creative commons

COMMONS DEED

저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.

변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

공학석사 학위논문

불연속 갤러킨 방법의 효율적인 계산을 위한 GPU 적용 연구

Study on Application of GPU Computing for Efficient Computation of Discontinuous Galerkin Method

2020 년 2 월

서울대학교 대학원

기계항공공학부

김 성 태

초 록

본 논문에서는 불연속 갤러킨 방법의 효율적인 계산을 위해 CUDA를 이용하여 GPU 계산을 적용하는 연구를 다루고 있다. GPU(Graphic processing unit)은 본래 화면을 출력하기 위해 사용되는 장치이지만 수많은 코어(core)를 기반으로 지연 감춤(latency hiding)을 통해 대규모 연산에서 큰 성능을 낼 수 있다는 사실이 알려지면서 고성능 컴퓨팅 적용을 위해 활발히 연구되고 있다. 불연속 갤러킨 방법에서는 격자 내부의 해를 기저함수들의 합으로써 근사하는데 이로 인해 작은 스텐실을 유지하면서 임의의 공간정확도를 구현할 수 있다. 또한 불연속 갤러킨 방법에서는 연산이 텐서 축약으로써 표현되며 높은 연산 강도(arithmetic intensity)로 인해 GPU 계산 적용이 유리하다는 특징이 있다. GPU에서 텐서 축약을 효율적으로 계산하기 위해 CUDA를 이용하여 GEMM(General matrix multiplication) 커널을 작성하였으며 cuBLAS(Cuda basic linear algebra) 라이브러리보다 더 높은 성능을 달성하였다. 완성한 GPU 코드로 수치실험을 진행하여 기존의 CPU 코드와 성능을 비교 및 분석하였다.

주요어 : 전산유체역학, GPU 계산, CUDA, 불연속 갤러킨 방법, 텐서 축약, GEMM 커널

학 번 : 2018-25343

목 차

초록 i

목차 ii

표 목차 iv

그림 목차 v

제 1 장 서 론 1

1.1 불연속 갤러킨 방법 1

1.2 GPU 계산 및 병렬화 원리 2

제 2 장 지배방정식 및 수치기법 5

2.1 지배방정식 5

2.2 불연속 갤러킨 방법 6

2.3 시간 적분 기법 7

제 3 장 CUDA 프로그래밍 최적화..... 9

3.1 CUDA 언어 10

3.1.1 스레드 체계 10

3.1.2 메모리 체계 12

3.2 텐서 축약 15

3.3 CUDA 커널 16

3.3.1 GEMM 커널 16

3.3.2 TC 커널 18
3.3.3 플럭스 계산 커널 20
3.3.4 시간 적분 커널 20
3.4 CUDA Stream 21

제 4 장 수치 실험 24
4.1 실험 환경 24
4.1.1 프로세서 24
4.1.2 라이브러리 25
4.2 실험 방법 25
4.2.1 해석 대상 및 조건 25
4.2.2 격자 27
4.3 실험 결과 29
4.3.1 외재적 시간 기법 29
4.3.2 내재적 시간 기법 36

제 5 장 결 론 43

참고문헌 45

Abstract 47

표 목차

표 3.1 GEMM 커널 메모리 대역폭(GB/s) 18
표 4.1 수치 실험을 수행한 프로세서 제원 24
표 4.2 수치 실험에 사용한 격자 종류 28
표 4.3 외재적 시간 기법 연산 크기 (Grid#1) 32
표 4.4 외재적 시간 기법 연산 크기 (Grid#2) 35
표 4.5 내재적 시간 기법 연산 크기 (Grid#3) 39
표 4.6 내재적 시간 기법 연산 크기 (Grid#4) 42

그림 목차

그림 1.1 TOP500 중 GPU를 장착한 슈퍼컴퓨터의 수 3
그림 1.2 메모리 지연 숨김(memory latency hiding) 4
그림 3.1 CPU와 GPU의 아키텍처 9
그림 3.2 그리드와 블록 10
그림 3.3 메모리 접근 체계 14
그림 3.4 GPU 메모리 체계 15
그림 3.5 불연속 갤러킨 방법의 텐서 축약 16
그림 3.6 GEMM 커널 정사각 행렬 곱 성능 (Gflop/s) 17
그림 3.7 GEMM 커널을 이용한 TC1 계산 19
그림 3.8 GEMM 커널을 이용한 TC2 계산 19
그림 3.9 CUDA Stream 개념도 21
그림 3.10 전체적인 코드 알고리즘 22
그림 3.11 계산 커널 구성 23
그림 4.1 점성 사인파 문제 초기 조건의 밀도 contour 26
그림 4.2 외재적 시간 기법 시간 비용 비교 (Grid#1) 31
그림 4.3 RHS 계산에서 달성한 성능 비교 (Grid#1) 31
그림 4.4 RHS 계산 커널들의 시간 비용 비교 (Grid#1) 32
그림 4.5 외재적 시간 기법 시간 비용 비교 (Grid#2) 33
그림 4.6 RHS 계산에서 달성한 성능 비교 (Grid#2) 34

그림 4.7 RHS 계산 커널들의 시간 비용 비교 (Grid#2) 34
그림 4.8 내재적 시간 기법 시간 비용 비교 (Grid#3) 38
그림 4.9 시스템 행렬 계산에서 달성한 계산 성능 비교
(Grid#3) 38
그림 4.10 내재적 시간 기법 계산 커널들의 시간 비용 비교
(Grid#3) 39
그림 4.11 내재적 시간 기법 시간 비용 비교 (Grid#4) 41
그림 4.12 시스템 행렬 계산에서 달성한 계산 성능 비교
(Grid#4) 41
그림 4.13 내재적 시간 기법 계산 커널들의 시간 비용 비교
(Grid#4) 42

제 1 장 서 론

1.1 불연속 갤러킨 방법

불연속 갤러킨(Discontinuous Galerkin, DG) 방법은 대표적인 고차 정확도 CFD(Computational fluid dynamics) 수치기법 중 하나로 유한체적법(Finite volume method, FVM)의 한계를 극복하기 위해 최근 활발히 연구되고 있다 [1]. 여러 산업분야에서 많이 사용되고 있는 유한체적법 기반 수치기법들은 대부분이 2차 공간 정확도이기 때문에 작은 스케일의 난류 유동을 정밀하게 포착하는데 문제를 겪고 있다. 또한, 유한체적법에서는 공간 정확도를 높이기 위해 인접한 cell들을 참조해야하기 때문에 스텐실이 커지게 되는데 이는 계산의 병렬화에 불리하게 작용한다. 반면, 불연속 갤러킨 방법에서는 cell 내부의 해를 기저함수들의 합으로 근사하기 때문에 작은 스텐실을 유지하면서 임의의 공간 정확도를 구현할 수 있고 작은 스케일의 난류 유동을 정확하게 포착할 수 있다.

한편, 불연속 갤러킨 방법에서는 대부분의 연산들을 텐서 축약(tensor contraction)의 형태로써 표현할 수 있다. 불연속 갤러킨 방법의 텐서 축약들은 차수가 높지 않기 때문에 컴퓨터 상에서 행렬 곱 함수를 이용해 효율적으로 계산할 수 있다 [2]. 유한체적법의 연산들과 달리 불연속 갤러킨 방법의 텐서 축약 연산은 부동소수점 연산 작업이 메모리

읽기 작업보다 높은 비율을 차지하므로 연산 강도(arithmetic intensity)가 높아 컴퓨터의 계산 성능을 크게 활용할 수 있다는 특징이 있다. 특히 대규모 행렬 연산에서 높은 성능을 보이는 GPU(Graphics processing unit)에서 텐서 축약을 효율적으로 계산할 수 있기 때문에 최근 고차 정확도 수치기법에 GPU 계산을 적용하려는 연구가 활발히 이루어지고 있다 [3, 4, 5].

1.2 GPU 계산 및 병렬화 원리

GPU는 원래 화면 출력 혹은 3D 그래픽 렌더링을 목적으로 만들어진 장치이지만 높은 부동소수점 이론적 연산 성능을 활용하여 과학 응용 프로그램들을 가속하기 위해 연구되기 시작했다. 초창기에는 GPU를 활용하기 위해 그래픽스 인터페이스에 대한 이해가 필수적이었지만 보다 더 편리한 GPU 프로그래밍을 위해 CUDA(Compute unified device architecture), OpenCL(Open computing language) 등의 프로그래밍 언어가 보급되면서 GPU를 활용한 고성능 컴퓨팅(High-performance computing, HPC) 연구가 더욱 활발히 이루어지고 있다. GPU 제조사에서도 이러한 흐름을 따라가기 위해 높은 배정밀도 부동소수점 연산 성능을 지닌 고성능 컴퓨팅 목적의 GPU를 출시하기 시작했으며 그림 1.1 [6]과 같이 오늘날 많은 수의 최신 슈퍼컴퓨터들이 GPU를 기반으로 하고 있다.

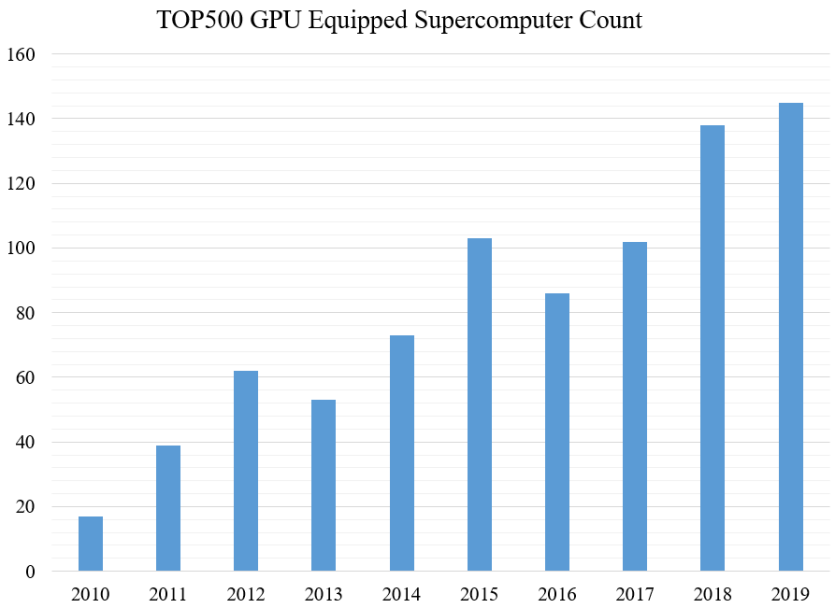


그림 1.1 TOP500 중 GPU를 장착한 슈퍼컴퓨터의 수

GPU가 높은 부동소수점 연산 성능을 지닐 수 있는 것은 GPU의 수많은 코어(core)들을 통해 성공적으로 메모리 지연을 숨기는 것(memory latency hiding)이 가능하기 때문이다. CPU는 코어 수가 GPU보다 적지만 높은 코어 클럭(clock), 빠르고 용량이 큰 캐시 메모리(cache memory)를 이용해 연산의 지연(latency)을 최소화 하도록 설계되었다. 이러한 점을 반영하여 대부분의 CPU 기반 유동 해석 코드는 계산 영역을 코어 수로 나누어 각 코어에 할당하는 큰 그레인 병렬화(coarse-grain parallelism)의 방식이다. 반면, GPU는 코어의 클럭은 상대적으로 낮지만 코어 수가 매우 많아서 연산의 동시성(concurrency)를 극대화 하는데 초점이 맞춰져 있다. 더 구체적으로 GPU의 스레드(thread)들은 상대적으로 가볍기(light-

weight) 때문에 문맥을 바꾸는 것(context switch)이 쉽다. 대부분의 연산 작업들에서 스레드는 계산에 필요한 데이터를 받을 때 메모리 읽기 작업의 높은 지연으로 인해 스톨(stall)에 걸리게 된다. GPU에서는 스레드가 상대적으로 가볍기 때문에 스케줄러(scheduler)가 다른 가용한 스레드들에 명령을 내림으로써 그림 1.2와 같이 메모리 지연을 숨길 수 있다. GPU에서는 주어지는 데이터의 크기가 클수록 메모리 지연의 효과가 크기 때문에 기존의 CPU 방식을 그대로 적용하면 높은 성능을 내기 어려우며 계산 및 데이터를 더 잘게 쪼개는 미세 그레인 병렬화(fine-grain parallelism) 을 적용해야 한다. [7]

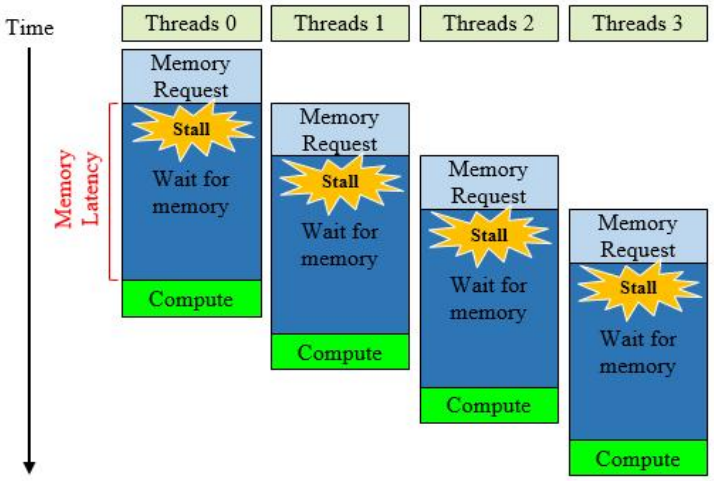


그림 1.2 메모리 지연 숨김(memory latency hiding)

본 연구에서는 불연속 갤러킨 방법 기반 유동 해석 프로그램에 효율적으로 GPU 계산을 적용한 CUDA 코드를 개발하고 기존의 CPU 코드와 성능을 비교, 분석하는 것을 목표로 하였다.

제 2 장 지배방정식 및 수치기법

2.1 지배방정식

점성 압축성 유동에 대한 지배방정식은 다음과 같다.

$$\frac{\partial \boldsymbol{q}}{\partial t} + \nabla \cdot (\mathbf{F}_c - \mathbf{F}_v) = 0 \tag{2.1}$$

여기서 $\boldsymbol{q}$ 는 보존 벡터(conservative vector), $\mathbf{F_c}$ 는 비점성 플럭스 텐서(convective flux tensor), $\mathbf{F_v}$ 는 점성 플럭스 텐서(viscous flux tensor)로 3차원에서 각각 다음과 같다.

$$\boldsymbol{q} = \begin{pmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ \rho e_t \end{pmatrix}, \ \mathbf{F}_c = \begin{pmatrix} \rho u & \rho v & \rho w \\ \rho u^2 + p & \rho uv & \rho uw \\ \rho vu & \rho v^2 + p & \rho vw \\ \rho wu & \rho wv & \rho w^2 + p \\ \rho uH & \rho vH & \rho wH \end{pmatrix}$$

$$\mathbf{F}_v = \begin{pmatrix} 0 & 0 & 0 \\ \tau_{xx} & \tau_{yx} & \tau_{zx} \\ \tau_{xy} & \tau_{yy} & \tau_{zy} \\ \tau_{xz} & \tau_{yz} & \tau_{zz} \\ u\tau_{xx} + v\tau_{xy} + w\tau_{xz} + q_x & u\tau_{yx} + v\tau_{yy} + w\tau_{yz} + q_y & u\tau_{zx} + v\tau_{zy} + w\tau_{zz} + q_z \end{pmatrix} \tag{2.2}$$

여기서 ρ, p, e_t, H 와 (u, v, w) 는 각각 밀도, 압력, 전에너지(total energy), 전엔탈피(total enthalpy)와 속도 벡터이다. 압력은 이상 기체 상태방정식에 의해 다음과 같이 구할 수 있다. 비열비 γ는 1.4이다.

$$p = (\gamma - 1)\rho\left(e_t - \frac{1}{2}(u^2 + v^2 + w^2)\right) \tag{2.3}$$

τ_{ij} 는 점성 플럭스 텐서로 뉴턴 유체(Newtonian fluid)를 가정하여 스토크스 가설(Stokes' hypothesis)에 의해 다음과 같이 표현할 수 있다.

$$\tau_{ij} = \mu\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} - \frac{2}{3}\frac{\partial u_k}{\partial x_k}\delta_{ij}\right) \tag{2.4}$$

q_x 는 열 플럭스로 푸리에 법칙(Fourier's law)에 의해 아래와 같이 표현된다.

$$q_i = -k\frac{\partial T}{\partial x_i} \tag{2.5}$$

여기서 μ, k, T는 각각 점성 계수, 열전도 계수, 온도를 뜻한다.

2.2 불연속 갤러킨 방법(DG 방법)

식 (2.1)에 기저함수 $\phi_i^{(j)}$를 곱하고 cell Ω_i에 대해서 적분하면,

$$\int_{\Omega_i} \frac{\partial \boldsymbol{q}_h}{\partial t}\phi_i^{(j)} dV + \int_{\partial\Omega_i} \phi_i^{(j)}\left(\widehat{\mathbf{F}_c \cdot \mathbf{n}} - \widehat{\mathbf{F}_v \cdot \mathbf{n}}\right) dA = \int_{\Omega_i} (\mathbf{F}_c - \mathbf{F}_v) \cdot \nabla\phi_i^{(j)} dV \tag{2.6}$$

와 같은 weak form을 얻을 수 있다. [1] 여기서 $\widehat{\mathbf{F}_c \cdot \mathbf{n}}$와 $\widehat{\mathbf{F}_v \cdot \mathbf{n}}$는 각각 단조 비점성 수치 플럭스(monotone numerical convective flux), 단조 점성 수치 플럭스(monotone numerical viscous flux)이다. $\widehat{\mathbf{F}_v \cdot \mathbf{n}}$와 $\mathbf{F}_v$는 BR2 기법에 의해 다음과 같이 계산할 수 있다. [8]

$$\widehat{\mathbf{F}_v \cdot \mathbf{n}} = \frac{1}{2}\left(\mathbf{F}_v(\boldsymbol{q}_h^-, \nabla\boldsymbol{q}_h^- + \mathbf{r}_e^-) + \mathbf{F}_v(\boldsymbol{q}_h^+, \nabla\boldsymbol{q}_h^+ + \mathbf{r}_e^+)\right) \cdot \mathbf{n} \tag{2.7}$$

$$\mathbf{F}_v = \mathbf{F}_v(\boldsymbol{q}_h, \nabla \boldsymbol{q}_h + \mathbf{R}^-) \tag{2.8}$$

여기서 ± 부호는 이웃한 면을 기준으로 계산이 neighbor cell 혹은 owner cell에서 이루어지는 것을 뜻하며 $\mathbf{r}_e^{\pm}$ 와 $\mathbf{R}^-$ 는 BR2 기법에서 local lifting operator와 global lifting operator로 계산할 수 있다. DG 방법에서는 수치 해를 식 (2.9)과 같이 기저함수들의 합으로 근사하는데 기저함수의 orthonormality에 의해 식 (2.6)를 식 (2.10)로 간소화할 수 있다.

$$\boldsymbol{q}_h(\mathbf{x}, t) = \sum_{j=0}^{N_k} q_i^{(j)}(t)\phi_i^{(j)}(\mathbf{x}) \tag{2.9}$$

$$\frac{d\boldsymbol{q}_i^{(j)}}{dt} = -\int_{\partial\Omega_i} \phi_i^{(j)}\left(\widehat{\mathbf{F}_c \cdot \mathbf{n}} - \widehat{\mathbf{F}_v \cdot \mathbf{n}}\right) dA + \int_{\Omega_i} (\mathbf{F}_c - \mathbf{F}_v) \cdot \nabla \phi_i^{(j)} dV \tag{2.10}$$

식 (2.10)의 면적분항과 부피적분항은 physical domain 상에서 Gauss 구분구적법을 사용하여 계산하였다.

2.3 시간 적분 기법

본 연구에서는 DG 방법의 안정적인 시간 적분을 위해 외재적 시간 기법으로 SSPRK45(5-stage, 4th-order Strong stability preserving Runge-Kutta) 기법을 사용하였다. [9]

$$\frac{dq}{dt} = L(q,t) \tag{2.11}$$

$$\begin{aligned}
&q^{(1)} = q^n + 0.391752226571890\Delta t L(u^n),\\
&q^{(2)} = 0.444370493651235q^n + 0.555629506348765u^{(1)}\\
&\quad +0.368410593050371\Delta t L(u^{(1)}),\\
&q^{(3)} = 0.620101851488403q^n + 0.379898148511597q^{(2)}\\
&\quad +0.251891774271694\Delta t L(q^{(2)}),\\
&q^{(4)} = 0.178079954393132q^n + 0.821820045606868q^{(3)}\\
&\quad +0.544974750228521\Delta t L(q^{(3)}),\\
&q^{n+1} = 0.51723167197058q^{(2)}\\
&\quad +0.096059710526147q^{(3)} + 0.063692468666290\Delta t L\left(q^{(3)}\right)\\
&\quad +0.386708617503269q^{(4)} + 0.226007483236906\Delta t L(q^{(4)})
\end{aligned} \tag{2.12}$$

내재적 시간 기법으로는 Rosenbrock 방법 계열의 RODAS5(8-stage, 5th-order Rosenbrock) 기법을 사용하였다. [10, 11]

$$\left(\frac{\mathbf{I}}{\gamma_{ij}\Delta t} - \mathcal{J}\right)\mathbf{g}_i = -\mathbf{R}\left(\hat{\mathbf{q}}|_{t_n} + \sum_{j=1}^{i-1} a_{ij}\mathbf{g}_j\right) + \sum_{j=1}^{i-1}\left(\frac{c_{ij}}{\Delta t}\right)\mathbf{g}_j\,, \quad i = 1, \cdots, N_{stages}$$

$$\mathcal{J} = \left.\frac{\partial \mathbf{R}}{\partial \hat{\mathbf{q}}}\right|_{t_n}$$

$$\hat{\mathbf{q}}|_{t_n+1} = \hat{\mathbf{q}}|_{t_n} + \sum_{i=1}^{N_{stage}} b_i \mathbf{g}_i \tag{2.13}$$

수치 해를 n차 다항식으로 근사한 불연속 갤러킨 방법에서 시간 전진은 다음과 같이 제한된다.

$$\Delta t = \frac{CFL}{2n+1}\frac{h}{|\lambda^{\max}|} \tag{2.14}$$

여기서 h는 cell의 부피에 세제곱근을 취한 값이고 $\lambda^{\max}$는 최대 정보 전파속도이다.

제 3 장 CUDA 프로그래밍 최적화

GPU는 그림 3.1과 같이 구조적으로 CPU와 다소 차이가 있기 때문에 CPU에서의 병렬화 방식을 그대로 GPU 계산에 적용하면 좋은 성능을 기대하기 어렵다. GPU의 성능을 잘 활용하여 효율적인 계산을 하기 위해서는 GPU의 아키텍처(architecture)를 고려하여 병렬화를 적용해야 한다. GPU에서 스레드들을 체계적으로 할당하고 이를 데이터에 대응시키는 것을 스레드 맵핑(thread mapping)이라고 하는데 이를 구현하는 방법은 무수히 많지만 스레드 체계와 메모리 체계와 같은 GPU 프로그래밍에 대한 이해가 반영되어야만 효율적인 GPU 계산이 가능하다.

본 장에서는 먼저 GPU 프로그래밍 특성들에 대해 설명하고 이를 CUDA 코드에 반영하여 효율적으로 불연속 갤러킨 방법을 계산하는 과정에 대해 다룰 것이다.

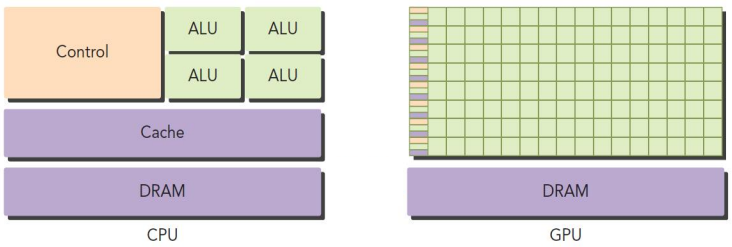


그림 3.1 CPU와 GPU의 아키텍처

3.1 CUDA 프로그래밍

GPU에는 CPU와 유사하게 코어(core), 스케줄러(scheduler), 레지스터(register), 캐시 메모리(cache memory) 등과 같은 자원들이 한정되어있다. 일반적인 CPU 프로그래밍에서는 상기한 자원들에 대한 직접적인 접근이 금지되어 있지만 CUDA에서는 프로그래머에게 스레드 체계와 메모리 체계를 노출함으로써 GPU의 자원들을 보다 더 잘 활용할 수 있도록 높은 자유도를 보장한다. [7]

3.1.1 스레드 체계

CUDA에서는 효율적인 자원 분배와 명령 수행을 위해 스레드들을 체계적으로 조직한다. 그림 3.2처럼 스레드들이 모여서 하나의 블록(block)이 되고, 블록들이 모여서 하나의 그리드(grid)가 된다. 여기서 커널(kernel)은 GPU 계산을 위해 데이터 병렬화가 적용된 함수를 뜻한다. 그리드와 블록은 최고 3차원까지 생성이 가능하며 블록 하나에 최대 1,024개의 스레드를 할당할 수 있다. 커널 호출 시 입력값으로 그리드와 블록을 넣으면 그에 맞게 GPU에 스레드가 할당되어 커널이 작동한다. 커널이 실행되면 스레드들은 CUDA 라이브러리의 built-in 변수를 이용해 각각 인덱스를 계산하고 인덱스에 따라 서로 다른 데이터에 접근하여 작업을 수행함으로써 데이터 병렬화가 이루어진다.

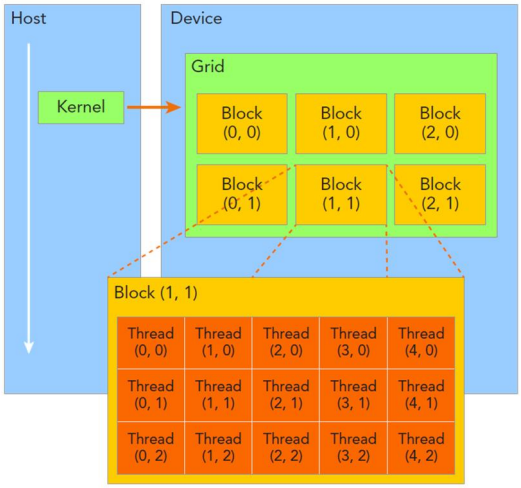


그림 3.2 그리드와 블록

2차원 혹은 3차원으로 스레드를 조직해도 실제로 GPU 상에서는 스레드가 항상 1차원으로 존재하는데 앞에서부터 32개씩 스레드들이 하나의 워프(warp)로 묶여 워프 스케줄러로부터 명령을 받아 수행하게 된다. 이러한 이유로 인해 블록 안의 스레드 개수가 32의 배수가 되게 하는 것이 좋다. 워프 스케줄러는 순서에 상관없이 가용한 워프부터 먼저 명령을 내리고 그 워프가 메모리 지연으로 인해 스톨에 걸리면 다시 다른 가용한 워프에 명령을 내림으로써 메모리 지연을 숨긴다.

기본적으로 스레드들은 커널 안의 명령문을 모두 똑같이 수행하지만 만일 조건문이 있다면 분기에 따라서 여러 번 명령을 수행하게 된다. 예를 들어 조건문에서 A 분기와 B 분기로 나뉘게 된다면 워프 안에서 먼저 A 분기에 해당되는 스레드들만 활성화하여 A 분기에 대한 명령을 수행한 다음 마찬가지로 B 분기에 해당되는 스레드들만 활성화하여 B 분기에 대한 명령을 수행한다. 조건문에서는 워프에 비효율적으로 여러

번 명령을 내리게 되는데 이를 워프 발산(warp divergence)라고 한다. 따라서 효율적인 계산을 위해서는 커널 안에 조건문을 최소화 해야한다.

본 연구에서는 이러한 사항들을 CUDA 코드에 잘 반영하였으며 최적의 스레드 구성을 실험적으로 구하여 적용하였다. [7, 12]

3.1.2 메모리 체계

GPU의 메모리는 효율적인 접근을 위해 많은 단계로 체계화 되어있다. GPU의 메모리 체계는 전역 메모리(global memory), 상수 메모리(constant memory), 텍스처 메모리(texture memory), 캐시 메모리(cache memory), 공유 메모리(shared memory) 그리고 레지스터(register)로 구성되어 있다. 전역 메모리는 물리적으로 DRAM(Dynamic random-access memory)에 위치해있으며 주로 CPU로부터 전달 받은 데이터를 저장하는데 사용된다. 전역 메모리는 크기가 크지만 캐시 메모리나 레지스터에 비해 지연이 크다. 전역 메모리의 대역폭(bandwidth)을 잘 활용하기 위해서는 통합적 메모리 접근(coalesced memory access)가 필요한데 이는 접근해야하는 데이터가 서로 인접해 있어야함을 뜻한다. 전역 메모리를 사용하기 위해서는 먼저 공간을 할당한 다음 PCIe를 통해 CPU로부터 데이터를 전달 받아야 한다. PCIe의 대역폭은 매우 낮기 때문에 CPU와 GPU 간의 통신을 최소화해야 한다. 상수 메모리와 텍스처 메모리는 각각 상수를 저장하고 그래픽 작업에서 텍스처를 저장할 때 사용하는 메모리이다. 텍스처

메모리는 Kepler 아키텍처에서 읽기 작업이 캐시된다는 특성 때문에 전역 메모리 대신 사용되곤 했으나 최신 아키텍처에서는 전역 메모리도 읽기 작업이 캐시되기 때문에 잘 사용되지 않는다.

GPU의 캐시 메모리는 CPU의 캐시 메모리와 유사하게 L1 캐시와 L2 캐시의 두 단계로 나뉘어 있다. CPU와 마찬가지로 L1 캐시는 L2 캐시보다 크기가 작지만 메모리 지연이 낮아 대역폭이 더 크다. 앞서 언급했듯이 최신 아키텍처에서는 전역 메모리가 캐시되는데 GPU에서는 오직 읽기 작업만이 L2 캐시에 캐시된다는 특징이 있다. 캐시 메모리는 스레드 체계에 대응하도록 할당되는데 그림 3.3과 같이 L2 캐시는 그리드 기준으로, L1 캐시는 블록 기준으로 할당된다. L1 캐시와 L2 캐시는 사용자가 제어할 수는 없지만 CUDA에는 공유 메모리라는 개념이 있다. 공유 메모리는 일종의 제어 가능한 캐시로 물리적으로 L1 캐시와 같은 공간을 사용하기 때문에 공유 메모리를 많이 할당하면 사용할 수 있는 L1 캐시의 크기가 줄어든다. 공유 메모리도 L1 캐시와 마찬가지로 블록 기준으로 할당되는데 주로 각 스레드들에게 공통적으로 필요한 데이터를 저장하여 불필요한 전역 메모리 접근을 줄이거나 스레드 간의 협력에 의한 중간 결과를 저장하는데 쓰인다.

레지스터는 크기는 제일 작지만 지연이 제일 작은 메모리로 캐시 메모리와 함께 물리적으로 GPU 칩에 존재하기 때문에 대역폭이 매우 높다. 레지스터는 코어의 산술 논리 장치(Arithmetic logic unit, ALU)의 입력과 출력을 저장하는데 사용되며 각 스레드가 자신만의 레지스터를 보유한다. 스레드 하나에 여러 개의 레지스터를 할당할 수 있는데 만일

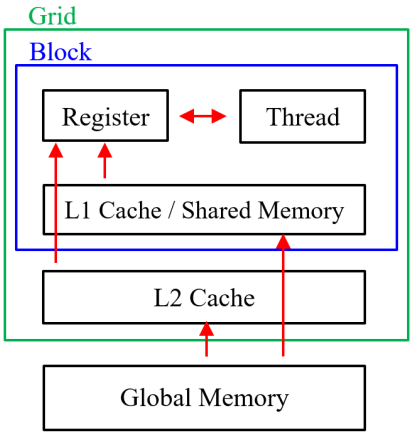


그림 3.3 메모리 접근 체계

GPU의 한계치를 초과하게 되면 레지스터 대신 지역 메모리(local memory)를 사용하며 이를 레지스터 넘침(register spill)이라고 한다. 아키텍처에 따라서 전역 메모리 혹은 캐시 메모리가 지역 메모리로 사용되며 메모리 지연이 레지스터보다 훨씬 커지기 때문에 레지스터 넘침을 적절히 억제하는 것이 중요하다. [7, 12]

GPU에서는 많은 연산들은 대역폭이 성능에 지배적으로 작용하는 memory-bound 연산이다. 코어의 연산 성능이 지배적인 compute-bound 연산에서도 대역폭이 낮아지면 메모리 작업이 병목의 원인이 되어 성능이 하락하므로 GPU 계산에서 높은 대역폭을 유지하는 것이 매우 중요하다고 할 수 있다. GPU의 메모리 체계는 그림 3.4와 같이 요약할 수 있는데 본 연구에서는 효율적인 GPU 계산을 위해 대역폭이 큰 메모리를 최대한 활용하도록 CUDA 코드를 개발하였다.

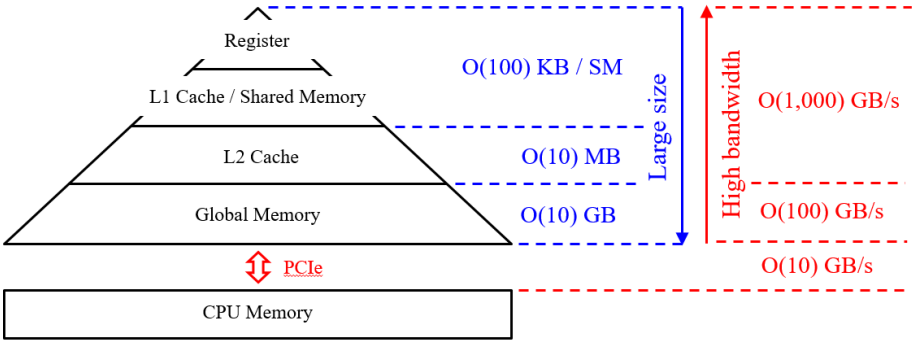


그림 3.4 GPU 메모리 체계

3.2 텐서 축약(TC)

불연속 갤러킨 방법에서는 플럭스 계산을 제외한 대부분의 계산을 텐서 축약 연산으로 구할 수 있다. 불연속 갤러킨 방법의 텐서 축약은 그림 3.5와 같이 3가지 종류가 있다. 그림 3.5에서 TC는 텐서 축약(Tensor contraction, TC)을 뜻하며 텐서 축약 식은 합 기호를 생략하고 합 인덱스를 붉은 색으로 표시하였다. 불연속 갤러킨 방법의 텐서 축약 연산은 차수가 높지 않기 때문에 행렬 곱 커널을 이용해 효율적으로 계산할 수 있으며 텐서 축약 연산의 크기가 $\mathcal{O}(100)$ 을 초과하지 않지만 계산해야 할 텐서 축약 연산의 수가 매우 많다는 특징이 있다. [2]

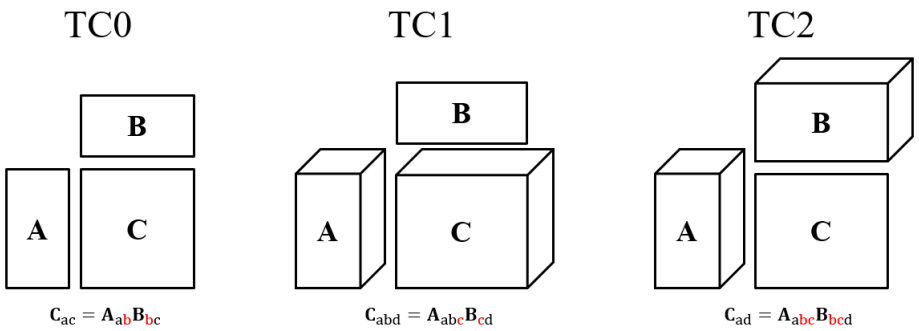


그림 3.5 불연속 갤러킨 방법의 텐서 축약

3.3 CUDA 커널

GPU에서 불연속 갤러킨 방법을 계산하는 CUDA 커널은 텐서 축약 커널, 플럭스 계산 커널, 시간 적분 커널로 구성되어있다. 본 절에서는 효율적인 GPU 계산을 위한 커널 최적화에 대해서 다룰 것이다.

3.3.1 GEMM 커널

GEMM(General matrix multiplication) 커널은 행렬 곱 만을 위해 최적화된 함수로 일반적으로 low-level까지 튜닝이 많이 되었기 때문에 성능이 매우 높다. 불연속 갤러킨 방법의 텐서 축약 연산은 차수가 높지 않기 때문에 GEMM 커널을 이용한 텐서 축약 커널로 효율적으로 계산할 수 있다.

대표적인 GEMM 커널로 CPU 상에서 Intel사의 MKL (Math kernel library)가 있으며, GPU 상에서는 Nvidia사의 cuBLAS(Cuda basic linear algebra subprogram)이 있다. 이 중 cuBLAS 라이브러리의 GEMM 커널은 $\mathcal{O}(1{,}000)$ 이상의 큰 행렬에 대해서만 높은 성능을 보인다고 알려져 있다. 불연속 갤러킨 방법의 텐서 축약 연산은 $\mathcal{O}(100)$을 초과하지 않기 때문에 본 연구에서는 텐서 축약 연산의 효율적인 계산을 위해 별도의 GEMM 커널을 제작하였다. 두 종류의 아키텍처에서 정사각 행렬 곱에 대한 성능을 측정해본 결과, 그림 3.6과 같이 작은 행렬에 대해서 캐시 메모리를 이용한 별도의 GEMM 커널이 cuBLAS 라이브러리보다 약 15~70% 정도 성능이 높았다.

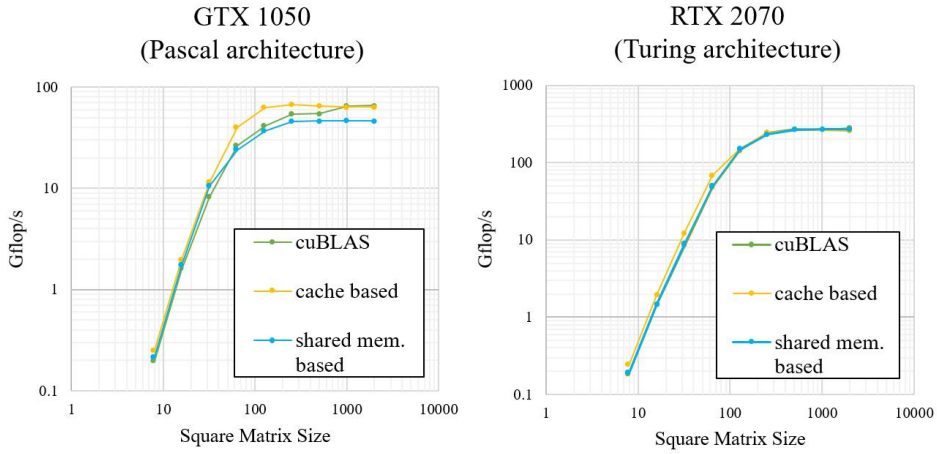


그림 3.6 GEMM 커널 정사각 행렬 곱 성능(Gflop/s)

캐시 메모리 기반의 GEMM 커널이 작은 행렬에 대해서 cuBLAS 라이브러리보다 성능이 높은 것은 최신 아키텍처의 용량이 큰 캐시

메모리로 인해 대역폭이 더 높았기 때문인 것으로 분석된다. GEMM 커널들의 캐시 hit ratio와 대역폭을 아래 표에 나타내었다.

GPU 종류	GTX 1050		RTX 2070	
커널 종류	cuBLAS	Cache based	cuBLAS	Cache based
메모리 대역폭	2.03GB/s	6.20GB/s	2.52GB/s	9.38GB/s
L2 캐시 hit ratio	83.3%	92.3%	83.05%	85.78%
L1 캐시 hit ratio	–	48.0%	–	61.54%

표 3.1 GEMM 커널 메모리 대역폭(GB/s)

3.3.2 TC 커널

본 연구에서는 불연속 갤러킨 방법의 TC 연산이 차수가 높지 않기 때문에 GEMM 커널을 기반으로 하는 TC 커널을 제작하였다. 그림 3.5의 TC0는 행렬 곱과 같기 때문에 GEMM 커널을 한 번 호출하는 것으로 계산할 수 있다. 그림 3.5에서 TC1은 합 인덱스가 1개인 3차 텐서와 2차 텐서 간의 TC인데 3차 텐서는 2차 텐서의 묶음으로 볼 수 있다. 따라서 TC1도 그림 3.7처럼 행렬 곱의 묶음으로 볼 수 있다. 이를 행렬 곱을 여러 번 호출함으로써 계산할 수 있지만 각각의 행렬 곱 연산에서 불필요한 전역 메모리 접근이 일어나게 되므로 비효율적이다. 행렬 곱을 여러 번 호출하는 대신 행렬 곱 연산들을 하나의 연산으로 합치면 불필요한 전역 메모리 접근을 없앨 수 있으며 연산의 크기가 커지기 때문에 성능이 높아진다. 이는 GPU가 일반적으로 연산의 크기가 커질수록 성능

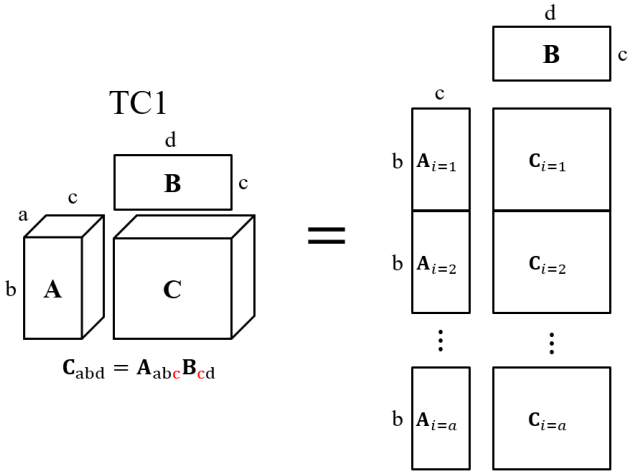


그림 3.7 GEMM 커널을 이용한 TC1 계산

이 높아지기 때문이다. 그림 3.5에서 TC2는 합 인덱스가 2개인 3차 텐서 간의 TC이다. TC2는 행렬 곱 연산을 누적해야하기 때문에 TC1처럼 행렬 곱 연산들을 하나의 연산으로 합칠 수는 없지만 행렬 곱 커널을 여러 번 호출하는 대신 하나의 커널 안에서 행렬 곱 연산을 누적함으로써 전역 메모리 접근을 최적화하였다.

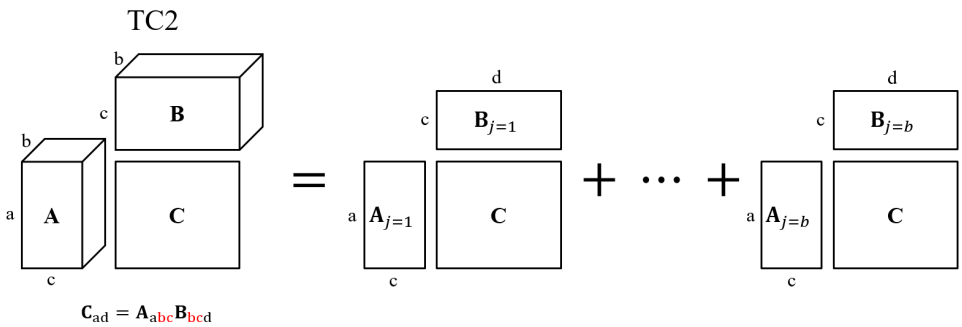


그림 3.8 GEMM 커널을 이용한 TC2 계산

3.3.4 플럭스 계산 커널

CPU 코드에서는 quadrature point에 대해서 순차적으로 플럭스를 계산한다. GPU에서는 미세 그레인 병렬화를 적용하여 스레드 하나가 하나의 quadrature point에 대한 플럭스를 계산하므로 모든 quadrature point에 대한 플럭스가 동시에 계산된다. 플럭스를 계산한 후에는 이를 RHS에 저장해야 하는데 식 (2.10)에서 면적분은 face 기준으로 되어 있고 부피적분은 cell 기준으로 되어있다. 그런데 RHS는 cell 기준으로 저장되므로 face에서의 RHS는 경합 조건(race condition)을 피하기 위해 순차적으로 저장되고 cell에서의 RHS는 경합 조건이 없으므로 동시적으로 저장된다.

3.3.4 시간 적분 커널

본 연구에서 외재적 시간 기법으로 사용한 SSPRK45 기법은 계산적으로 여러 배열(array)들의 합이다. 이는 GPU에서 벡터 합 커널을 이용해 계산할 수 있는데 cuBLAS 라이브러리에서 제공하는 벡터 합 커널은 입력 받을 수 있는 벡터의 수가 제한적이다. cuBLAS 라이브러리의 벡터 합 커널을 여러 번 호출하면 해가 저장되는 배열에 대해 불필요한 전역 메모리 접근이 일어나게 된다. 따라서 본 연구에서는 여러 개의 배열을 입력 받을 수 있는 시간 적분 커널을 제작하여 전역 메모리 접근을 최적화하였다.

내재적 시간 기법으로 사용한 RODAS5 기법에서는 시간 전진을 위해 선형 시스템을 풀어야 하는데 본 연구에서는 MAGMA(Matrix algebra on GPU and multicore architecture) 라이브러리의 GMRES (Generalized minimal residual) 방법과 Jacobi preconditioner를 사용하였다. 시스템 행렬(system matrix)가 차지하는 GPU 메모리를 줄이기 위해 CSR(Compressed sparse row) 포맷으로 저장하였다. [16]

3.4 CUDA Stream

CUDA 라이브러리에서는 연산이 충분히 크지 못해 GPU의 모든 코어들을 활용하지 못할 경우, CUDA Stream이라는 기능을 통해 그림 3.9와같이 여러 개의 커널을 동시적으로 실행시킬 수 있다. CUDA Stream 기능을 이용하면 아키텍처에 따라 최대 32개 혹은 128개의 커널을 동시적으로 실행할 수 있는데 GPU의 자원이 모자랄 경우, 커널의 일부분만이 실행된다.

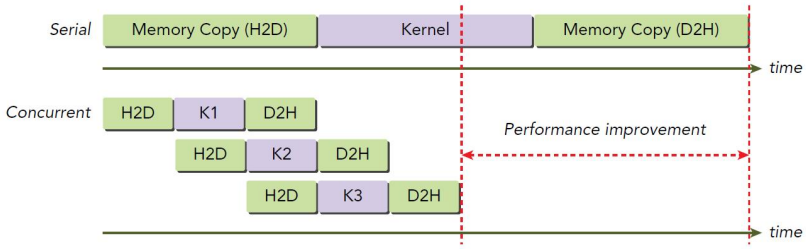


그림 3.9 CUDA Stream 개념도

CUDA Stream을 이용하면 계산 성능을 개선할 수 있다는 장점 외에도 각각의 커널들이 서로 독립적이기 때문에 문제 없이 혼합 격자를 다룰 수 있다는 장점이 있다. 따라서 본 연구에서는 효율적인 GPU 계산을 위해 불연속 갤러킨 방법에 CUDA Stream 기능을 적용하였다.

CUDA Stream이 적용된 전체적인 코드 알고리즘은 그림 3.10과 같다. 먼저 전처리 과정에서는 CPU 상에서 GPU 계산에 필요한 데이터들을 계산한 다음, GPU에 메모리를 할당하고 CPU와 GPU 간의 통신을 최소화하기 위해 모든 데이터를 GPU로 전달한다. 그 다음, GPU 상에서 CUDA Stream을 이용해 동시적으로 계산 커널들을 실행하고 시간 적분 커널로 해를 갱신(update)한다. 이후 GPU에서 해를 CPU로 전달해 CPU 상에서 시간 간격(time step)을 계산한다. CPU 상에서 시간 간격을 계산하는 이유는 시간 간격을 계산할 때 배열에서 최솟값을 찾는 과정이 있는데 이것이 GPU 상에서 매우 비효율적이기 때문이다. 시간 간격을 계산한 후에는 GPU에서 다시 계산을 시작해 시뮬레이션 종료 조건이 만족될 때까지 이 과정을 반복한다.

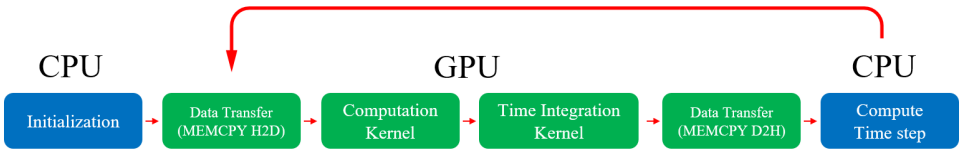


그림 3.10 전체적인 코드 알고리즘

그림 3.10의 계산 커널은 그림 3.11처럼 TC 커널과 플럭스 계산 커널, RHS를 저장하는 커널로 구성되어 있다. 본 연구에서는 계산 커널을 32개의 스트림(stream)으로 실행하였다.

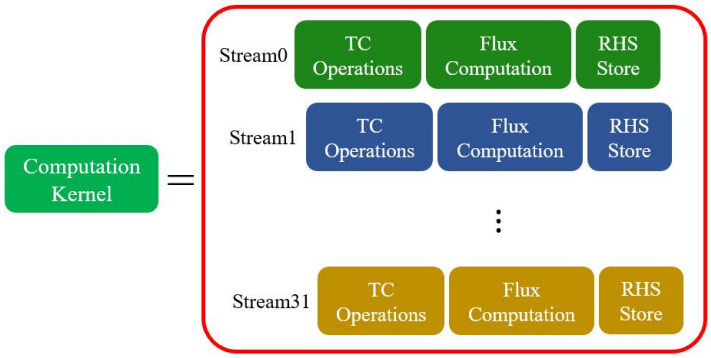


그림 3.11 계산 커널 구성

제 4 장 수치 실험

4.1 실험 환경

4.1.1 프로세서

본 연구의 수치 실험에 사용한 프로세서의 종류와 제원을 표 4.1에 나타내었다. 수치 실험은 CPU에서 코어 1개와 4개를 이용해 각각 순차 계산과 병렬 계산을 수행하였으며, GPU에서는 단일 GPU 상에서 연산의 크기에 맞게 유동적으로 스레드를 배치하고 32개의 CUDA Stream을 이용하여 병렬 계산을 수행하였다.

프로세서	i7-6700	GTX 1050	RTX 2070
프로세서 종류	CPU	GPU	GPU
마이크로아키텍처	Skylake	Pascal	Turing
코어 수	4	640	2,304
기본 클럭	3.4GHz	1.35GHz	1.41GHz
배정밀도 부동소수점 이론적 성능	54.4Gflop/s(1core) 217.6Gflop/s(4core)	58.2Gflop/s	233.3Gflop/s
메모리 종류	DDR3	GDDR5	GDDR6
메모리 크기	16GB	2GB	8GB

표 4.1 수치 실험을 수행한 프로세서 제원

4.1.2 라이브러리

본 연구에서는 리눅스 계열 OS인 Ubuntu 16.04에서 수치 실험을 진행하였으며 사용한 라이브러리는 다음과 같다.

CPU 라이브러리로 병렬 계산을 위한 Intel MPI 2018.5.274, 텐서 축약 계산을 위한 Intel MKL 2018.5.274, 그리고 선형 시스템 솔버로 PETSc(Portable, extensible toolkit for scientific computation) 3.10.5를 사용하였다. [13, 14]

GPU 라이브러리로 GPU 병렬 계산을 위한 CUDA 10.2, 그래픽 드라이버로 Nvidia 그래픽 드라이버 440.31 그리고 선형 시스템 솔버로 MAGMA(Matrix algebra on GPU and multicore architecture) 2.5.2를 사용하였다. [15, 16]

4.2 실험 방법

4.2.1 해석 대상 및 조건

본 연구에서는 성능 측정을 위해 무차원화된 Navier-Stokes 방정식을 지배방정식으로 하는 3차원 점성 사인파(sinewave) 문제를 해석하였다. 계산 영역은 $[0,1] \times [0,1] \times [0,1]$ 이며 초기 조건은 식 (4.1)과 같다.

$$
\begin{aligned}
&\rho_{t_0}(\mathrm{x},\mathrm{y},\mathrm{z}) = 1.0 + 0.2\sin(2\pi\mathrm{x})\sin(2\pi y)\sin(2\pi z)\\
&u_{t_0}(\mathrm{x},\mathrm{y},\mathrm{z}) = 1.0\\
&v_{t_0}(\mathrm{x},\mathrm{y},\mathrm{z}) = 0.5\\
&w_{t_0}(\mathrm{x},\mathrm{y},\mathrm{z}) = 0.5\\
&p_{t_0}(\mathrm{x},\mathrm{y},\mathrm{z}) = 1.0
\end{aligned}
\tag{4.1}
$$

유동 조건은 마하 수가 0.1, 레이놀즈 수가 200, 프란틀 수가 0.72이며 초기 조건의 밀도 contour를 그림 4.1에 나타내었다. 수치 플럭스 기법은 local Lax-Friedrichs 기법을 사용하였으며 외재적 시간 기법으로 SSPRK45 기법을, 내재적 시간 기법으로 RODAS5 기법을 사용하여 CFL 수 0.9로 계산을 20번 반복하는데 소요된 계산 시간을 측정하였다.

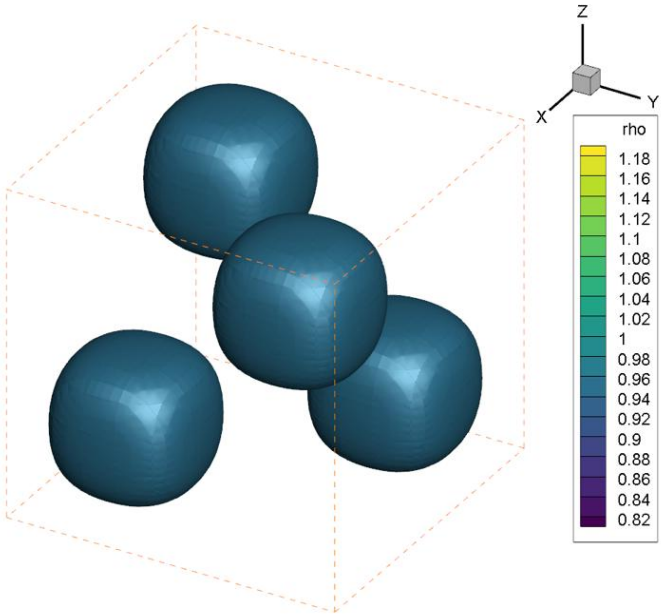


그림 4.1 점성 사인파 문제 초기 조건의 밀도 contour ($\rho = 0.95$)

4.2.2 격자

본 연구에서는 수치 실험을 위해 4가지 격자를 사용하였는데 이를 표 4.2에 나타내었다. 표 4.2에서 Grid#1과 Grid#2는 외재적 시간 기법으로 수치 실험을 수행하는데 사용하였고, Grid#3과 Grid#4는 내재적 시간 기법으로 수치 실험을 수행하는 사용하였다. 내재적 시간 기법에서는 외재적 시간 기법보다 메모리가 더 많이 필요하므로 격자 수가 적다. Grid#1과 Grid#3은 육면체(hexahedral) 요소로만 이루어져있으며 Grid#2와 Grid#4는 사면체(tetrahedral) 요소와 사각뿔(pyramid) 요소로 이루어진 혼합 격자이다.

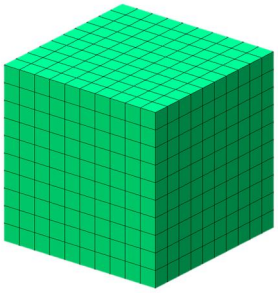
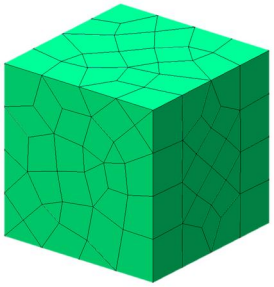
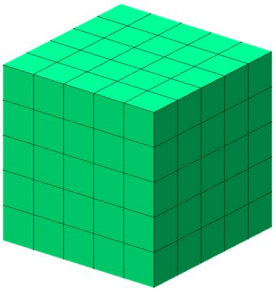
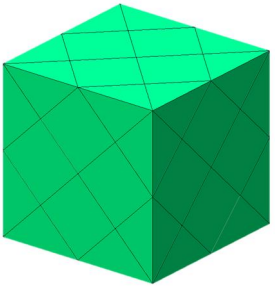
격자 종류	격자	격자 요소
Grid#1		육면체 요소 1,000개
Grid#2		사면체 요소 1,237개 사각뿔 요소 134개
Grid#3		육면체 요소 125개
Grid#4		사면체 요소 234개 사각뿔 요소 24개

표 4.2 수치 실험에 사용한 격자 종류

4.3 실험 결과

본 절에서는 수치 실험 결과를 토대로 CPU와 GPU의 시간 비용과 계산 성능(Gflop/s)를 비교, 분석하였다. 또한, 계산 커널을 구성하는 각각의 커널에서 소요한 시간 비용과 연산의 크기가 계산 성능에 미치는 영향에 대해 분석하였다.

4.3.1 외재적 시간 기법

육면체 요소로만 구성된 격자인 Grid#1에서 소요한 시간 비용을 그림 4.2에, 달성한 계산 성능을 그림 4.3에 나타내었다. $p5$에서 GTX 1050은 메모리 부족으로 인해 계산을 수행하지 못하였다. 그림 4.2에서 차수가 증가함에 따라 CPU는 시간 비용이 급격하게 증가하지만 GPU는 시간 비용이 상대적으로 완만하게 증가하는 것을 볼 수 있다. 차수가 작을 때는($p < 5$) CPU가 GPU보다 시간 비용이 적지만 차수가 증가하면서 CPU의 계산 비용이 급격히 증가해 초고차에서 CPU의 시간 비용이 GPU의 시간 비용보다 커졌다. 이는 차수가 커질수록 텐서 축약 연산의 크기가 커져서 GPU에 유리하게 작용했기 때문인 것으로 분석된다. 그림 4.3에서 막대 그래프는 달성한 계산 성능(Gflop/s)이고 꺾은선 그래프는 달성한 계산 성능을 프로세서의 이론적 계산 성능으로 나눈 것을 백분율로 나타낸 것이다. 그림 4.3에서 CPU는 낮은 차수에서도 이론적 성능 대비 달성한 성능이 15%를 넘지만 GPU는 낮은 차수에서

이론적 성능 대비 달성한 성능이 상대적으로 낮다. 이는 앞서 언급했듯이 낮은 차수에서 텐서 축약 연산의 크기가 충분히 크지 않지 않아서 GPU의 성능을 많이 활용하지 못했기 때문인 것으로 분석된다. 차수가 증가하며 텐서 축약 연산의 크기가 커지자 GPU에서 달성한 계산 성능이 급격히 증가하는 것을 확인할 수 있다. 더 심층적인 분석을 위해 각각의 계산 커널들에서 소요한 시간 비용과 차수에 따른 연산의 크기를 각각 그림 4.4와 표 4.3에 나타내었다. 외재적 시간 기법에서 계산 커널은 face에서 RHS를 계산하는 커널과 cell에서 RHS를 계산하는 커널로 구성되어있다. 그림 4.4에서 CPU는 전체 시간 비용에서 face와 cell이 차지하는 비율이 서로 비슷하지만 GPU에서는 face가 차지하는 비율이 cell이 차지하는 비율보다 더 큰 것을 확인할 수 있다. 이는 크게 2가지 이유 때문인 것으로 분석된다. 첫번째 이유는 cell에서의 텐서 축약 연산이 face보다 더 효율적으로 계산되기 때문이다. 표 4.3의 (a)는 요소 하나에 대한 평균적인 연산의 크기를, (b)는 전체 요소 개수를 고려한 전체 연산의 크기를 나타낸 것인데 평균적인 연산의 크기는 cell이 face보다 훨씬 크지만 전체 연산의 크기는 비교적 비슷한 것을 확인할 수 있다. 따라서 cell에 대한 텐서 축약 연산이 크기가 face보다 더 크기 때문에 GPU에 유리하게 작용해 더 효율적으로 계산되었다고 할 수 있다. GPU에서 face의 계산 비용이 cell의 계산 비용보다 큰 두번째 이유는 cell에서 RHS를 더 효율적으로 저장하기 때문이다. 3장의 3.3절에 언급했듯이 cell에서는 경합 조건이 없어서 RHS를 동시적으로 저장하지만 face에서는 경합 조건을 피하기 위해 RHS를 순차적으로 저장하여 상대적으로 비효율적이다.

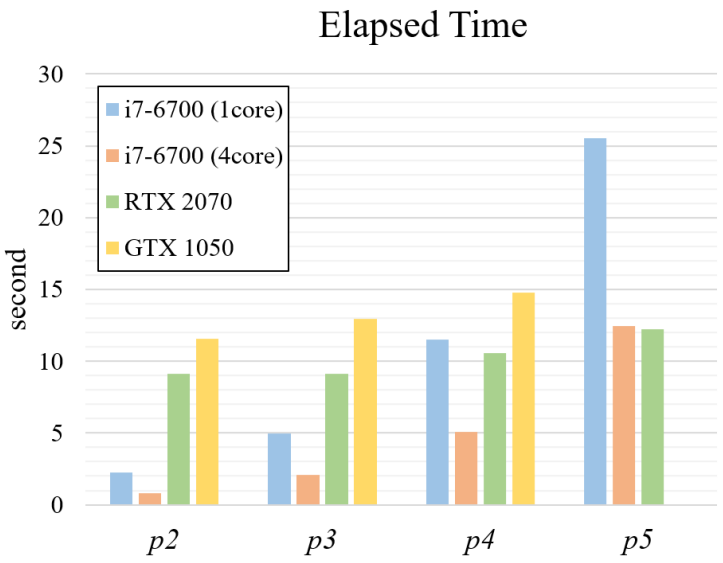


그림 4.2 외재적 시간 기법 시간 비용 비교 (Grid#1)

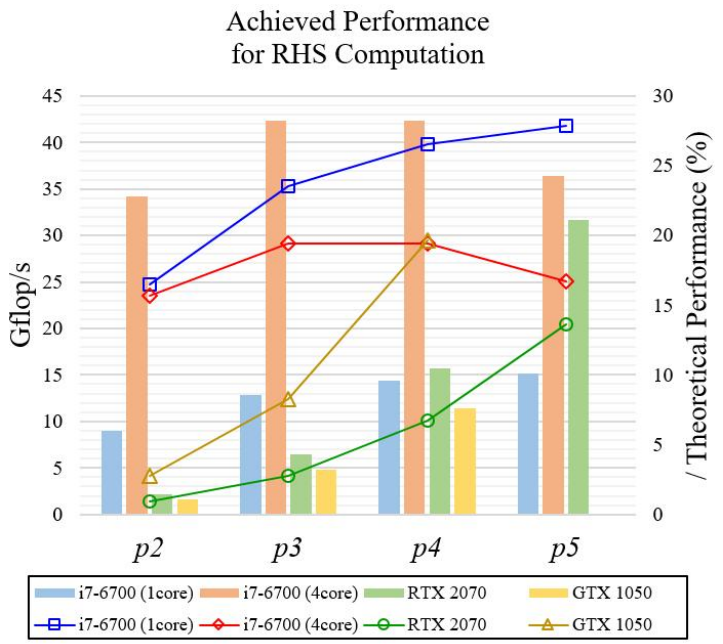


그림 4.3 RHS 계산에서 달성한 계산 성능 비교 (Grid#1)

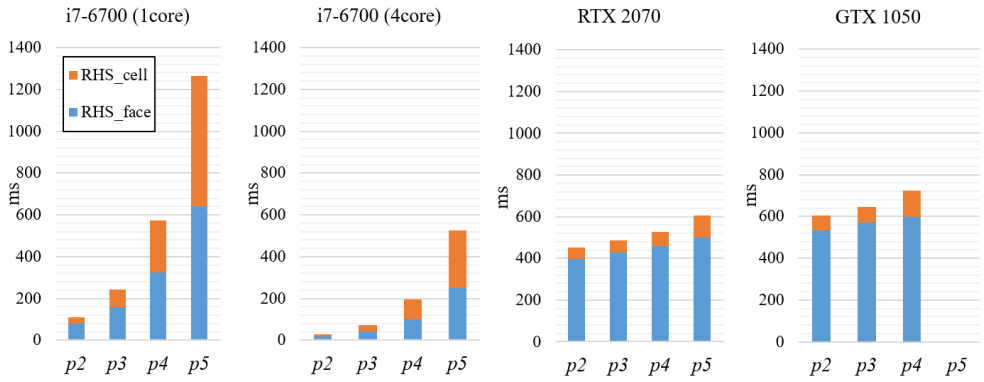


그림 4.4 RHS 계산 커널들의 시간 비용 비교 (Grid#1)

차수	*p*2	*p*3	*p*4	*p*5
face	3.11	9.1	22.25	47.71
cell	14.68	53.66	157.55	394.24

(a) 요소 별 평균 연산 크기(Kflops/TC kernel)

차수	*p*2	*p*3	*p*4	*p*5
face	0.12	0.35	0.87	1.86
cell	0.07	0.27	0.79	1.97

(b) 전체 연산 크기(Gflops)

표 4.3 외재적 시간 기법 연산 크기 (Grid#1)

사면체 요소와 사각뿔 요소로 이루어진 혼합 격자 Grid#2에 대한 수치 실험 결과를 그림 4.5, 그림 4.6, 그림 4.7 그리고 표 4.4에 나타내었다. Grid#1과 마찬가지로 *p*5에서 GTX 1050은 메모리 부족으로 인해 계산을 수행하지 못하였다. 전체적인 경향은 Grid#1과 유사하지만 GPU의 성능이 조금 하락하였다. 이는 Grid#2의 대부분이 사면체 요소로 이루어져 있는데 육면체 요소보다 quadrature point 개수가 적어서 텐서 축약 연산의 크기가 줄어들었기 때문인 것으로 분석된다. 표 4.4에서 Grid#1에 비해 줄어든 연산의 크기가 이를 뒷받침한다.

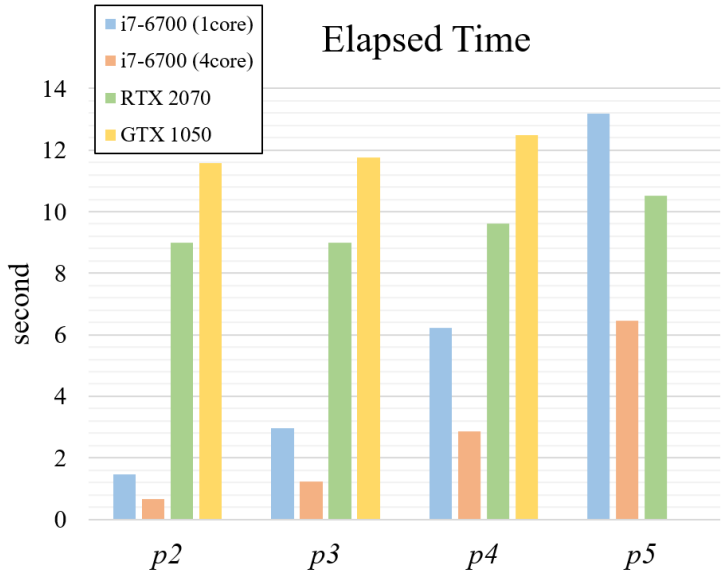


그림 4.5 외재적 시간 기법 시간 비용 비교 (Grid#2)

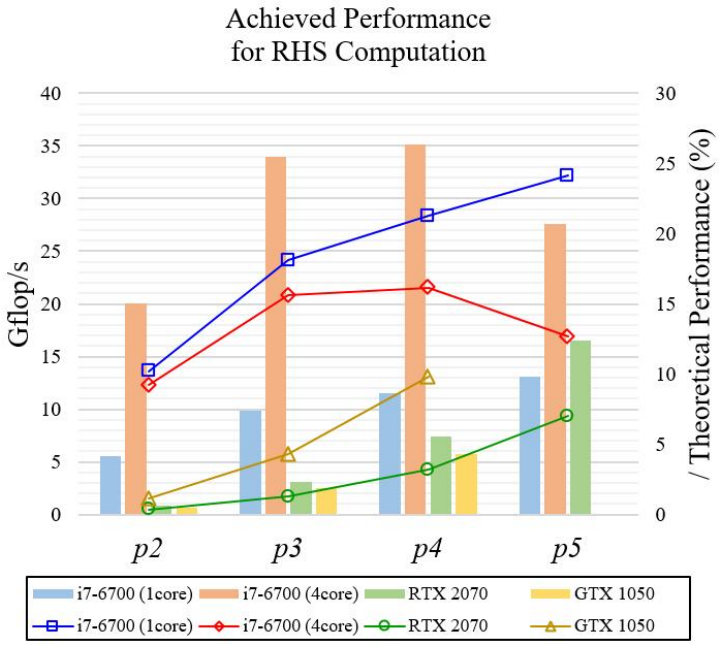


그림 4.6 RHS 계산에서 달성한 계산 성능 비교 (Grid#2)

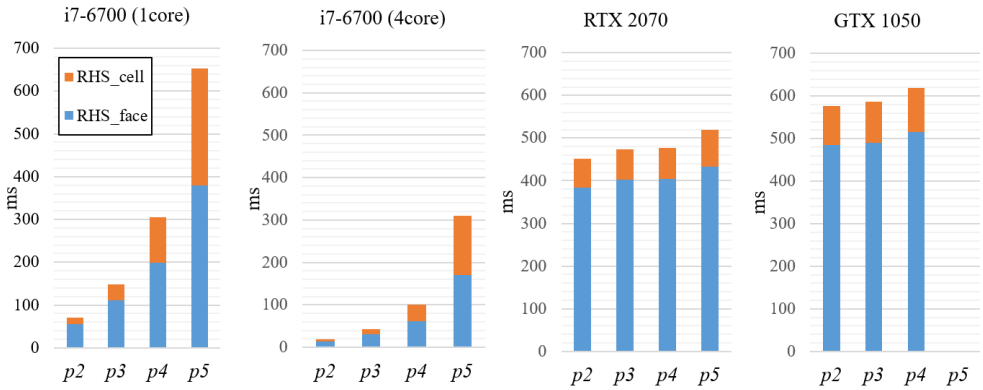


그림 4.7 RHS 계산 커널들의 시간 비용 비교 (Grid#2)

차수	$p2$	$p3$	$p4$	$p5$
face	1.45	5.64	12.26	28.26
cell	4.33	14.54	45.67	122.53

(a) 요소 별 평균 연산 크기(Kflops/TC kernel)

차수	$p2$	$p3$	$p4$	$p5$
face	0.06	0.22	0.48	1.10
cell	0.02	0.07	0.23	0.61

(b) 전체 연산 크기(Gflops)

표 4.4 외재적 시간 기법 연산 크기 (Grid#2)

4.3.2 내재적 시간 기법

육면체 요소 125개로 구성된 격자인 Grid#3에서 소요한 시간 비용과 달성한 계산 성능을 각각 그림 4.8와 4.9에 나타내었다. 그림 4.2는 시간 비용을 로그 스케일로 나타낸 것인데 CPU와 GPU 모두 차수가 증가함에 따라 시간 비용이 급격히 증가하는 것을 확인할 수 있다. 아래에서 더 서술하겠지만 GPU에서 선형 시스템을 푸는데 상대적으로 오래 걸렸기 때문에 GPU가 CPU보다 시간 비용이 크다. 그림 4.9는 시스템 행렬을 계산하는데 달성한 계산 성능을 나타낸 것인데 막대 그래프는 달성한 계산 성능(Gflop/s)이고 꺾은선 그래프는 이론적 성능 대비 달성한 계산 성능을 백분율로 표현한 것이다. 외재적 시간 기법과 비교했을 때 차수가 낮을 때도 GPU의 성능이 상대적으로 높은 것을 확인할 수 있다. 이는 시스템 행렬 계산을 위한 텐서 축약 연산의 크기가 매우 크기 때문이다. 차수가 증가할 때는 외재적 시간 기법과 유사하게 GPU의 계산 성능이 급격히 증가하여 $p5$에서 GPU의 계산 성능이 CPU의 계산 성능의 약 2배가 되었다. 외재적 시간 기법과 마찬가지로 차수가 증가하자 텐서 축약 연산의 크기가 커져서 GPU의 계산 성능이 높아진 것으로 분석된다. 그림 4.10에서 내재적 시간 기법의 계산 커널은 선형 시스템을 푸는 커널, face에서 블록 행렬을 계산하는 커널, face에서 블록 행렬을 시스템 행렬에 저장하는 커널, cell에서 블록 행렬을 계산하는 커널 그리고 cell에서 블록 행렬을 시스템 행렬에 저장하는 커널로 구성되어 있다. GPU에서 시스템 행렬의 계산과 저장은 CPU에 비해 매우 효율적이지만 MAGMA 라이브러리를 이용해 선형 시스템을 푸는 것이 CPU

에서 PETSc 라이브러리를 이용해 선형 시스템을 푸는 것보다 상대적으로 비효율적인 것을 확인할 수 있다. 한편, 외재적 시간 기법과는 달리 cell에서 시스템 행렬을 계산하는 것이 face에서 시스템 행렬을 푸는 것보다 시간 비용을 더 소요하는 것을 확인할 수 있는데 이는 cell에 대해서 시스템 행렬을 계산할 때 face처럼 인접한 cell들을 고려해야 하기 때문이다. 따라서 텐서 축약 연산 일부를 순차적으로 계산해야 하고 텐서 축약 연산의 수도 늘어나서 표 4.5에서 평균적인 연산 크기 뿐만 아니라 전체 연산 크기도 cell이 face보다 큰 것을 확인할 수 있다. 이러한 이유들로 인해 cell에서 시스템 행렬을 계산하는 것이 face보다 시간 비용을 더 많이 소요하는 것이다. 표 4.5에서 내재적 시간 기법의 연산의 크기가 외재적 시간 기법보다 매우 큰 것을 확인할 수 있는데 이로 인해 내재적 시간 기법에서 외재적 시간 기법보다 더 높은 계산 성능을 달성하였다.

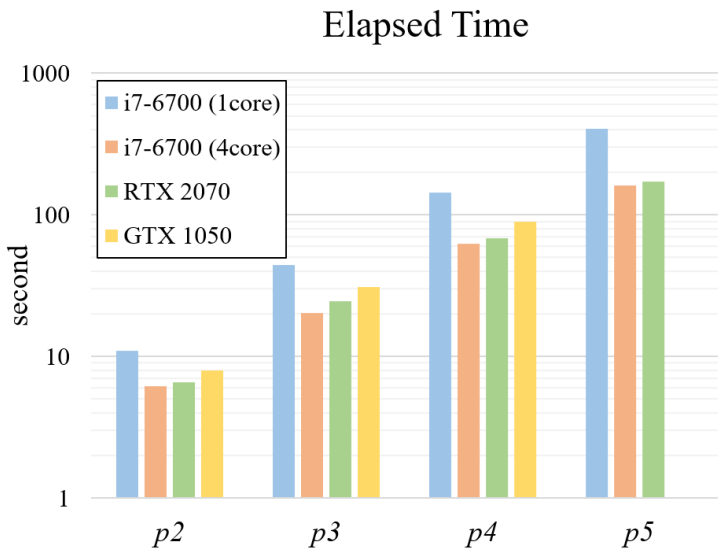


그림 4.8 내재적 시간 기법 시간 비용 비교 (Grid#3)

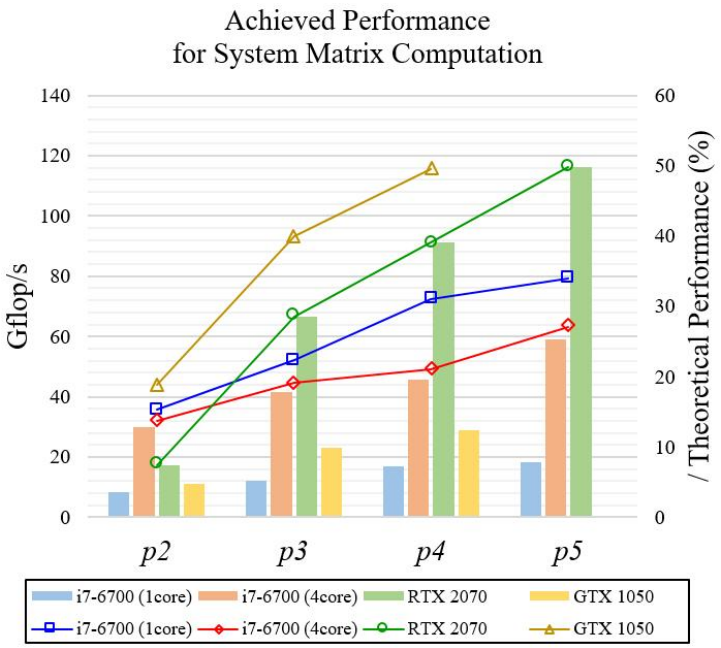


그림 4.9 시스템 행렬 계산에서 달성한 계산 성능 비교 (Grid#3)

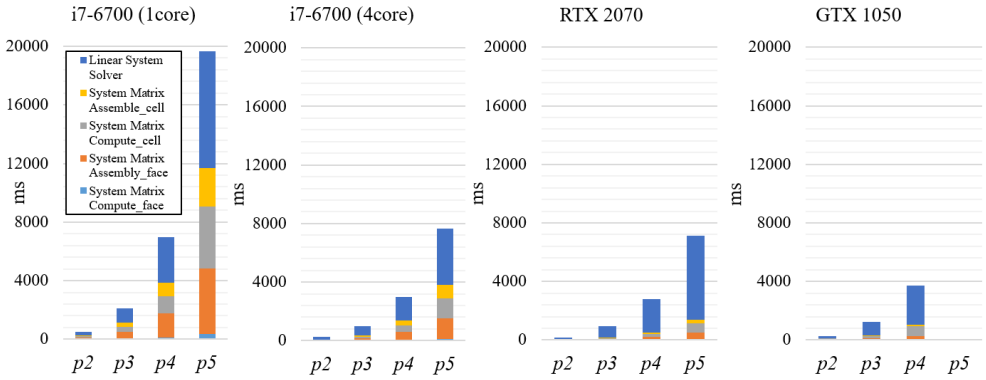


그림 4.10 내재적 시간 기법 계산 커널들의 시간 비용 비교 (Grid#3)

차수	*p*2	*p*3	*p*4	*p*5
face	0.03	0.13	0.53	1.75
cell	0.44	3.17	16.17	64.6

(a) 요소 별 평균 연산 크기(Mflops/TC kernel)

차수	*p*2	*p*3	*p*4	*p*5
face	0.19	0.94	3.77	12.50
cell	0.50	3.56	18.19	72.67

(b) 전체 연산 크기(Gflops)

표 4.5 내재적 시간 기법 연산 크기 (Grid#3)

사면체 요소와 사각뿔 요소로 이루어진 혼합 격자 Grid#4에 대한 수치 실험 결과를 그림 4.11, 그림 4.12, 그림 4.13 그리고 표 4.6에 나타내었다. Grid#4는 Grid#4보다 격자수가 많아 메모리가 더 많이 필요하기 때문에 GTX 1050은 $p2$에서만 계산을 수행하였다. 외재적 시간 기법과 유사하게 Grid#3과 비교하였을 때 전체적인 경향은 비슷하지만 GPU의 성능이 하락하였다. 이 역시 Grid#4가 대부분이 사면체 요소로 구성되어있기 때문에 quadrature point 개수가 육면체 요소보다 적어서 텐서 축약 연산의 크기가 감소했기 때문이다. 특히 표 4.6에서 cell에서 시스템 행렬 계산을 위한 텐서 축약 연산의 크기가 Grid#3에 비해 약 70% 정도 감소하였다. 한편, Grid#4 역시 Grid#3과 마찬가지로 시스템 행렬의 계산과 저장은 GPU가 CPU보다 더 효율적이지만 라이브러리를 이용해 선형 시스템을 푸는 것은 상대적으로 비효율적임을 확인할 수 있다.

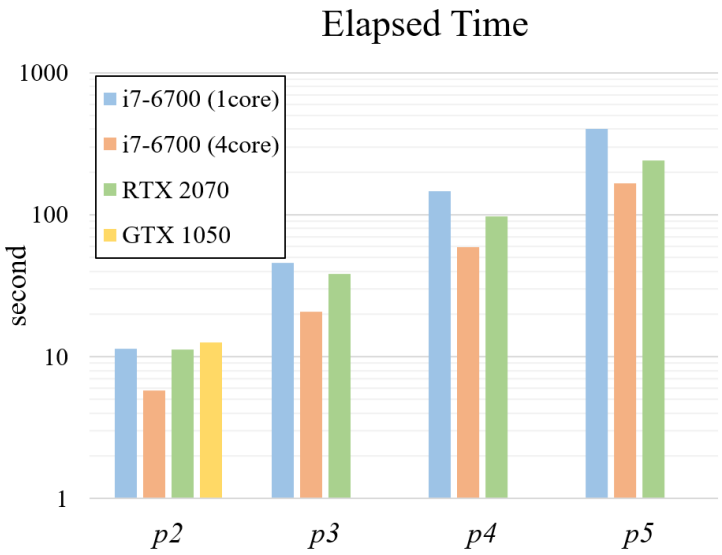


그림 4.11 내재적 시간 기법 시간 비용 비교 (Grid#4)

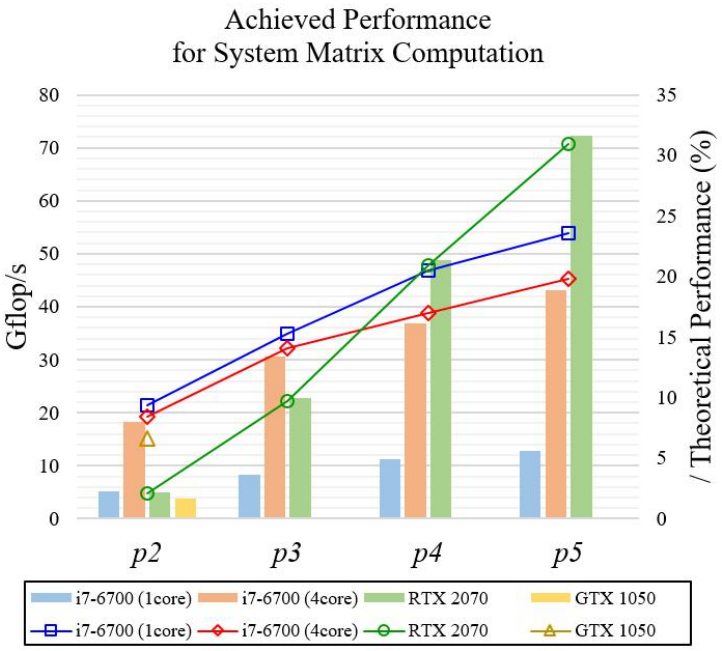


그림 4.12 시스템 행렬 계산에서 달성한 계산 성능 비교 (Grid#4)

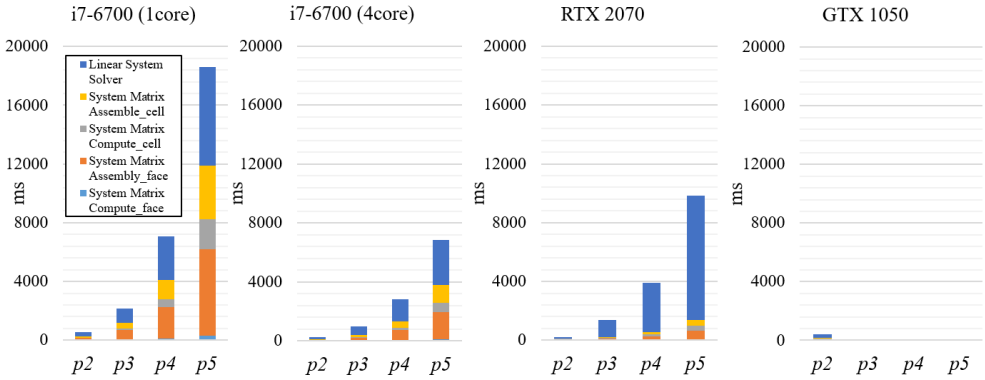


그림 4.13 내재적 시간 기법 계산 커널들의 시간 비용 비교 (Grid#4)

차수	*p*2	*p*3	*p*4	*p*5
face	0.01	0.08	0.29	1.04
cell	0.13	0.85	4.63	19.84

(a) 요소 별 평균 연산 크기(Mflops/TC kernel)

차수	*p*2	*p*3	*p*4	*p*5
face	0.09	0.58	2.07	7.38
cell	0.14	0.95	5.21	22.32

(b) 전체 연산 크기(Gflops)

표 4.6 내재적 시간 기법 연산 크기 (Grid#4)

제 5 장 결 론

불연속 갤러킨 방법 기반의 유동 해석 프로그램에 CUDA를 이용해 효율적인 GPU 계산을 적용하였고 수치 실험을 수행하여 CPU 코드와 계산 성능을 비교, 분석하였다.

GPU와 CPU는 하드웨어적 차이로 인해 병렬화의 방식에 다소 차이가 있는데 GPU는 근본적으로 수많은 코어를 기반으로 한 메모리 지연 감춤(memory latency hiding)을 통해 부동소수점 연산에서 높은 성능을 낼 수 있다. 본 연구에서는 이러한 GPU의 특성을 CUDA 코드에 잘 반영하기 위해 GPU의 스레드 체계와 메모리 체계에 대한 이해를 바탕으로 GPU 계산 커널을 최적화하였다. 불연속 갤러킨 방법의 텐서 축약 연산들을 효율적으로 계산하기 위해 GEMM(General matrix multi-plication) 커널 기반의 TC(Tensor contraction) 커널을 제작하였으며 작은 크기의 행렬에 대해 cuBLAS 라이브러리보다 높은 성능을 달성하였다. 플럭스 계산 커널에서 quadrature point에 대해 미세 그레인 병렬화(fine-grain parallelism)을 적용하였고 시간 적분 커널에서 전역 메모리 접근을 최소화하여 메모리 접근을 최적화하였다. 추가적으로 32개의 CUDA Stream을 이용해 계산의 동시성을 극대화시켰다. 내재적 시간 기법에서는 GPU 상에서 선형 시스템을 풀기 위해 MAGMA (Matrix algebra on GPU and multicore architecture) 라이브러리를 사용하였다.

수치 실험을 수행한 결과, 먼저 외재적 시간 기법에서는 차수가 낮을 때 텐서 축약 연산의 크기가 충분히 크지 못해 GPU의 계산 성능이 상대적으로 낮았지만 초고차에서는 텐서 축약 연산의 크기가 증가해 GPU의 계산 성능이 높아졌다. 격자를 구성하는 요소의 quadrature point 개수에 따라 텐서 축약 연산의 크기가 증감하면 GPU의 계산 성능에 영향을 미치는 것을 확인할 수 있었다. 내재적 시간 기법에서는 시스템 행렬 계산에 필요한 텐서 축약 연산의 크기가 매우 크기 때문에 외재적 시간 기법에 비해 GPU의 계산 성능이 높아졌다. 시스템 행렬 계산과 저장은 GPU가 CPU보다 더 효율적이었지만 GPU에서 MAGMA 라이브러리를 이용해 선형 시스템을 푸는 것이 CPU에서 PETSc (Portable, extensible toolkit for scientific computation) 라이브러리를 이용하는 것보다 상대적으로 비효율적이었다.

본 연구에서는 계산 커널 일부를 경합 조건(race condition)을 피하기 위해 순차적으로 실행하였는데 이로 인해 face에 대한 계산이 상대적으로 비효율적으로 수행되었다. 따라서 경합 조건 없이 효율적으로 face에 대해 계산할 수 있는 새로운 알고리즘에 대한 추가적인 연구가 필요할 것으로 보인다. 또한, 불연속 갤러킨 방법에서 각각의 연산들의 평균적인 크기가 커질수록 GPU의 계산 성능이 상승하는 경향이 있으므로 각각의 연산들을 하나의 거대한 연산으로 합치는 알고리즘에 대한 추가 연구가 필요하다. 마지막으로 GPU에 선형 시스템 솔버가 더 효율적으로 적용되면 GPU에서의 시간 비용을 많이 줄일 수 있을 것으로 예상된다.

참고 문헌

[1] B. Cockburn and C.-W. Shu, “TVB Runge-Kutta local projection discontinuous Galerkin finite element method for conservation laws. II. General framework,” *Mathematics of Computation*, vol. 52, no. 186, pp. 411-435, 1989

[2] H. You and C. Kim, "Complete-search Tensor Contractions for Optimizing High-order Methods," in *AIAA Scitech 2020 Forum*, (Orlando, FL), 2020.

[3] K. Banaś, P. Płaszewski and P. Macioł, “Numerical integration on GPUs for higher order finite elements,” *Computers & Mathematics with Applications*, vol. 67, no. 6, pp. 1319-1344, 2014.

[4] A. Abdelfattah, M. Baboulin, V. Dobrev, J. Dongarra, A. Haidar, I. Karlin, ... and S. Tomov, “Small Tensor Operations on Advanced Architectures for High-order Applications,” Technical Report UT-EECS-17-749, 04-2017, 2017.

[5] K. Świrydowicz, N. Chalmers, A. Karakus and T. Warburton, “Acceleration of tensor-product operations for high-order finite element methods,” *The International Journal of High Performance Computing Applications*, vol. 33, no. 4, pp. 735-757, 2019.

[6] https://www.top500.org/

[7] J. Cheng, M. Grossman and T. McKercher, *Professional Cuda C Programming*. John Wiley & Sons, 2014.

[8] F. Bassi, A. Crivellini, S. Rebay and M. Savini, "Discontinuous Galerkin solution of the Reynolds-averaged Navier–Stokes and k–ω turbulence model equations," *Computers & Fluids*, vol. 34, no. 4-5, pp. 507-540, 2005.

[9] R. J. Spiteri and S. J. Ruuth, "A new class of optimal high-order strong-stability-preserving time discretization methods," *SIAM Journal on Numerical Analysis*, vol. 40, no. 2, pp. 469-491, 2002.

[10] G. Wanner and E. Hairer, *Solving ordinary differential equations II*. Springer Berlin Heidelberg, 1996.

[11] G. Di Marzo, *RODAS5 (4)-Méthodes de Rosenbrock d'ordre 5 (4) adaptées aux problemes différentiels-algébriques*. MS Thesis, UNIVERSITY OF GENOVA, 1993.

[12] https://docs.nvidia.com/cuda/

[13] https://software.intel.com/en-us/parallel-studio-xe

[14] https://www.mcs.anl.gov/petsc/

[15] https://developer.nvidia.com/cuda-downloads

[16] https://icl.cs.utk.edu/magma/

Abstract

Study on Application of GPU Computing for Efficient Computation of Discontinuous Galerkin Method

Sungtae Kim

Department of Mechanical and Aerospace Engineering

The Graduate School

Seoul National University

This paper deals with study on application of GPU computing for efficient computation of discontinuous Galerkin method, by using CUDA. Although GPU (Graphics processor unit) is a device designed for generating display output, it has been studied actively in order to utilize GPU in HPC (High performance computing) as GPU is known to be capable of massive computation through latency hiding based on numerous cores. Meanwhile, in discontinuous Galerkin method, it is possible to achieve an arbitrary order of spatial accuracy while maintaining the stencil compact, because numerical solution is approximated as a linear combination of basis functions. In addition, computations in discontinuous Galerkin method can be represented as a form of tensor contraction with high arithmetic intensity, which leads to favorable chance of implementing GPU computing. In this research, GEMM (General matrix multiplication) kernel, which is based on CUDA programming, was used to compute

tensor contractions efficiently and high performance was achieved compared to cuBLAS (Cuda Basic linear algebra) library. Numerical experiments and performance analysis were conducted with the original CPU code and the developed CUDA code.

Keywords : Computational fluid dynamics, GPU computing, CUDA, Discontinuous Galerkin method, Tensor contraction, GEMM kernel
Student Number : 2018-25343