나 GPU 메모리는 256M바이트에서 1G바이트로 대용량이다. 효율적인 CUDA 프로그래밍을 위해서는 이러한 메모리의 적절한 사용도 매우 중요하다.



[그림 1] CUDA에서 멀티프로세서 및 메모리

CUDA는 쓰레드의 효율적인 관리와 공유 메모리 사용의 극대화를 위하여 다음과 같은 2가지 원칙에 의하여 쓰레드를 생성, 관리한다.

- 하나의 블록에 할당된 모든 쓰레드는 하나의 스트리밍 멀티프로세서에 의하여 실행된다. 또한 하나의 블록에 할당되는 쓰레드 수는 최대 512개 이다.
- 하나의 스트리밍 멀티프로세서는 *warp*이라는 단위로 쓰레드를 생성, 관리 및 실행한다. 하나의 *warp*는 32개의 쓰레드이다.

첫 번째 원칙은 공유 메모리의 사용을 극대화하기 위함이다. 공유 메모리는 해당 멀티프로세서 내에 존재하는 스트리밍 프로세서들만 이 메모리를 액세스할 수 있다. 따라서 서로 다른 블록에 속한 쓰레드간 통신은 외부 메모리인 전역 메모리를 사용하여야 하는데, 이 경우 속도가 현저히 저하된다. 두 번째 원칙은 스레드의 생성, 실행 관리의 오버헤드를 최소화하기 위함이다. 각 스트리밍 프로세서는 4개의 쓰레드를 담당하고 이는 명령어의 최소 단위에 대하여 순차적으로 4번을 반복 실행하여 4개의 쓰레드 효과를 나타낸다. 따라서 CUDA를 이용한 병렬처리를 극대화하기 위해서는 커널 프로그램에 따라 적절한 블록당 쓰레드 수와 블록의 개수를 제어하여야 한다.

2.1.2 CUDA 프로그래밍

CUDA의 병렬 프로그램은 크게 호스트 프로그램과 GPU 프로그램으로 나뉜다. CUDA SDK에 의하여 작성된 프로그램은 가장 먼저 메인 메모리에 저장된 데이터를 GPU의 전역 메모리로 전송한다. 이렇게 전송된 데이터에 대하여 적절한 함수를 호출하는 것이 두 번째 순서이고, GPU는 호출된 함수를 스트리밍 프로세서를 사용하여 이 함수를 병렬적으로 실행하는 것이 세 번째 순서이다. 마지막으로 처리된 데이터를 다시 메인 메모리로 전송하는 것이다.

[그림 2]는 CUDA 프로그램의 가장 단순한 예이다. 여기서 '__global__'의 의미는 이 함수는 호스트에서 호출할 수 있는 GPU 함수를 의미하고, '__device__'는 GPU내에서만 호출될 수 있는 GPU 함수를 의미한다. [그림 2]에서 '// (번호)'는 CUDA 프로그램이 일반적으로 실행되는 순서를

나타낸다. 먼저 함수 *cudaMalloc*와 *cudaFree*는 GPU내의 전역 메모리를 할당받고, 해제하는 함수이다. 이의 사용은 동적 메모리 할당 함수인 *malloc*, *free*와 매우 유사하다. *cudaMemcpy* 함수는 호스트 데이터를 GPU에 보낼 수도 있고, GPU 데이터를 호스트로 보낼 수 있는 API인데 이의 방향은 마지막 인자를 이용하여 제어한다.

```
__device__ int dfunc(...){
   ...
}
__global__ void kfunc(...){                    // (3)
   ...
   dfunc(...);
   ...
}
int main(){
   ...
   cudaMalloc(&gMem, ...);
   cudaMemcpy(gMem, hMem, size, 1);            // (1)
   kfunc<<<blocks, threads>>>(...);            // (2)
   cudaMemcpy(hMem, gMem, size, 0);            // (4)
   cudaFree(gMem);
   ...
}
```

[그림 2] CUDA 프로그램의 예

[그림 2]에서 가장 중요한 것은 (2)번과 (3)번이다. (2)번은 *kfunc*이라는 함수를 병렬적으로 처리하라는 명령을 호스트가 GPU에게 하달하는 것이고 (3)은 이 명령을 처리하는 함수이다. 특히 (2)에서 *blocks*와 *threads*가 나오는데 이는 GPU가 병렬처리를 실행할 때 쓰레드 운영 방식을 제어하는 것이다. [그림 2]에서 (2)의 의미는 *threads*개의 쓰레드를 사용하여 *kfunc*라는 함수를 동시에 실행하되 이를 *blocks*만큼 반복하라는 의미이다. CUDA의 병렬처리 프로그램에서 이들의 선택은 중요하다. 이것은 이들의 올바른 선택만이 좋은 성능 개선을 가져올 수 있기 때문이다. 이들의 값은 *kfunc*의 프로그램 내용과 이 프로그램이 실행되는 컴퓨터의 GPU가 가지는 스트리밍 멀티프로세스 수에 의존적이다. 예를 들어 Geforce 9600M GT는 4개의 스트리밍 멀티프로세서로 구성되어 있어 32개의 스트리밍 프로세서가 있다. '*kfunc*<<<128, 8>>>(...)'의 경우 4개의 스트리밍 멀티프로세서에 의하여 병렬적으로 실행되고 따라서 *kfunc*는 각 스트리밍 멀티프로세서에 의하여 32(=128/4)번 반복 실행된다. 그러나 하나의 스트리밍 멀티프로세서는 32개의 쓰레드를 생성하여 그 중 8개의 쓰레드만 사용하면서 *kfunc*의 8호출을 병렬적으로 실행하게 된다. 반면 '*kfunc*<<<32, 32>>>(...)'로 호출하는 경우 *kfunc*는 각 스트리밍 멀티프로세서에 의하여 8(=32/4)번 반복 실행되고, 하나의 스트리밍 멀티프로세서는 32개의 쓰레드를 사용하여 32 호출을 동시에 실행하게 된다. *kfunc*의 프로그램 내용에 따라서 '*kfunc*<<<128, 8>>>(...)'은 가능하나 '*kfunc*<<<32, 32>>>(...)'은 불가능할 수도 있다. 그러나 만약 둘 다 가능하다면 후자의 방법이 훨씬 효율적인 실행이 될 것이다.

2.2 무리 짓기 알고리즘

무리 짓기란 무리를 지어 날아다니는 새들과 같이 특별한 중앙 제어도 없이 가까운 에이전트들의 영향력을 받아 상호작용을 하면서 움직이는 현상이다. [1,2,12]에서는 이웃 에이전트들의 영향력을 분리힘, 정렬힘 및 결합힘으로 구성하였다. 분리힘은 이웃 에이전트간 충돌을 피하는 힘이며, 정렬힘은 이웃 에이전트들과 같은 방향으로 움직이려는 힘이며, 마지막으로 결합힘은 이웃 에이전트들과 너무 멀리 떨어지지 않으려는 힘이다. 이러한 힘들의 계산에서 가장 중요한 것이 영향을 미치는 이웃 에이전트를 탐색하는 것이다[2,12]. 따라서 무리 짓기 알고리즘 성능은 에이전트들의 수에 많은 영향을 받는다.

```
void CPUFlocking(Agents *agents[], int n, int k)
01:  for(int i=0;i<n;i++){
02:      for(int j=0;j<n;j++){
03:          distance[j]=GetDistance(agents[i], agents[j]);
04:      }
05:      kNN= FindKnn(agents, distance, n, k)
06:      agents[i]->force=GetForce(agents[i], kNN);
07:  }
     // 각 에이전트에 대한 새로운 위치 계산
```

[그림 3] 기본적인 무리 짓기 알고리즘

[그림 3]은 무리 짓기에 대한 기본적인 알고리즘이다. 이 알고리즘의 입력은 에이전트의 집합인 *agents*, 에이전트의 수 n 및 영향을 받는 이웃 에이전트의 수를 제한하는 상수 k이다. 이는 공간상에 존재하는 모든 에이전트들에 대하여 각각 k개의 가장 가까운 이웃 에이전트들을 찾고 이들로부터 정렬힘, 결합힘 및 분리힘을 계산하여 새로운 방향과 위치를 결정하는 것이다.

[그림 3]에서 *GetDistance*는 두 에이전트간 거리를 계산하는 함수이며, *FindKnn*은 거리가 가장 가까운 k개의 에이전트를 찾아 리턴하는 함수이다. *GetForce*은 하나의 에이전트에 대하여 찾아진 이웃 에이전트들을 이용하여 이 에이전트에 미치는 분리힘, 정렬힘 및 결합힘을 계산하는 함수이다. 이 알고리즘은 $O(n^2)$의 시간 복잡도를 갖는다. 이러한 비용은 매 프레임마다 계산되어야 한다. 즉, 게임이 최소한 초당 30프레임의 화면을 보인다면 초당 30번씩 $O(n^2)$ 만큼 계산되어야 한다. 따라서 대규모 무리 짓기와 같이 n이 큰 경우 이 비용은 매우 크다. 이러한 비용을 줄이기 위하여 [2,12]에서는 공간분할 방법을 사용해 왔다. 이 방법은 매 프레임마다 에이전트들을 분할된 공간에 매핑하므로써 이웃 에이전트들을 효율적으로 탐색하도록 한다. [2,12]에 의하면 이 비용은 $O(kn)$이 된다. 대부분의 경우 k가 n에 비하여 매우 작으므로 이 방법은 무리 짓기의 성능을 크게 향상시킨다.

3. 무리 짓기의 병렬처리

본 논문에서는 CUDA를 이용하여 무리 짓기의 성능을 개선하는 것이다. 즉, [그림 3]의 알고리즘을 CUDA의 환경에 적합하도록 병렬화하여 성능을 개선하는 것이다. [그림 4]와 [그림 5]는 [그림 3]에 대한 병렬 알고리즘이다. [그림 4]는 호스트 프로그램으로 GPU에 필요한 데이터를 전달하고, 병렬처리된 데이터를 GPU로부터 가져와서 조종힘을 계산하는 프로그램이고, [그림 5]는 다수의 GPU 쓰레드에 의하여 동시에 실행되는 GPU 프로그램이다.

[그림 4]의 라인 3에서 *GetLocation*은 에이전트들의 위치 정보만 추출하여 *hLocs*에 저장하는 함수이다. 이것은 GPU에서의 연산에서는 오직 에이전트의 위치 정보만 필요하기 때문에 GPU에 전달하는 데이터를 최소화하기 위함이다. 라인 6은 GPU에서 계산된 데이터를 CPU로 가져오는 것인데, 이때 데이터는 각 에이전트별 가장 가까운 k개의 이웃 에이전트들에 대한 인덱스를 가져온다. 즉, *hKNNs*은 $k \times n$개의 정수 배열로 구성되는데 각 에이전트에 대하여 가장 가까운 k개의 에이전트들의 인덱스가 순차적으로 저장되어 있다. 라인 9에서 *GetKnn*은 에이전트 i에 대한 k개의 가장 가까운 에이전트를 *hKNNs*로부터 찾아오는 함수이다. 라인 5에서 *GetNeighbors*는 GPU 함수이다. 즉, 호스트 프로그램에서 호출하는 GPU 프로그램이다. 이 함수는 *GetNeighbors<<<n, n/2>>*이므로 $n/2$개의 쓰레드를 사용하여 병렬적으로 하나의 에이전트에 대한 k개의 가장 가까운 이웃 에이전트들을 찾는다. 따라서 이것은 $\frac{n}{\text{멀티프로세서수}}$만큼 반복적으로 실행될 것이다.

```
void GPUFlocking(Agents *agents[], int n, int k)
01: int    *hKNNs, *dKNNs;
02: Vec3 *hLocs, *dLocs;;
    /* 여기서 malloc과 cudaMalloc를 이용하여
       각각 hLocs, hKNNs와 dLocs, dKNNs를 할당
       받았다고 가정한다.
    */
03: GetLocation(hLocs, agents, n);
04: cudaMemcpy(dLocs, hLocs, n*sizeof(Vec3), 1);
05: GetNeighbors<<<n, n/2>>>(dLocs, n, k, kNNs);
06: cudaMemcpy(hKNNs, dKNNs, n*k*sizeof(int), 1);
07: for(int i=0;i<n;i++){
08:     kNN= GetKnn(i, hKNNs, agents);
09:     agents[i]->force= GetForce(agents[i], kNN);
10: }
    // hKNNs, dKNNs, hLocs, dLocs를 해제한다.
    // 각 에이전트에 대한 새로운 위치 계산
```

[그림 4] 무리 짓기의 호스트 병렬 알고리즘

```
__global__
void GetNeighbors(Vec3* dLocs, int n, int k, int *neighbors)
01: int target= blockIdx.x, agentNo=threadIdx.x;
02: __shared__ Data_t data[MAXAGENTS];
    /* 에이전트간 거리 계산후 id와 거리 저장,
       n/2개의 쓰레드에 의하여 병렬처리됨
    */
03: data[agentNo].dist=|dLocs[target]-dLocs[agentNo]|
04: data[agentNo].agentNo=agentNo;
05: agentNo += n/2;
06: data[agentNo].dist=|dLocs[target]-dLocs[agentNo]|
07: data[agentNo].agentNo=agentNo;

    // 거리의 가까움을 기준으로 정렬
08: BitonicSort(data);

    // 가장 가까운 k개의 에이전트만 저장
09: if(threadIdx.x < k)
10:     neighbors[target*k+agentNo]=
                          data[agentNo].agentNo;
```

[그림 5] 무리 짓기의 GPU 병렬 알고리즘

[그림 5]는 하나의 블록에 의하여 병렬적으로 처리되는 GPU 프로그램이다. [그림 4]에서 하나의 블록이 n/2개의 쓰레드로 구성되도록 하였으므로 이 함수는 n/2개의 쓰레드에 의하여 동시에 실행된다. 따라서 라인 1에서 *target*은 이웃 에이전트를 찾는 에이전트가 되어 0~n-1중의 하나가 되며, *agentNo*는 0~n/2-1중의 하나의 번호 즉, 쓰레드 번호가 된다. 라인 2의 '__shared__'라는 키워드는 뒤의 변수 *data*를 공유 메모리에 할당하라는 의미이다. 각 쓰레드는 n개의 에이전트에 대하여 거리를 계산해야 하므로 하나의 쓰레드는 두 개의 에이전트에 대하여 거리를 계산한다. 예를 들어 0번 쓰레드는 0번, n/2 에이전트와 1번 쓰레드는 1번, n/2+1 에이전트와의 거리를 계산하여 공유 메모리에 저장한다. 라인 3~7사이가 이것을 하는 코드이다. 이렇게 저장된 데이터는 병렬 정렬 방법 중의 하나인 바이토닉 병렬 정렬[10]을 이용하여 정렬한다. 마지막으로 정렬된 데이터로부터 k개만 선택하여 전역 메모리인 *neighbors*에 저장한다. 이것도 k개의 쓰레드를 사용하여 병렬로 처리된다. 이 함수의 시간 복잡도는 라인 3~7의 비용이 $O(2)$인 반면, 라인 8의 *BitonicSort*이 $O((\log_2 n)^2)$이므로 전체적으로 $O((\log_2 n)^2)$이 된다.

4. 성능비교

성능을 비교하기 위하여 기존의 잘 알려진 공간 분할 방법과 제안된 방법을 윈도우즈상에서 구현하였다. 알고리즘의 구현은 비주얼 스튜디오 9를 사용하였고 컴퓨터 언어는 C/C++를 사용하였다. 실험을 위하여 인텔 코어2쿼드 Q8200 2.33Ghz의 CPU와 2GB 메모리, NVIDIA 사의 Geforce GTX 285 GPU로 구성된 컴퓨터를 사용하였다. 이 GPU는 30개의 스트리밍 멀티프로세서로 구성되어 있어 전체 스트리밍 프로세서는 240개로 구성된다. 또한 전역 메모리는 1G바이트이며, 각 멀티프로세서가 소유하는 공유 메모리는 일반 NVIDIA 사의 GPU와 동일하게 16K바이트이다. 또한 CUDA는 CUDA SDK 2.3를 사용하였다.

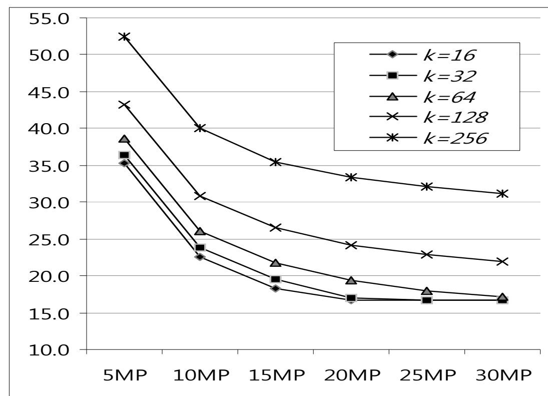
무리 짓기에 참여하는 에이전트의 수는 128, 256, 512 및 1,024개로 하였으며, 이웃 에이전트의 수 k는 각 에이전트 수 n에 대하여 5%, 10%, 20%, 30%, 40%, 50%로 변화하면서 기존의 공간 분할 방법과 제안된 방법의 실행 시간을 측정하여 비교하였다. 실행 시간은 무리 짓기에 대하여 1,000프레임을 연속적으로 시뮬레이션할 때 소요된 시간을 초단위로 측정한 것이다. [표 1]은 성능 비교 결과이다. [표 1]에서 T_{CPU}는 기존의 공간분할 방법의 실행 시간을 표시하며, T_{GPU}는 제안된 방법의 실행 시간을 표시한다. 비율은 단순히 $\frac{T_{CPU}}{T_{GPU}}$를 계산한 값이다. 실험 결과 나타난 현상은 크게 두 가지로 평가할 수 있다. 첫 번째는 에이전트 수가 증가함에 따라 성능 개선 효과가 크다는 것이다. n이 128의 경우 평균 4.9배의 개선 효과가 있으나, n이 1,024인 경우 평균 7.4배의 개선 효과가 있음을 알 수 있다. 이것은 대규모 에이전트가 참여하는 무리 짓기의 성능 개선을 위하여 GPU를

사용한 병렬처리가 효과적임을 의미한다. 두 번째는 이웃 에이전트의 수가 증가함에 따라 성능 개선 효과가 크다는 것이다. 이는 [그림 5]에서 가장 큰 비용을 차지하는 *BitonicSort*가 k의 값에 관계없이 일정한 반면, 공간 분할 방법은 k에 크기에 따라 그 성능이 크게 의존하기 때문이다. [표 1]로부터 제안된 방법이 최대 9배의 성능 개선 효과가 있음을 알 수 있다. 30개의 스트리밍 멀티프로세서를 가진 GTX 285 GPU 사용하고도 최대 9배의 성능 개선 효과밖에 얻지 못함은 스트리밍 멀티프로세서의 성능이 인텔 코어2쿼드 Q8200 CPU보다 현저히 낮고, 또한 다수의 쓰레드가 공유 메모리 및 전역 메모리를 액세스할 때 일어나는 버스 충돌이 영향을 주며, [그림 4]에서 라인 5를 제외한 다른 부분의 비 병렬처리 비용도 적지 않기 때문이다.

[표 1] 무리 짓기 성능비교

k	방법	n = 128	256	512	1024
$5\% \times n$	T_{CPU}(초)	0.9	2.7	10.2	56.2
	T_{GPU}(초)	0.5	1.1	3.1	10.1
	비율	1.7	2.4	3.2	5.6
$10\% \times n$	T_{CPU}(초)	1.7	5.2	19.4	80.8
	T_{GPU}(초)	0.7	1.4	4.2	14.2
	비율	2.4	3.8	4.7	5.7
$20\% \times n$	T_{CPU}(초)	3.0	10.4	37.5	155.7
	T_{GPU}(초)	0.8	1.9	6.3	20.6
	비율	3.8	5.5	6.0	7.6
$30\% \times n$	T_{CPU}(초)	5.1	16.3	57.3	228.7
	T_{GPU}(초)	0.9	2.4	7.7	27.9
	비율	5.5	6.8	7.5	8.2
$40\% \times n$	T_{CPU}(초)	7.3	22.9	77.3	301.8
	T_{GPU}(초)	1.0	2.9	9.5	35.5
	비율	7.4	7.8	8.1	8.5
$50\% \times n$	T_{CPU}(초)	10.0	30.7	99.6	375.6
	T_{GPU}(초)	1.1	3.5	11.1	42.8
	비율	8.8	8.8	8.9	8.8

CUDA 프로그래밍은 적절한 반복문을 사용하여 병렬처리에 사용되는 스트리밍 멀티프로세서의 수를 제한할 수 있다. [그림 6]은 GTX 285상에서 사용된 멀티프로세서의 수를 5, 10, ..., 30으로 변화시킬 때 병렬처리 무리 짓기 성능을 비교한 것이다. 1,024개의 에이전트에 대하여 1,000프레임을 연속하여 시뮬레이션할 때 소요된 시간을 측정한 것이다. [표 1]에서 예측할 수 있듯이 k가 증가함에 따라 실행 시간이 증가되는 것은 스트리밍 멀티프로세서 수에 관계없이 동일하다. 특이한 점은 스트리밍 멀티프로세서의 수에 비례하여 성능 개선이 일어나지 않는 다는 것이다. 즉, [그림 6]에서 k가 256인 경우 멀티프로세서의 수가 15인 경우와 30인 경우 각각 35초와 31초의 실행 시간을 나타낸다. 이것 역시 앞에서와 마찬가지로 스트리밍 멀티프로세서의 증가는 전역메모리를 액세스할 때 일어나는 버스 충돌이 영향을 주고, 또한 [그림 4]에서 라인 5를 제외한 다른 부분의 비용도 적지 않기 때문이다.



[그림 6] 스트리밍 멀티프로세서(MP)수에 따른 병렬처리 방법의 성능

5. 결 론

논문은 최근 GPGPU로 많이 사용되는 CUDA 환경에서 무리 짓기에 대한 병렬 알고리즘을 제안하였다. 무리 짓기의 알고리즘에서 가장 많은 비용을 요구하는 이웃 에이전트를 찾는 것을 병렬처리 함으로써 성능 개선을 하였다. 성능 비교를 위하여 제안된 알고리즘을 GTX 285상에서 구현하였다.

다양한 실험을 통하여, CUDA를 사용한 병렬 처리는 에이전트 수가 1,024인 경우 최대 약 9배의 성능 개선이 있음을 알 수 있었다. 또한 스트리밍 멀티프로세서의 증가가 얼마나 성능 개선에 영향을 미치는지도 분석하였다.

GPU 프로그램에서 가장 큰 오버헤드는 *k*개의 가장 가까운 이웃을 찾기 위하여 *n*개의 데이터를 정렬하는 *BitonicSort*이다. 보다 좋은 성능 개선을 위하여 이에 대한 효율적인 연구가 필요하다.

참고문헌

[1] Reynolds, C. W., “Flocks, Herds, and Schools: A Distributed Behavioral Model”, SIGGRAPH, 21(4), pp. 25-34, 1987.

[2] Mat Buckland, “Programming Game AI by Example”, ISBN 1556220782, Wordware Publications, 2005.

[3] 이만휘, 박인규, 원석진 조성대, “GPU를 이용한 DWT 및 JPEG2000의 고속 연산,” 전자공학회 논문지 제44권 SP편 제6호, 2007.

[4] J. S. Charles, T. E. Potok, R. M. Patton, X. Cui, Flocking-based Document Clustering on the Graphics Processing Unit, DOE Office of Science Journal of Undergraduate Research, Volume VIII, 2008.

[5] Svetlin A Manavski and Giorgio Valle, “CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment,″ BioMed Center Bioinfformatics, 8:S10, 2008.

[6] 곽성호, 유민준, 이인권, “GPU를 이용한 실시간 분수 시뮬레이션을 위한 파티클 시스템,” 정보과학회 가을 학술발표논문집 Vol. 35, No. 2(B), 2008.

[7] NVIDIA. NVIDIA CUDA Programming Guide, 2.3 edition, August 2009.

[8] 이상길, 신병석, “GPU의 병렬 처리 기능을 이용한 메쉬 평탄화 가속 방법,″ 한국게임학회 논문지 v.9, no.2, 2009.

[9] P. Bakkum and K. Skadron. “Accelerating SQL Database Operations on a GPU with CUDA,″ In Proc. General-Purpose Computation on Graphics Processing Units, 2010.

[10] 이재문, “바이토닉 정렬을 사용한 CUDA의 성능 분석,” 한성대학교 공학연구 논문집 Vol.8, No.1, 2010.

[11] 권대중, 이남희, 김민성, 이재문, 조세홍, “CUDA를 이용한 무리 짓기 성능 개선,” 한국게임학회 춘계학술발표대회 논문집, 2010.

[12] Jae Moon Lee, “An efficient algorithm to find k-nearest neighbors in flocking behavior,″ Information Processing Letters, Vol. 110, Issues 14-15, 2010.

이 재 문 (Lee, Jae Moon)

1986 한양대학교 전자공학과(학사)
1988 한국과학기술원 전기및전자공학과(석사)
1992 한국과학기술원 전기및전자공학과(박사)
1994-현재 한성대학교 공과대학 멀티미디어공학과 교수

관심분야 : 데이터베이스, 기계학습, 게임프로그래밍