논 문
58-5-23

CUDA를 이용한 Particle Swarm Optimization 구현

Implementation of Particle Swarm Optimization Method Using CUDA

김 조 환* · 김 은 수* · 김 종 욱†
(Jo-Hwan Kim · Eun-Su Kim · Jong-Wook Kim)

Abstract - In this paper, particle swarm optimization (PSO) is newly implemented by CUDA (Compute Unified Device Architecture) and is applied to function optimization with several benchmark functions. CUDA is not CPU but GPU (Graphic Processing Unit) that resolves complex computing problems using parallel processing capacities. In addition, CUDA helps one to develop GPU softwares conveniently. Compared with the optimization result of PSO executed on a general CPU, CUDA saves about 38% of PSO running time as average, which implies that CUDA is a promising frame for real-time optimization and control.

Key Words : PSO, Heuristics, Global Optimization, GPU, Parallel Processing, CUDA

1. 서 론

현대인들은 수많은 전자 기기의 중심에서 살아가고 있다. 많은 전자 기기들 가운데 사람과 가장 밀접한 관계를 맺고 있는 것이 PC이며, PC는 메인보드, CPU등 많은 전자 부품으로 이루어져 있다. 그 중 GPU(Graphic Processing Unit)는 대부분의 사람들이 3D 게임 가속 능력에만 관심을 가지고 있고 그래픽처리만을 담당한다고 인식하고 있다. 하지만 GPU는 3D 게임은 물론 PC의 많은 작업들과 밀접한 관계를 맺고 있다.

GPU에 대해서 간단히 설명하면 GPU란 대용량 부동소수점 데이터를 병렬 고속 처리를 통해 매우 빠르게 계산할 수 있는 컴퓨터 부품의 핵심으로 3D 게임과 3D 자동차 시뮬레이션, H.264 등의 방송 콘텐츠와 U-헬스케어 분야에서 주로 활용되고 있다. 반면 CPU는 고정, 부동소수점을 모두 처리할 수 있지만 부동소수점 연산에서 GPU의 성능을 따라잡기엔 한계가 있다.

최근 PC의 발전으로 인해 CPU, RAM등 PC의 핵심 부품들의 성능이 향상되었다. 더불어 GPU의 성능 또한 CPU의 발전과 더불어 크게 발전했다. 그림 1은 GPU와 CPU의 성능을 비교한 그림이다[1]. 최근에 GPU의 연산 처리 능력은 CPU보다 훨씬 크게 발전하고 있다. 이렇게 GPU가 획기적으로 발전하면서 GPU의 본래 기능인 그래픽 처리 부분 뿐만 아니라 CPU가 처리해야 할 데이터를 대신 처리하게 되는 일이 빈번해졌다. 특히 NVIDIA社에서 CUDA(Compute Unified Device Architecture) 라이브러리와 AMD社의 ATI stream, Intel社의 Larrabee등 GPU 관련 기술들이 계속 등장하면서 'GPU컴퓨팅'이라는 말이 대두되었다.

GPU컴퓨팅은 GPGPU(General Purpose computation on GPUs)로 표현할 수 있는데, GPGPU는 GPU를 비 그래픽 애플리케이션에 응용하려는 것을 통칭해 부르는 용어이다. GPU는 복잡한 3D 게임을 지원하기 위해 빠른 속도로 발전했지만 최신 게임이 아닌 이상 GPU의 연산 능력을 100% 활용하는 경우는 드물다. GPGPU 기술은 GPU의 남은 연산 능력을 CPU의 연산을 보조하는데 쓰자는 취지에서 출발하였으나, 지금은 복잡한 과학 연산이나 암호 알고리듬 계산 등 그래픽 이외의 영역에서도 많이 사용되고 있다.

하지만 GPU를 비 그래픽 애플리케이션에 사용하는 것은 쉽지 않았다. GPU는 CPU와는 달리 GPU에 대한 아키텍처나 인스트럭션에 대한 정보가 드물었고, OpenGL이나 DirectX 뿐만 아니라 GLSL(OpenGL Shading Language), HLSL(High Level Shader Language) 등 각종 셰이딩 언어를 사용할 수 있어야 GPU를 활용할 수 있었다. 그러나 NVIDIA社에서 발표한 CUDA 라이브러리는 C언어 기반의 개발 환경이기 때문에 그래픽 API(Application Programming Interface)를 모르더라도 손쉽게 GPU를 활용해 성능 향상을 시킬 수가 있다.

현재 NVIDIA社에서는 CUDA를 이용해 동영상 인코딩 고속 구현, 금융 파생상품 측정 소프트웨어 가속화 등 다양한 어플리케이션의 고속화를 이루었다. 그리고 CUDA를 이용한 고속화 및 구현에 관한 논문들이 발표되는 등 다수의 관련 연구보고가 이루어지고 있다. CUDA 라이브러리를 사용한 블록 암호의 고속 구현화도 발표되었고[2], CUDA와 OpenMP를 이용한 신경망 구현에 대한 연구가 보고되었다[3].

* 준 회 원 : 동아대학 전자공학과 석사과정
† 교신저자, 정회원 : 동아대학 전자공학과 교수 · 공박
E-mail : kjwook@dau.ac.kr
접수일자 : 2009년 1월 16일
최종완료 : 2009년 4월 6일

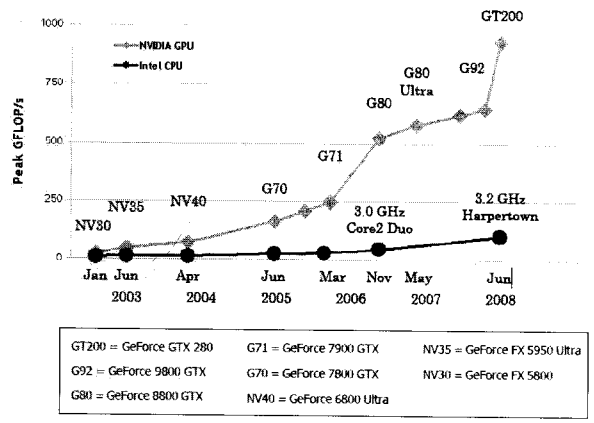

그림 1 GPU와 CPU의 성능 비교

Fig. 1 Performance comparison between CPU and GPU

본 논문에서는 전역 최적화 알고리듬 중 알고리듬이 간단하고 계산 시간이 짧으며, 고속의 최적화 능력을 갖는 particle swarm optimization(PSO) 알고리듬을 CUDA를 이용해 새롭게 구현하였다. 특히 GPU의 병렬 처리 기능을 PSO 알고리듬에 적용하였으며, 연산속도 개선 효과를 비교하기 위해 다양한 벤치마크 함수에 대해 CPU를 이용한 PSO 알고리듬의 탐색 속도와 비교 분석하였다. 그 결과 CUDA를 사용할 경우 GPU의 병렬 처리 기능으로 인해 전역 최적화 탐색속도가 크게 향상되었음을 보인다.

2. CUDA(Compute Unified Device Architecture)

2.1 GPU 구조

그래픽 카드는 CPU에서 생성한 디지털 신호를 영상신호로 바꾸고, 모니터로 출력하는 장치이며, GPU, VRAM, 비디오 BIOS, RAMDAC(Random Access Memory Digital to Analog Converter) 등으로 이루어져있다. 이 가운데 주요 핵심 부품은 전용 마이크로프로세서인 그래픽 프로세서(GPU)와 비디오 메모리이다. GPU란 CPU의 부족한 그래픽 작업의 병목 현상을 해결하기 위해 고안된 특수 목적 처리 장치로 3차원 그래픽 렌더링에 필수적인 부동 소수점 연산들을 주로 담당하는 전용 그래픽 마이크로프로세서이다.

그래픽 카드는 CPU로부터 받은 정보를 그래픽 메모리에 저장하고 이 정보를 CPU의 지시에 따라 GPU에서 영상정보로 바꾸고 다시 그래픽 메모리에 저장한다. 이때 호스트 컴퓨터로부터 받은 데이터를 프레임 버퍼에 넣는 과정에서 탁월한 병렬 처리 기능을 가지고 있어서 32비트 부동소수점 연산을 빠르게 수행할 수 있다.

이러한 병렬 처리 능력은 그림 2를 통해 확실히 알 수 있다. 그림 2는 CPU와 GPU의 비교 그림으로서, CPU는 대부분을 제어부분과 캐시 메모리가 차지하고 있고, 고성능 연산 능력을 가진 연산기(ALU)는 적은 수만을 가지고 있다. 반면 GPU는 제어 부분과 캐시 메모리가 거의 없는 것이 특징이고, 그 대신 저성능의 연산능력을 가진 연산기가 대부분을 차지하고 있다. 그러므로 연산이 많은 프로그램 수행 시 CPU에 비해 현저히 빠른 연산 성능을 보여준다[1].

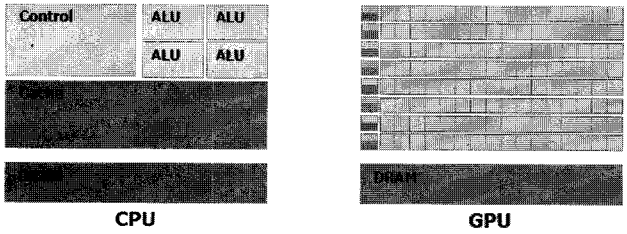

그림 2 CPU와 GPU 구조 비교

Fig. 2 Structure comparison between CPU and GPU

2.2 CUDA를 이용한 GPGPU 기술

서론에서 말한 바와 같이 일반 프로그래머가 GPU 프로그래밍을 하기에는 많은 어려움이 있다. 기존의 GPGPU 기술은 OpenGL, DirectX 등의 3D 그래픽 API를 통하지 않고 직접 하드웨어를 제어하기란 무척 어려웠다. 그래서 3차원 그래픽 API 등을 변용하거나 GLSL, HLSL 등 셰이딩 언어를 사용해 GPU를 다루는 방법이 사용되었다. 그러므로 그래픽 애플리케이션 작성에 익숙하지 않으면 GPU 활용은 거의 불가능에 가까웠다. 이렇듯 GPGPU 기술은 GPU와 셰이딩 언어의 전문가만이 사용할 수 있었다.

NVIDIA社에서 2007년에 발표한 CUDA는 범용 GPU 프로그래밍 라이브러리로서 C언어를 사용하여 GPU를 통한 범용 컴퓨팅을 할 수 있게 해주는 GPGPU 기술이다. NVIDIA社에서는 현재 드라이버 및 소프트웨어 toolkit 등을 무료로 제공하고 있으며, 행렬연산 라이브러리인 CUBLAS, 병렬연산 라이브러리인 CUDPP, Matlab 플러그인, 포토샵 플러그인 등 CUDA를 효율적으로 사용할 수 있는 기술들을 계속해서 발표하고 있다. CUDA는 리눅스와 윈도우 환경에서 동작하며 아직은 관련 문서도 부족하고 디버깅 환경이 불안정하지만, GPU의 성능을 최대한 활용할 수 있는 도구이다. AMD社나 Intel社의 경우 관련 GPGPU 기술이 문서나 개발 계획으로만 존재하는데 비해 CUDA는 CUDA 아키텍처의 이해와 C언어 코딩 능력만 있으면 누구나 GPGPU 기술을 손쉽게 활용할 수 있는 유일한 기술이다. CUDA는 기본적으로 GPU를 사용하므로 자사의 Geforce 80 series 이상의 그래픽카드에서만 사용 할 수 있다.

CUDA는 GPU의 메모리를 크게 GPU 내부의 온칩(on-chip)과 GPU 외부의 오프칩(off-chip)으로 나누어서 관리한다. 온칩 메모리에는 레지스터(register) 메모리, 공유(shared) 메모리가 있고, 오프칩 메모리에는 전역(global device) 메모리, 지역(local device) 메모리, 텍스처(texture) 메모리, 상수(constant) 메모리가 있다. 그림 3은 CUDA 메모리 구조와 각각의 메모리간의 연관 관계를 나타낸 그림이다[1]. 온칩 메모리인 레지스터 메모리와 공유 메모리는 GPU 전용 메모리로서 사용되며 한 사이클로 동작하여 매우 빠른 속도를 보인다. 오프칩 메모리인 텍스처 메모리와 상수 메모리는 읽기 전용 메모리로서, DRAM에 위치해 있고 캐시 메모리에 따라 1~100 사이클로 온칩 메모리의 속도보다는 느리지만 비교적 빠르게 동작한다. 그리고 지역 메모리와 전역 메모리는 읽기, 쓰기가 모두 가능한 메모리로서, 역시 DRAM에 위치해있고 대부분의 그래픽카드 메모리를

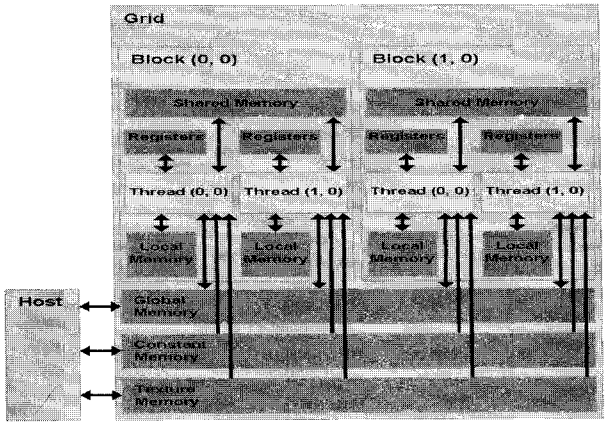

그림 3 CUDA 메모리 구조와 구성
Fig. 3 Structure and components of CUDA memory

사용하고 있다. 두 메모리 모두 캐시 메모리가 존재하지 않기 때문에 동작 속도는 느리다. 지역 메모리는 전역 메모리의 일부를 쓰레드별 고유영역으로 가지고 있다.

메모리는 그리드(grid), 블록(block), 쓰레드(thread)의 구성에 의해 영향을 받으며, 메모리의 계층 구조를 활용하여 그리드와 쓰레드로 프로그램을 구성한다. 모든 쓰레드에 동일한 프로그램이 할당되며, 실행 시 쓰레드의 고유번호가 파라미터로 사용되어 수행하는 내용이 달라지는 방식으로 작성된다. 그리고 프로그램 초기에 고정된 메모리양을 할당한 뒤 필요에 따라 데이터를 업로드하고 알고리듬을 수행하여 결과를 피드백 하는 방법을 사용한다.

2.3 CUDA 프로그래밍 기법

GPU의 다중 처리 장치는 쓰레드별로 호스트 PC에서 GPU로 호출되는 서브프로그램인 커널(kernel)을 수행한다. 커널은 그리드도 구성되고 그리드는 블록들로 구성되며 블록은 다시 여러 개의 쓰레드로 구성된다. 쓰레드들은 하나의 블록으로 묶여서 동시에 수행되고 블록 내의 쓰레드들은 서로 영향을 주지만 다른 블록 내부의 쓰레드들에게는 영향을 주지 않는다.

CUDA 프로그램을 수행 시킬 때 GPU에서 커널을 호출하기 전에 그리드를 지정해줘야 한다. 그리드를 지정해 주는 방법은 쓰레드의 크기와 블록의 개수를 지정해 주어서 커널에 사용될 그리드의 크기를 지정한다. 이렇게 구성된 쓰레드들은 서로 데이터를 공유하면서 병렬처리를 수행한다.

GPU로 데이터를 전달하는 방법은 대량의 데이터를 메인 메모리에서 그래픽 카드의 메모리로 데이터를 복사한 후 그 포인터를 전달한다. 그 후 GPU에서 처리를 한 후 실행 결과를 다시 그래픽 카드의 메모리에 넣은 후, 메인 메모리에서 다시 포인터로 읽어온다. 그림 4는 호스트 PC와 GPU 사이의 프로그램 연산 과정을 나타낸 것이다[1]. 호스트 PC의 DRAM에 저장된 데이터는 그래픽 카드의 전역 메모리를 통해 복사된다. 그리고 전역 메모리와 지역 메모리를 사용해서 GPU의 레지스터 메모리와 공유 메모리를 통해 연산을

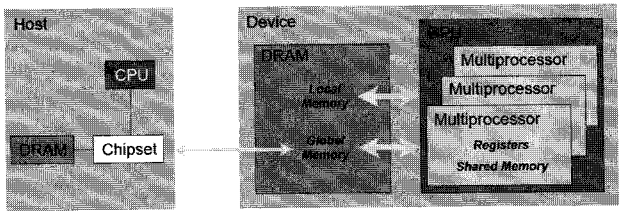

그림 4 호스트 PC와 GPU의 프로그램 연산과정
Fig. 4 Program processing of a host PC and GPU

수행한다. GPU에서 연산을 한 후 다시 지역 메모리와 전역 메모리를 통해 GPU 내의 DRAM으로 정보를 저장하게 된다. 그 후 다시 전역 메모리를 통해 호스트 PC의 DRAM으로 데이터를 복사 한 후 나머지 연산을 수행하게 된다.

CUDA 프로그래밍 기법은 기본 C 프로그래밍과 거의 유사하다. 기존 C 프로그래밍 소스에 GPU를 통해 연산을 수행 할 부분을 CUDA 라이브러리를 사용하여 프로그래밍하면 된다. 기존의 데이터를 커널에서 사용하기 위해서는 GPU 메모리로 복사가 되어야 한다. 이를 위해 cudaMalloc 함수를 통해 GPU에 사용할 메모리 공간을 할당하고, 호스트 PC의 DRAM에 있는 데이터를 GPU의 DRAM으로 복사하기 위해서는 호스트에서 디바이스 메모리 복사가 이루어져야 하므로, 일반적인 memcpy가 아닌 cudaMemcpy 함수의 cudaMemcpyHostToDevice 옵션을 사용한다. 그 후 커널 함수를 통해 연산을 한다. 반대로 GPU를 통한 연산이 다 끝난 후 GPU에서 CPU로 메모리를 복사해 와야 하는데, 이 때는 cudaMemcpy 함수의 cudaMemcpyDeviceToHost 옵션을 사용한다. 그리고 GPU의 연산을 위해 확보한 메모리 공간을 해제하는 cudaFree 함수를 사용한다.

CUDA 프로그램은 그림 5와 같이 구성된다. 가장 상위에는 CPU/GPU가 혼합된 소스 코드가 있고, BLAS나 FFT 등 미리 작성된 최적화 라이브러리가 있다. 이 소스 코드는 CUDA C 컴파일러인 nvcc로 컴파일 된다. 호스트 코드는 일반 C 컴파일러를 거쳐서 CPU에서 수행될 수 있는 오브젝트 코드가 되고, 디바이스 코드는 nvcc가 시스템에 의존하지 않는 어셈블리(PTX)로 컴파일 한다. PTX는 CUDA 드라이버를 통해 GPU에서 수행되는 코드가 된다. 보통 커널은 *.CU 확장자로 저장하는 것이 일반적이다.

3. PSO(Particle Swarm Optimization) 알고리듬

PSO 알고리듬은 1995년 Kennedy와 Eberhart에 의해 제안된 전역최적화 기법으로서, 조류나 어류 등의 무리가 서로의 정보를 공유하면서 먹이를 찾아가는 과정을 모사한 알고리듬이다[4]. 즉, PSO는 무리가 먹이를 찾아가는 과정에서 무리내의 각각의 개체가 가지고 있는 정보와 무리 전체가 가지고 있는 정보를 기초로 하여 행동한다는 개념을 최적화 과정에 도입하였다. PSO는 진화 계산을 기반으로 한 방법으로 유전 알고리듬(Genetic Algorithm, GA)과 비슷하나, 교배와 돌연변이 등을 하지 않아 알고리듬이 간단하고 계산 시간이 짧으며 대량의 메모리가 필요 없다. 또한 연속형과

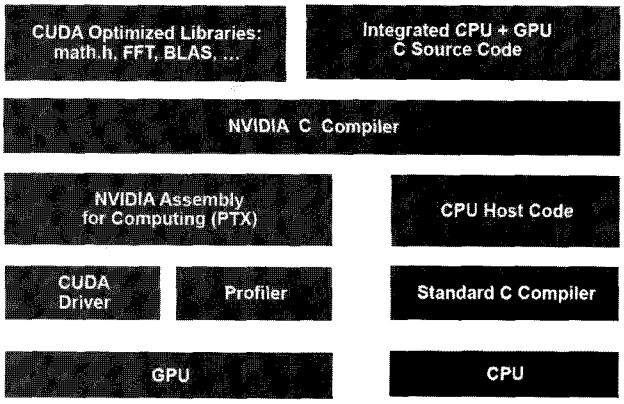

그림 5 CUDA 컴파일러의 연산과정
Fig. 5 Processing of CUDA compiling

비연속형의 문제 양쪽에 적용 가능하다[4].

PSO에서 각각의 개체들은 자신들이 발견한 최적의 목적함수 $F(pbest)$와 그 해의 위치 벡터인 $pbest$를 기억하고 있다. 그리고 집단 전체에서 발견한 해 중에서 가장 최적의 목적함수 $F(gbest)$와 그 해의 위치 벡터 $gbest$의 정보를 공유하고 있다. 각 개체는 현재의 위치 벡터와 속도벡터, 그리고 $pbest$와 $gbest$의 정보를 이용해서 아래 식 (1)에 의해 다음 위치로 이동을 하게 된다. 그리고 각 개체의 위치 벡터의 수정은 현재의 위치와 수정된 속도를 이용해서 아래 식 (2)와 같이 행해진다[5].

$$V_k^{next} = a_1 V_k^{now} + a_2 rand(pbest_k - S_k^{now}) + a_3 rand(gbest - S_k^{now}), \quad k = 1,2,\cdots,N \tag{1}$$

$$S_k^{next} = S_k^{now} + V_k^{next}, \quad k = 1,2,\cdots,N \tag{2}$$

k, N : 각 에이전트와 전체 에이전트의 수

V_k^{now}, V_k^{next} : 현재와 다음의 속도 벡터

S_k^{now}, S_k^{next} : 현재와 다음의 위치 벡터

$pbest_k, gbest$: 각 에이전트의 최적해의 위치벡터와 전체 에이전트의 최적해의 위치벡터

a_1, a_2, a_3 : 가중치 계수

$rand$: 0과 1 사이의 난수 발생 함수

그림 6은 각각의 개체들이 현재의 위치에서 다음 위치로 이동하는 과정을 나타낸 그림이다[6]. 각각의 개체들은 각자의 최적해인 $pbest$의 정보를 가지고 있고, 집단에서 발견한 $gbest$의 정보를 가지고 있다. 각 개체들은 다음의 방법으로 위치를 수정하게 된다. 각 개체는 현재 향하고 있는 $pbest$에서 집단 전체가 발견한 최적의 해인 $gbest$가 있는 방향으로 위치를 변경하고자 한다. 이때 식 (1)의 우변의 2, 3항을 이용해서 $pbest$와 $gbest$의 합인 V_k^{mod}와 현재의 속도인 V_k^{now}의 합을 이용해 새로운 속도 V_k^{next}를 계산하고, V_k^{next}에 따라서 각 개체는 새로운 위치인 S_k^{next}로 위치를 변경하게 된다.

PSO 알고리듬은 다음의 연산 과정을 거친다[5].

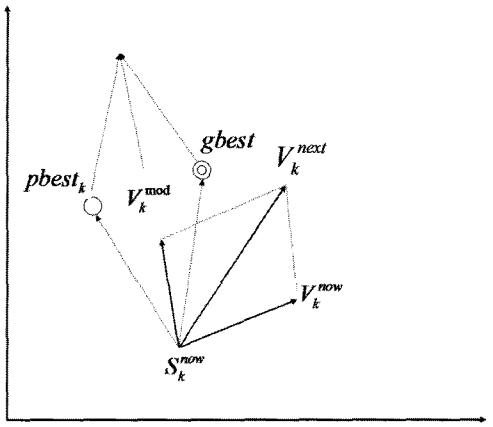

그림 6 각 개체의 위치와 속도 변화 과정
Fig. 6 Search principle of PSO for changing velocity and position of each particle

1) 전체 에이전트에 대해 초기 위치 벡터 s와 속도 벡터 v를 난수를 이용해 설정한다. 이 때 초기의 위치벡터 s를 $pbest$로 한다.
2) 1)에서 가장 우수한 $pbest$를 전체 에이전트에 관한 최적의 벡터 $gbest$로 한다.
3) 에이전트의 속도 벡터 v를 식 (1)을 사용해 갱신하고 위치 벡터 s를 식 (2)를 사용해 갱신한다.
4) 에이전트에 관해 현재의 위치에서 목적함수 값 $F(s)$가 $F(pbest)$보다 우수한 목적함수 값을 가지면 s를 $pbest$로 할당한다.
5) 전체 에이전트에 대해서 $F(pbest)$가 $F(gbest)$보다 우수한 목적함수 값을 가지면 $pbest$를 $gbest$로 할당한다.
6) 충분히 좋은 적합 도를 가진 해를 얻거나 충분히 많은 세대를 거치게 될 때까지 3)부터 루프를 반복 수행한다.

4. CUDA를 이용한 PSO 알고리듬 구현

본 논문에서는 CUDA의 성능을 비교하기 위해 PSO 알고리듬을 Microsoft Visual Studio에서 코딩하고 컴파일한 후 CPU로 구동했다. 실험에 사용된 CPU의 클럭은 듀얼코어 2.4Ghz이며, CUDA를 위한 그래픽카드로는 NVIDIA社의 GeForce 8500GT를 사용했다.

식 (1)에서 a_1 가중치 계수는 0.8, a_2와 a_3의 가중치 계수는 동등 가중치인 2로 설정하였다. 이것은 식 (1)의 제 2항과 제3항의 계수가 0에서 1까지의 난수와 가중치 계수의 곱으로 되어있기 때문에 그 평균이 1이 되도록 하기 위한 것이다[7]. 그리고 에이전트들은 30으로 설정하였으며, 이러한 PSO 파라미터들은 [8]을 참고하였다.

실험에 사용된 테스트 함수는 Six-hump camel-back (CA) 함수, Branin(BR) 함수, Goldstein-Price(GP) 함수, Rastrigin(RA) 함수, Hartman(H3, H6) 함수들이다[9]. Hartman 함수에서 H3는 3차원 함수를, H6는 6차원 함수를 나타내고 계수 c_i, α_{ij}, p_{ij}는 [10]에 기재되어 있다.

표 1 테스트 함수 정보

Table 1 Specification of test functions

Test Function	수식	알려진 전역 최적값	전역해
CA	$f_{CA}(x)=4x_1^2-2.1x_1^4+\frac{1}{3}x_1^6+x_1x_2-4x_2^2+4x_2^4$	-1.031628	0.089880 -0.712600
BR	$f_{BR}(x)=(x_2-\frac{5.1}{4\Pi^2}x_1^2+\frac{5}{\Pi}x_1-6)^2+10(1-\frac{1}{8\Pi})\cos x_1+10$	0.397883	3.142000 2.275000
GP	$f_{GP}(x)=[1+(x_1+x_2+1)^2(19-14x_1+3x_1^2-14x_2+6x_1x_2+3x_2^2)]$ $\times[30+(2x_1-3x_2)^2(18-32x_1+12x_1^2+48x_2-36x_1x_2+27x_2^2)]$	3.000000	0.000000 -1.000000
RA	$f_{RA}(x)=x_1^2+x_2^2-\cos 18x_1-\cos 18x_2$	-2.000000	0.000000 0.000000
H3	$f_{H3}(x)=-\sum_{i=1}^{4}c_i\exp[-\sum_{j=1}^{3}\alpha_{ij}(x_j-p_{ij})^2]$	-3.862782	0.114 0.556 0.852
H6	$f_{H6}(x)=-\sum_{i=1}^{4}c_i\exp[-\sum_{j=1}^{6}\alpha_{ij}(x_j-p_{ij})^2]$	-3.322368	0.201 0.150 0.477 0.275 0.311 0.657

표 2 테스트 함수 실험 결과

Table 2 Optimization results of test functions

Test Function		CA	BR	GP	RA	H3	H6
CPU를 이용한 PSO 알고리듬	비용함수 계산횟수	2495	2006	5080	5911	2426	2540
	수행속도 (ms)	2.977	2.840	6.157	9.873	9.617	12.865
CUDA를 이용한 PSO 알고리듬	비용함수 계산횟수	2445	1990	5032	5895	2317	2428
	수행속도 (ms)	1.515	1.561	4.028	7.066	6.173	8.392
	수행속도 감소정도 (%)	49.11	45.04	34.58	28.43	35.81	34.77

표 1은 테스트 함수에 관련된 사항을 나타낸 것으로, 각 테스트 함수의 수식과 현재 알려진 최적값, 그리고 해당 전역해를 나타냈다. 작성된 PSO 알고리듬은 해당 함수에서 현재 탐색된 전역 최적값과 알려진 전역 최적값과의 오차가 10^{-6} 이내로 들어오면 연산이 종료가 되도록 구현되었다. 그리고 연산이 종료할 때까지의 비용함수 계산횟수와 수행속도를 측정하였다.

표 2는 총 100번의 무작위 연산 결과의 평균값을 나타낸 것이다. 비용함수 계산횟수는 오차 범위가 10^{-6} 이하가 되어 연산이 종료 될 때 까지 목적함수를 몇 번 계산했는지를 나타낸 값이다. 그리고 수행속도는 시뮬레이션이 종료될 때까지 컴퓨터 운영 시간을 나타낸 수치이다. 수행속도 감소정도는 두 가지 경우의 PSO 알고리듬의 수행속도를 비교하여 CUDA를 이용했을 때 수행속도의 향상도를 나타낸 것이다.

비용함수 계산횟수는 두 경우가 비슷한 결과를 보이는데 이는 동일한 PSO 알고리듬을 사용했기 때문이다. 그러나 수행속도 면에서는 CUDA를 이용한 PSO 알고리듬이 CPU를 이용한 PSO 알고리듬보다 탁월한 성능을 보였다. 테스트 함수가 비교적 간단한 Six-hump camel-back 함수와 Branin 함수의 경우 기존의 PSO 알고리듬에 비해 49.11%, 45.04% 연산 시간이 단축되었다. 그리고 다차원 테스트 함수인 Hartman 함수의 경우는 3차원에 대해서는 35.81%, 6차원에 대해서는 34.77%의 수행속도 개선이 있었다. Rastrigin 함수의 경우 가장 낮은 값인 28.43%의 개선도를 보였는데, 이는 수식에 있는 삼각함수에 기인한다. 그러므로 CUDA는 삼각함수를 계산할 경우 크게 속도를 개선하지 못함을 알 수 있다. 결론적으로 간단한 테스트 함수일수록 수행속도의 개선도가 커졌고, 연산 시간이 오래 걸리는 테스트

함수일수록 수행속도의 개선도가 낮았다. 그리고 수행속도의 감소정도, 즉 개선도는 최소 28.43%에서 최대 49.11%이며, 평균 개선도는 37.96%이다.

5. 결 론

본 논문에서는 최적화 알고리듬 중 고속의 전역 최적화 성능을 갖는 PSO 알고리듬을 GPU 병렬 기능이 있는 CUDA를 사용하여 구현하고, CPU를 이용한 PSO의 알고리듬과 성능을 비교 분석하였다. 일반적인 연산 성능 비교를 위해 6개의 테스트 함수에 대해 적용한 결과 CUDA가 CPU를 사용한 경우에 비해 평균 38% 정도 연산 속도가 개선됨을 확인했다. 현재 GPU 성능이 꾸준히 개선되고 있으므로 이러한 연산 속도 개선이 더욱 급속히 진행 될 것이며, 이는 빠른 시일 내에 GPU를 이용하여 실시간 최적화나 실시간 제어가 가능함을 의미한다. 향후 연구는 휴머노이드 로봇 이족보행 시뮬레이션에서 사용된 유전 알고리듬이나 DEAS(Dynamic Encoding Algorithm for Searches)를 CUDA로 구현하고, 로봇 비전에 사용될 각종 이미지 처리 어플리케이션 프로그램을 CUDA를 이용해서 구현할 예정이다.

감사의 글

이 논문은 동아대학교 학술연구비 지원에 의하여 연구되었습니다.

참 고 문 헌

[1] NVDIA CUDA Programming Guide V2.0, http://kr.nvidia.com/object/cuda_develop_kr.html, accessed on 13 April 2009.

[2] 염용진, 조용국, "GPU용 연산 라이브러리 CUDA를 이용한 블록암호 고속 구현", 정보보호학회논문지, 제18권, 제3호, pp.23-32, June 2008.

[3] 장홍훈, 정기철, "CUDA와 OpenMP를 이용한 신경망 구현", 한국정보과학회 종합학술대회, 제35권, 제1호, pp.289-290, June 2008.

[4] J. Kennedy and R. Eberhart, "Particle swarm optimization," *IEEE International Conference on Neural Network*, vol. 1, IV, Perth, Australia, Nov./Dec. 1995.

[5] 이중상, 이상욱, 장석철, 석상문, 안병하, "Particle swarm optimization을 이용한 블랙 숄츠 옵션가격 결정모형", 한국경영과학회 춘계학술대회, pp.745-747, April 2005.

[6] 유명련, "Particle swarm optimization 탐색과정의 가시화를 위한 툴 설계", 멀티미디어학회논문지, 제6권, 제2호, pp.332-339, May 2003.

[7] Y. Shi and R Eberhart, "Parameter selection in particle swarm optimization," *Annual Conference on Evolutionary Programming*, San Diego, USA, 1998.

[8] L. Chuan and F. Quanyuan, "The standard particle swarm optimization algorithm convergence analysis and parameter selection," *Natural Computation*, vol. 3, pp.823-826, Aug. 2007.

[9] J. -W. KIM and S. W. KIM, "A fast computational optimization method: univariate dynamic encoding algorithm for searches (uDEAS)," *IEICE Trans. Fundamentals of Electronics*, vol. E90-A, pp.1679-1689, Aug. 2007.

[10] A. Törn and A. Žilinskas, *Global Optimization*, Springer-Verlag, Berlin, 1989.

저 자 소 개

김 조 환(金 祚 煥)

1983년 월 일생. 2009년 동아대 전자공학과 졸업. 2009년~현재 동 대학원 전자공학과 석사과정 재학중.

Tel : 051-200-5579

E-mail : kimhades@hotmail.com

김 은 수(金 銀 宿)

1983년 1월 26일생. 2008년 동아대 전자공학과 졸업. 2008년~현재 동 대학원 전자공학과 석사과정 재학중.

Tel : 051-200-5579

E-mail : tacctics@hotmail.com

김 종 욱(金 鍾 旭)

1970년 10월 24일생. 1998년 포항공대 전자전기공학과 졸업. 2000년 동 대학원 전자전기공학과 졸업(석사). 2004년 동 대학원 전자전기공학과 졸업(박사). 2004년~2006년 포스코 기술연구소 전기강판연구 그룹 연구원2. 2006년~현재 동아대학교 대학원 전자공학과 조교수.

Tel : 051-200-7714

Fax : 051-200-7712

E-mail : kjwook@dau.ac.kr