Journal of the KSTLE Vol. 28, No. 4, August 2012, pp. 160~166
The Korean Society of Tribologists & Lubrication Engineers

그래픽처리장치를 이용한 레이놀즈 방정식의 수치 해석 가속화

명훈주† · 강지훈‡ · 오광진
한국과학기술정보연구원 슈퍼컴퓨팅센터

Accelerating Numerical Analysis of Reynolds Equation Using Graphic Processing Units

Hun-Joo Myung†, Ji-Hoon Kang‡ and Kwang-Jin Oh
Supercomputing Center, Korea Institute of Science and Technology Information (KISTI)
(Received May 15, 2012; Revised June 25, 2012; Accepted June 30, 2012)

Abstract – This paper presents a Reynolds equation solver for hydrostatic gas bearings, implemented to run on graphics processing units (GPUs). The original analysis code for the central processing unit (CPU) was modified for the GPU by using the compute unified device architecture (CUDA). The red-black Gauss-Seidel (RBGS) algorithm was employed instead of the original Gauss-Seidel algorithm for the iterative pressure solver, because the latter has data dependency between neighboring nodes. The implemented GPU program was tested on the nVidia GTX580 system and compared to the original CPU program on the AMD Llano system. In the iterative pressure calculation, the implemented GPU program showed 20-100 times faster performance than the original CPU codes. Comparison of the wall-clock times including all of pre/post processing codes showed that the GPU codes still delivered 4-12 times faster performance than the CPU code for our target problem.

Keywords – graphic processing unit (그래픽처리장치), reynolds equation (레이놀즈방정식), red-black gauss-siedel iterative method (레드-블랙 가우스-사이델 반복법)

1. 서 론

레이놀즈 방정식을 수치적으로 푸는 유체윤활 해석은 보다 현실적인 모델링 방법의 발달과 점점 더 복잡해지는 해석 대상 문제들로 인해 더 많은 계산량과 계산시간을 필요로 하고 있다. 이러한 수치해석의 복잡도는 컴퓨터의 발전속도와 함께 증가하여 왔으며, 특히 계산량이 많은 열탄성유체윤활 (Thermo-elastohydrodynamic lubrication, TEHL)문제나 비정상상태의 동적 해석 문제의 경우 문제의 규모에 따라 수시간~수일에 걸친 계산이 필요한 경우도 있다.

이처럼 문제 해석에 필요한 계산량의 증가 때문에 컴퓨터 계산 성능 향상은 지속적으로 요구되어 왔으며, 실제로 2000년대 중반까지는 무어의 법칙에 따르는 CPU 클럭 속도의 가파른 증가 덕분에 별다른 노력 없이도 컴퓨터의 향상된 계산 성능을 이용할 수 있었다. 그러나 2000년대 중반 이후 전력 및 발열 문제로 인해 CPU 클럭 속도는 더 이상 증가할 수 없는 지경에 이르렀고, 2004년 인텔 CPU 기준으로 약 3 GHz의 클럭 속도를 한계로 더 이상 증가하고 있지 않다[1]. 클럭 속도의 증가 대신 선택된 것은 멀티 코어 혹은 다중 코어라 불리는 방법으로 높아진 집적도를 이용하여 하나의 CPU에 여러 개의 계산 코어를 집적시키는 방법이다. 이는 하나의 CPU에 대해 많은 계산 코어를 집어 넣어 단일 계산 코어의 계산 속도를 증가시키는 대신, 한 번에 처리할 수 있는 계산량을 증가시키는 방법이라 할 수 있겠다. 다만 이 경우 기존의 순차적인 계

†주저자 : hjmyung@kisti.re.kr
‡책임저자 : jhkang@kisti.re.kr

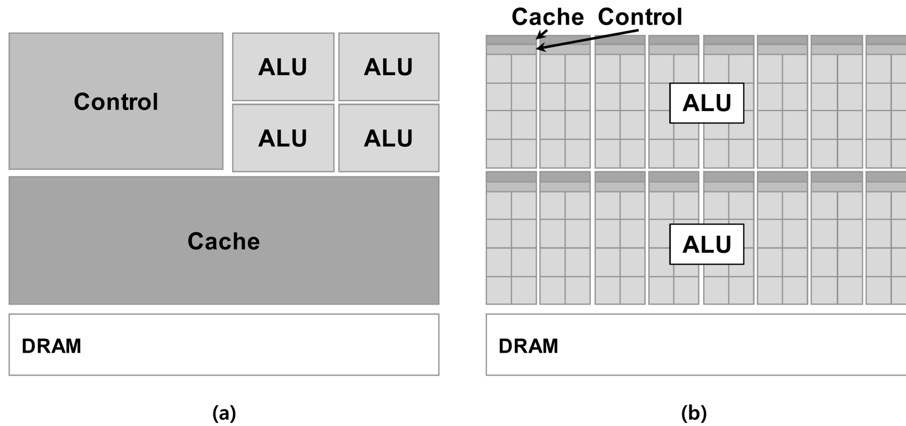

Fig. 1. Schematics of (a) CPU and (b) GPU.

산 알고리즘 대신 여러 개의 계산 코어를 동시에 활용할 수 있는 병렬 계산 알고리즘으로 프로그램이 구현되어야 한다.

병렬 계산 알고리즘에 가장 널리 쓰이고 있는 병렬 프로그래밍 모델로는 CPU에 기반한 OpenMP[1,2]와 MPI[3,4] (Message Passing Interface)를 꼽을 수 있다. OpenMP는 지시어에 기반한 병렬 프로그래밍 모델로, 병렬로 처리될 부분에 간단한 지시어를 추가하여 해당 부분만 각각의 계산 코어에 할당하여 계산한 후 계산결과를 하나로 합하는 fork-join 형태의 병렬 계산을 수행하게 하는 방법이다. 이 방법은 지시어 사용을 통해 비교적 쉽게 병렬 계산 알고리즘을 구현할 수 있다는 장점이 있지만, fork-join 모델에 기반한 병렬 작업의 생성 및 소멸에 부하가 걸리기 때문에 많은 계산 코어를 사용할 수 없는 단점이 있다. MPI는 각각의 계산 코어가 계산할 부분을 미리 사용자가 나누어준 후, 각각의 계산에 필요한 데이터들은 메시지 전달을 통해 주고 받는 방법이다. MPI는 계산 영역 분할과 데이터 통신들을 모두 사용자가 프로그래밍해야 하기 때문에 OpenMP에 비해 구현이 어려운 반면에 통신 알고리즘의 최적화를 통해 훨씬 크고 복잡한 문제를 훨씬 많은 CPU에서 실행되게 할 수 있다. 최근에는 CPU에 기반한 병렬 프로그래밍 모델 이외에도 핵심 계산 부분을 CPU와 나누어 수행하는 가속기 (accelerator) 또는 보조프로세서(Co-processor)를 활용하는 병렬 프로그래밍 모델 또한 활발히 도입되고 있다. 이러한 가속기 또는 보조프로세서 중에서 현재 가장 널리 활용되고 있는 것으로는 그래픽 처리장치 (Graphic Processing Unit, GPU)을 꼽을 수 있다.

GPU는 도입 초기부터 가장 대표적인 보조프로세서의 하나로 자리매김하며 CPU대신 그래픽 처리를 수행해 왔다. GPU는 CPU와는 달리 그래픽 처리에 특화된 수많은 산술 연산 유닛과 이에 관련된 인스트럭션 (instruction)을 담고 있어 CPU보다 훨씬 빠른 그래픽 처리를 수행할 수 있다[5]. 이와 같은 GPU의 강력한 산술 연산 기능에 매력을 느낀 많은 연구자들은 GPU 도입 초기부터 GPU를 CPU처럼 일반 연산에 활용하고자 시도해 왔으나, GPU를 이용하기 위해서는 C나 Fortran과 같은 일반 프로그래밍 언어 대신 그래픽 함수 및 라이브러리를 이용하는 GPU 전용 shade 프로그래밍을 해야 하는 까닭에 일반 응용연구자들이 GPU를 이용하기란 거의 불가능에 가까운 일이었다. 이와 같은 GPU 프로그래밍 언어의 높은 장벽은 GPU 회사 중 하나인 nVidia에서 CUDA (compute Unified Device Architecture) 라는 소프트웨어 아키텍쳐를 2007년에 발표함으로써 상당히 낮아지게 되었다. CUDA는 기존의 C 언어 문법과 거의 유사한 방식으로 프로그래밍이 가능하며 특히 GPU가 보유한 수많은 산술 연산 유닛과 인스트럭션의 활용, 그리고 GPU 메모리에 대한 접근을 C언어와 호환되는 인터페이스를 통해 가능하게 함으로써 GPU 프로그래밍에 대한 부담을 크게 낮추었다. 특히 C 언어로 프로그래밍된 기존 프로그램 중에서 많은 계산이 필요한 부분만 CUDA를 이용하여 병렬 프로그래밍할 수 있기 때문에, 기존 프로그램의 틀을 그대로 유지한 채 필요한 부분에서만 GPU를 이용할 수 있게 되어 프로그램의 유지보수 및 확장이 훨씬 용이해졌다.

Fig. 1은 CPU와 GPU의 구조를 나타내고 있다.

GPU가 CPU와 구별되는 중요한 특징은 하나의 칩에 스트림프로세서(Stream processor, SP)라 불리는 산술 연산을 위한 산술논리연산자(arithmetic logic unit, ALU)가 수백개 이상 집적되어 있다는 점이다. CPU는 범용의 복잡한 인스트럭션을 처리하기 위한 제어부(control)와 복잡한 인스트럭션 처리시에 임시 저장소로 활용할 커다란 캐쉬(cache) 메모리, 그리고 범용의 인스트럭션셋(set)을 가지는 소수의 산술논리연산자로 구성되어 있다. 반면 GPU는 Fig. 1(b)처럼 수백개에 달하는 산술논리연산자들을 가지고 있지만, 처리해야 하는 인스트럭션이 단순한 산술 연산에 국한되어 있으며 산술 처리된 결과 또한 재활용할 일이 거의 없기 때문에 CPU에 비해 훨씬 단순한 제어부와 훨씬 작은 크기의 캐쉬 메모리를 가지고 있다. 이러한 구조 때문에 GPU는 수백개의 스트림프로세서를 통해 동일한 형태의 산술 연산을 동시다발적으로 처리하는데 매우 우수한 강점을 가지고 있다. 따라서 GPU는 가장 계산이 많이 필요한 반복 수렴 계산이나 행렬 연산 등 시간이 많이 걸리는 단순한 형태의 반복계산을 담당하고, CPU는 그 외 입출력이나 계산 전후처리 등의 일반적인 계산을 담당하는 형태로, GPU는 CPU의 보조계산 장치 또는 가속계산장치의 역할을 하게 된다.

이러한 장점에 힘입어 CUDA를 활용한 GPU 프로그래밍은 5년 내외의 짧은 역사에도 불구하고 분자동역학[6], 금융공학[7], 전산유체역학[8] 등 많은 계산과학분야에서 활용되고 있으며, 순수 연구용의open-source 프로그램뿐만 아니라 상당수의 상용 프로그램[9]에도 이미 적용되어 그 활용 범위를 점점 넓혀가고 있다.

본 논문에서는 이처럼 차세대 고성능 계산 도구로 주목받고 있는 GPU 프로그래밍를 이용하여 레이놀즈 방정식을 푸는 비정상 유체윤활해석 프로그램을 구현하고 성능 향상도를 조사하였다. 이를 위해 CPU 용으로 개발된 기존 수치 해석 프로그램의 순차 계산 알고리즘을 GPU 구조에 맞도록 새로이 병렬 계산 알고리즘으로 변경하고, 이를 CUDA를 이용하여 구현하였다. 새롭게 구현된 CUDA 코드에 대해 GPU 병렬화로 얻어진 계산 시간의 감소량과 GPU 초기화나 메모리 동기화 등 추가로 발생하는 계산 시간의 증가량을 측정하여 전체적인 성능 향상도를 조사하였으며, 또한 격자수를 달리하여 테스트를 수행하여 격자수에 따른 계산량의 변화가 전체 성능 향상도에 미치는 영향을 조사하였다.

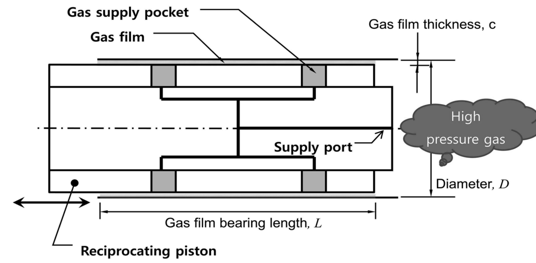

Fig. 2. Schematic of analysis model.

2. 구현 방법

2-1. 해석 대상 및 해석 방법

본 연구에서 해석 대상으로 선택한 정압기체베어링을 Fig. 2에 나타내었다. 피스톤 헤드 쪽의 고압 기체는 공급포트와 공급포켓을 통해 윤활면에 공급되어 피스톤의 왕복운동을 지지하게 된다. 지배방정식으로는 윤활면에서 식 (1)과 같은 레이놀즈 방정식을 사용하였으며, 공급 포켓에서는 식(2)와 같이 윤활면으로 나가는 유량과 공급포트를 통해 들어온 유량이 같아야 하는 유량 보존 방정식을 사용하였다.

$$\frac{\partial}{\partial Z}\left(PH^3\frac{\partial P}{\partial Z}\right)+\frac{\partial}{\partial \theta}\left(PH^3\frac{\partial P}{\partial \theta}\right)=\Lambda\frac{\partial PH}{\partial Z}+\Gamma\frac{\partial PH}{\partial \tau} \quad (1)$$

$$\dot{M}=\oint_{\partial S}(\Lambda PH-PH^3\nabla P)\cdot\vec{n}ds=A(P_{supply}^2-P_{pocket}^2) \quad (2)$$

여기서 P, H, Z 는 각각 무차원화된 압력, 유막두께, 길이방향좌표이며 Λ과 Γ는 각각 무차원화된 베어링수와 시간상수이다. 유량보존 방정식에서의 P_{supply}와 P_{pokcet}은 각각 피스톤 헤드부의 공급기체압력과 공급포켓에서의 압력이고, A는 유량상수로써 공급포트가 capillary restrictor라 가정하면 다음과 같다.

$$A=\frac{12\mu R_g T}{c^3 p_a^2}\frac{\rho_s \pi d^4}{256\mu L_S} \quad (3)$$

여기서 μ, R_g, T, c, p_a는 각각 기체점도, 기체상수, 온도, 정압베어링 간극, 대기압이며, ρ_s, d, L_s은 기체밀도, 공급포트의 직경, 공급포트의 길이이다. 피스톤이 왕복운동을 하기 때문에 베어링 길이 L은 다음과 같이 시간에 대한 sin 함수로 나타낼 수 있다.

$$L=L_0+l\sin(2\pi ft) \quad (4)$$

여기서 L_0, l, f는 각각 베어링길이의 평균값, 왕복운

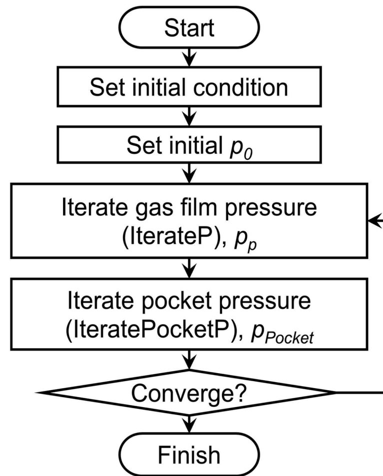

Fig. 3. Flow chart.

동 진폭, 그리고 왕복운동의 주파수를 나타낸다.

레이놀즈 방정식은 유한체적법을 이용하여 차분화 했으며 차분화된 방정식은 Gauss-Seidel 반복법을 이용해서 계산하였다. 또한 레이놀즈 방정식과 공급포켓에서의 유량보존식을 만족시키기 위해 베어링 내부의 압력값과 공급포켓에서의 압력값을 교대로 수렴시키면서 계산하였다. 프로그램의 순서도를 Fig. 3에 나타내었다.

2-2. Red-Black Gauss-Seidel

Gauss-Seidel 반복법에서는 현재 격자점의 값을 업데이트 할 때 필요한 주변 격자점 값으로 항상 마지막에 업데이트된 값을 이용한다. Fig. 4(a)에 나타낸 2차원 문제를 예를 들어보면, (i, j) 격자점 값을 업데이트 하기 위해 필요한 5개의 격자점 중 (i, j), $(i+1, j)$, $(i, j-1)$의 세 격자점은 아직 계산이 안되었기 때문에 이전 스텝에서 업데이트된 값을 이용하지만, $(i-1,j)$과 $(i,j-1)$의 두 격자점은 현재 스텝에서 이미 업데이트가 되었기 때문에 현재 스텝에서의 값을 이용하게 된다. 따라서 마지막 격자점 값을 새로이 업데이트하기 위해서는 결국 모든 격자점의 업데이트된 값이 필요하게 되며 이는 각 격자점 값을 업데이트 하는 작업은 모두 순차적으로 이루어져야 하는 것을 의미한다. 이처럼 일반적인 Gauss-Seidel 방법에서는 현재 스텝에서의 모든 격자점의 값들이 서로 의존되어 있기 때문에 병렬 연산이 불가능하다.

앞서 기술한 격자점 값끼리의 의존성 때문에 병렬

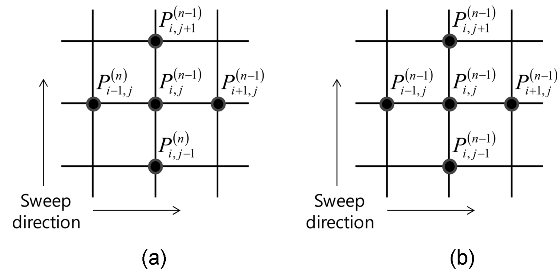

Fig. 4. Schematic of (a) Gauss-Siedel and (b) Jacobi iterations.

컴퓨팅에서는 일반적인 Gauss-Seidel 반복법 대신 Jacobi 반복법이나 red-black Gauss-Seidel(RBGS)을 이용해 왔다. Fig. 4(b)에 나타낸 2차원 Jacobi 반복법에서는 현재 격자점 값을 업데이트 할 때 필요한 5개의 주변 격자점 값으로 현재 스텝의 값을 이용하는 대신 이전 스텝의 값을 이용하기 때문에, 현재 스텝에서의 격자점 값 의존성이 전혀 없으며 이전 스텝의 격자점 값만 알고 있다면 모든 격자점들의 값을 독립적으로 업데이트할 수 있다. 다만 Jacobi 반복법은 수렴속도가 느리고 발산하는 경우가 많기 때문에 본 연구에서는 red-black Gauss-Seidel (RBGS)방법을 이용하였다.

RBGS는 본 논문에서 다루고 있는 문제처럼 한 격자점의 값을 업데이트하는데 바로 인접한 격자점들의 값만 필요한 경우, 계산 스텝을 반으로 나누어 $(n+1/2)$ 스텝과 $(n+1)$ 스텝에서 계산할 격자점들을 달리함으로써 데이터끼리의 의존성을 제거하는 방법이다. 우선 계산할 격자점들을 서로 Fig. 5와 같이 검은색(filled)과 붉은색(empty) 격자점으로 나눈다. $(n+1/2)$ 스텝에서의 계산시에는 붉은색 격자점에 대해서만 계산을 수행한다. 붉은색 격자점의 값을 계산할 때에는 검은색 격자점과 이전 스텝의 값만 필요하기 때문에, 모든 붉은 격자점의 값들은 데이터 의존성이 없이 완전히 독립적으로 계산될 수 있다. 두 번째로 $(n+1)$ 스텝에서는 $(n+1/2)$ 스텝에서 계산된 붉은색 격자점의 값과 이전 스텝의 값을 이용하여 검은색 격자점의 값을 계산한다. 이 경우 또한 $(n+1/2)$ 스텝에서 처럼 모든 검은 격자점들의 값들을 완전히 독립적으로 계산할 수 있다. 따라서 각각의 격자점들을 GPU의 계산 스레드(thread)에 할당하여 데이터 의존성 없이 병렬 계산을 수행할 수 있다.

본 연구에서는 동일한 θ 값을 갖는 격자점들을 스레드들의 묶음인 스레드 블록에 할당하였고, 이후 Z값에

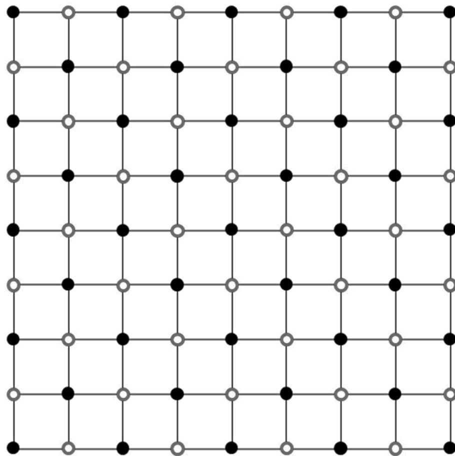

Fig. 5. Red-Black nodes in Red-Black Gauss-Seidel iteration.(red: empty, black: filled)

```
// Red-Black Gauss-Siedel iterative pressure solver structure
void Pressure_CUDA::iterate(){
    ...
    do{
        redblack = 0;      //1st pass
        iteratePressure_kernel<<<blocks, threads, Mem_size>>>(…,redblack);
        cutilSafeCall(cudaMemcpy(…, cudaMemcpyDeviceToHost));
        redblack = 1;      //2nd pass
        iteratePressure_kernel<<<blocks, threads, Mem_size>>>(…,redblack);
        cutilSafeCall(cudaMemcpy(…, cudaMemcpyDeviceToHost));
        // … Error calculation
    }while(error>criteria);
}
__global__ void iteratePressure_kernel(int *d_MeshType, …, int redblack ) {
    int iIdx = threadIdx.x;
    int jIdx = blockIdx.x;
    if(((iIdx+jIdx)&1) == redblack) { // Red or Black node
        // … Coefficient calculation
        // … Pressure update
        // … Error calculation
    }
}
```

Fig. 6. Part of implemented code.

따라 각 격자점들을 스레드 블록 내의 스레드에 할당하였다. 이렇게 구현된 코드의 일부를 Fig. 6에 나타내었다. 변수 redblack값이 0일 때에는 Fig. 5의 붉은 격자점들이, redblack이 1일 때에는 검은 격자점들이 계산된다.

2-3. 테스트 조건

GPU 성능을 테스트 하기 위해 테스트 장비로 nVidia의 GTX580을 사용하였다. GTX580은 클럭스피드가 795 MHz이고 512개의 스트림프로세서를 가지고 있다. 성능 향상 비교를 위해 사용한 CPU는 AMD의 Llano A8-3850으로, 2.9 GHz의 클럭스피드를 가지고 있다. 문제의 규모 및 계산량을 달리하기 위해 격자크기를 각각 56×80, 84×120, 112×160, 168×240, 252×360으로 달리하였다. GPU 계산의 경우 red-black Gauss-Seidel 알고리즘을 적용한 것 이외에는 모든 계산조건과 수렴조건에 대해 CPU 계산과 동일한 조건을 사용하였다.

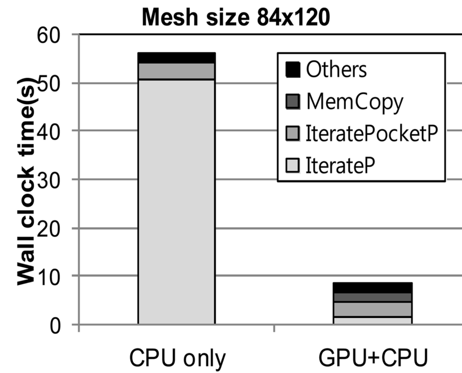

Fig. 7. Wall-clock-time comparison.

3. 결과 및 고찰

3-1. 성능 향상

Fig. 7은 격자크기 84×120인 문제에 대해 CPU 만 사용한 경우와 GPU를 함께 사용한 경우의 wall-clock-time을 비교한 결과를 나타내고 있다. Fig. 7에서 IterateP 항목은 윤활면의 압력을 계산하는데 걸린 시간이고 IteratePocketP는 공급포켓의 압력을 계산하는데 걸린 시간이다. 또한 GPU 그래프에서만 MemCopy라는 항목을 볼 수 있는데, 이는 GPU에서의 계산을 위해 필요한 데이터를 GPU 메모리로 복사하는데 걸린 시간이다. 마지막으로 Others 항목은 계산 전후처리 항목으로 이 부분은 공통적으로 CPU만 계산을 수행하는데 걸린 시간이다.

Fig. 7에서 알 수 있듯이 IterateP 부분의 계산 시간이 크게 줄어든 것을 볼 수 있다. CPU만 이용했을 경우 약 51초 걸리던 시간이 GPU를 이용함으로써 1.4초까지 줄어들어 IterateP부분에서만 약 35배의 성능향상을 얻을 수 있었다. 반면 IteratePocketP 부분의 계산시간은 거의 줄어들지 않았는데, 격자점의 수가 매우 작고 또한 공급포켓의 압력평균값을 계산하는 부분

Table 1. Timing results of three mesh sizes

Mesh size	56×80		112×160		252×360	
CPU/ GPU	CPU only	CPU+ GPU	CPU only	CPU+ GPU	CPU only	CPU+ GPU
IterateP	17.7 (86%)	0.8 (16%)	135.4 (92%)	2.3 (14%)	2861.6 (93%)	26.8 (10%)
Iterate pocketP	2.2 (11%)	2.6 (52%)	6.0 (4%)	4.9 (29%)	72.2 (2%)	33.4 (13%)
Mem Copy	-	0.8 (16%)	-	3.7 (21%)	-	54.1 (21%)
Others	0.7 (3%)	0.8 (16%)	5.7 (4%)	6.0 (36%)	140.6 (5%)	141.9 (56%)
Total (sec.)	20.6 (100%)	5.0 (100%)	147.1 (100%)	16.9 (100%)	3074.4 (100%)	256.2 (100%)

때문에 뚜렷한 성능향상을 얻을 수 없었다. MemCopy에 소모된 추가 시간과 GPU 병렬화가 안되어있는 Others 부분을 고려하더라도 전체 wall-clock-time은 56초에서 8.5초로 줄어들어, GPU를 이용할 경우 CPU에 비해 약 1/6정도의 시간으로 동일한 문제를 계산할 수 있었다.

3-2. 격자수 영향

일반적으로 GPU는 반복 계산량이 많아질수록 그 성능을 드러낼 수 있기 때문에 본 연구에서도 격자크기를 달리하며 성능향상 정도를 측정하였다. 격자 크기가 각각 56×80, 112×160, 252×360인 경우에 대해 테스트를 수행하여, 앞서 설명한 각 항목의 계산 시간 및 총 wall-clock-time과 전체 계산 시간대비 백분율을 Table 1에 나타내었다.

GPU를 이용할 경우 공통적으로 IterateP 부분에서 계산 시간의 감소가 눈에 띄게 확인되며, 격자수가 커질수록 IteratePocketP 부분의 계산 시간 또한 감소하는 것으로 보인다. 특히 격자수가 커질수록 따라 CPU만 이용할 경우 IterateP의 계산 시간이 두드러지게 증가하는 반면, GPU를 이용할 경우 이 부분의 계산 시간이 급격하게 감소하는 것을 알 수 있다. IterateP부분만 놓고 보면, GPU를 이용했을 때의 계산 시간은 CPU만 이용했을 때에 비해 격자수 56×80인 경우 약 1/22, 격자수 112×160인 경우 약 1/60, 격자수 252×360인 경우 약 1/106으로 줄어들었다. 이처럼 대부분의 계산을 차지하는 IterateP 부분의 계산 시간을 격자수가 커질수록 GPU가 매우 효율적으로 줄여주기 때문에 전체 계산 시간 또한 격자수가 커질수록 줄어들게 된다. GPU 이용시 전체 계산 시간은, 격자수 56×80인 경우 약 1/4, 격자수 112×160인 경우 약 1/9, 격자수 252×360인 경우 1/12로 감소하였다. 또한 격자수 252×360인 경우를 살펴보면, 가장 시간이 많이 걸리는 부분은 CPU가 계산하는 부분인 Others 부분임을 볼 수 있다. 따라서 GPU를 통해 반복 계산에 대해 계산 시간을 수십배 이상 단축시킬 수 있음에도 불구하고, 실제 문제의 경우 GPU로 구현되지 않은 부분 때문에 전체 성능은 크게 향상되지 않을 수도 있다. 이 경우 병렬화되지 않은 CPU 계산 부분이 전체 성능을 저하시키는 주요 병목지점(bottle-neck)으로 되고 있다.

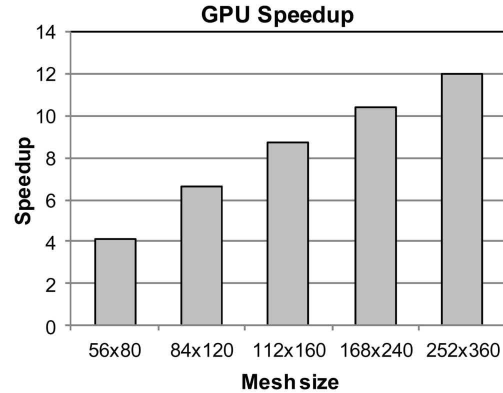

Fig. 8. Speedup of GPU for various mesh sizes.

3-3. 스피드업(Speedup)

병렬 컴퓨팅에서 많이 등장하는 개념인 스피드업은 병렬 프로그램의 성능 향상을 측정하는 척도로써 다음과 같이 원본 프로그램 대비 몇 배의 계산 시간 단축을 가져왔는지를 나타낸다.

$$\text{스피드업} = \frac{\text{원본 프로그램 실행 시간}}{\text{병렬 프로그램 실행 시간}}$$

이를 이용하여 본 연구에서 구현한 GPU 병렬 프로그램의 스피드업을 Fig. 8에 나타내었다. 격자수가 증가할수록 IterateP부분의 계산 시간이 증가하여 GPU의 활용도가 증가하므로 격자수가 커질수록 전체 스피드업 또한 증가하게 된다. GPU를 이용함으로써 CPU만 이용했을 때에 비해 최소 4배, 최대 12배의 스피드업을 얻을 수 있었다.

4. 결 론

본 논문에서는 CUDA를 이용한 GPU 프로그래밍을 통해 레이놀즈 방정식을 푸는 비정상 유체윤활해석 프로그램을 구현하고 성능 향상도를 조사하였다. 기존 수치 해석 프로그램의 순차 계산 알고리즘을 red-black Gauss-Seidel 알고리즘을 이용하여 GPU 구조에 맞도록 병렬 계산 알고리즘으로 변경하였다. 구현된 GPU 병렬 프로그램을 통해 CPU만 이용했을 때에 비해 반복 계산 부분에서는 최대 106배, 전체 계산으로는 최대 12배의 성능 향상을 달성할 수 있었다. 격자 크기가 늘어나고 반복 계산 부분이 많아질수록 GPU이용시의 성능은 CPU만 이용했을 때에 비해 월등히 앞섰으며, 이 경우 CPU가 담당하는 병렬화되지 않은 부분이 전체 성능 향상에 걸림돌이 되는 병목 현상 또한 확인할 수 있었다.

참고문헌

1. 정영훈, “OpenMP 병렬프로그래밍,” Chap. 1, pp. 4-7, 프리렉, 대한민국, 2011.
2. The OpenMP API Specification for parallel programming, http://www.openmp.org.
3. The Message Passing Interface (MPI) standard, http://www.mcs.ahl.gov/research/projects/mpi.
4. Open MPI: Open Source High Performance Computing, http://www.open-mpi.org.
5. Kirk, D.B. and Hwu, W.W., “Programming Massively Parallel Processors: Hands-on Approach,” Chap.1, pp. 19-36, Elsevier, 2010.
6. Stone, J.E., Hardy, D.J., Ufimtsev, I. S., and Schulten, K., “GPU-accelerated Molecular Modeling Coming of age,” *Journal of Molecular Graphics and Modeling*, Vol. 29, No. 2, pp.116-125, 2010.
7. Surkov, V., “Parallel Option Pricing with Fourier Space Time-stepping Method on Graphics Processsing Units,” *Parallel Computing*, Vol. 36, No. 7, pp. 372-380, 2010.
8. Tölke, J. and Krafczyk, M., “TefaFLOP Computing on a Desktop PC with GPUs for 3D CFD,” *International Journal of Computational Fluid Dynamics*, Vol. 22, No. 7, pp. 443-456, 2008.
9. Browell, R. and Hutchings, B., “Bigger, Better, Faster: HPC Technology Leadership,” ANSYS Advantage, Vol. 5, No. 3, pp. 6-8, 2011.