# Position Based Dynamics의 GPU 병렬화 연구
# GPU Parallelization Study of Position Based Dynamics

서지완, 박채림, 조세홍, 계희원*
Jiwan Seo, Chailim Park, Sae-Hong Cho, Heewon Kye*

(02876) 서울 성북구 삼선동 삼선교로16길 116 한성대학교 컴퓨터공학부
wannaseoji@naver.com
kuei@hansung.ac.kr

## 요약

의료영상 데이터를 이용한 의료 시뮬레이션 시스템은 피부조직과 같은 장기의 실제 거동을 모방하기 위해 물리 연산이 필요하다. 의료 시뮬레이션 프로그램은 크기가 큰 데이터를 사용하므로 연산 속도에 대한 제약을 받는다. 본 논문에서는 정밀성과 속도를 적절히 고려하여 Position Based Dynamics(PBD) 기반의 시뮬레이션을 실행한다. 그리고 GPU 병렬 처리를 통해 PBD를 고속으로 움직임을 계산할 수 있도록, Nvidia의 CUDA C/C++를 기반으로 프로그램을 구현하였다. 병렬화 과정에서 발생하는 반복되는 커널 실행의 비효율성 문제를 스레드 독립성을 이용하여 해결하였고, 변형체 해상도에 따른 성능 변화를 CPU와 GPU별로 비교하였다. 또한, CPU 및 GPU를 사용한 병렬 처리에서 병렬성의 한계를 검토하였다. GPU 병렬 처리를 통해 CPU 병렬 처리에서는 불가능한 큰 변형체에 대한 시뮬레이션의 대화적 시간의 확보에 성공하였다.

## Abstract

A simulation system using medical image data requires calculation to imitate the actual behavior of an organ such as skin tissue. Medical simulation programs have a problem in that they take a long calculation time because they use large amounts of data. In this paper, we conducted a simulation based on Position Based Dynamics (PBD) with consideration of precision and speed. In addition, we programmed a simulation system based on Nvidia's CUDA C/C++ so that PBD can be calculated at high speed through GPU parallel processing. Using thread independence property, we addressed efficiency of iterative kernel launch in the parallelization process and we compared the performance of CPU and GPU according to the resolution of the deformable object. In addition we examined the efficiency limitations of parallel processing in CPU and GPU. With GPU parallel processing, we succeeded in securing interactive time for simulations on medical deformable object that are not possible in CPU parallel processing.

키워드: GPU, 병렬화, Position Based Dynamics
Keyword: GPU, Parrallelization, Position Based Dynamics

## 1. 서론

가상 의료 시뮬레이션 소프트웨어는 인체에 대한 3차원 볼륨 데이터를 이용하여 의료인이 가상 환경에서 실제 환경을 모방한 시뮬레이션을 통해 경험 위주의 훈련이 가능하도록 돕는다. 이는 가상 의료 시뮬레이션에서의 반복된 훈련을 통해 경험 부족으로 발생하는 의료 사고를 줄이고 구체적인 수술 계획을 수립하도록 하여 의료 품질 향상에 이바지한다. 또한, 이러한 소프트웨어는 환자들을 위한 설명 자료로 사용될 경우 수술 과정에 대한 환자의 이해를 높일 수 있다. 따라서 가상 의료 시뮬레이션은

여러 방면에서 매우 필요한 기술이다[1].

의료 시뮬레이션 소프트웨어에서 의료인은 기구를 통해 피부를 절개하여 수술을 위한 공간을 확보하거나 찢어진 피부를 봉합하는 등의 외과적 의료행위를 주로 수행한다. 수술 시, 절개 및 봉합 과정에서 발생하는 피부조직과 같은 대상의 움직임을 실제와 유사하게 표현하기 위해서는 물리 시뮬레이션 연산을 적용해야 한다.

물리 시뮬레이션을 계산하는 방법에는 FEM(finite element method)[2], 질량-스프링[3], 체인메일(chain-mail)[4-6], 그리고 PBD(Position Based Dynamics)[7-9] 등 여러 알고리즘이 있다. FEM은 많은 연산을 수행하여 정밀도가 높은 방법으로, 절개와 같이 위상(topology)의 변형이 일어나는 경우 요소의 개수에 변화가 생겨 더 많은 계산이 필요하다. [10]과 같이 절개선에 맞추어 정점을 이동하는 해결 방법이 제안되었으나 많은 수의 입자를 다루기에는 속도가 느리다. 질량-스프링 모델은 명시적 적분 방법(explicit integration)을 사용하여 간단하게 움직임을 계산할 수 있으나, 잘 알려져 있듯이 불안정성 문제가 발생한다. 암시적 적분을 이용하여 큰 시간 간격에 대한 시뮬레이션을 수행할 수 있으나[11] 대규모의 선형 방정식을 풀어야 한다는 단점이 있다. 체인 메일 알고리즘은 연산 속도가 각 입자 당 한 번으로 매우 빠르지만, 쇠사슬이 늘어진 형태로 탄력이 없는 부자연스러운 움직임을 보인다.

본 연구에서 사용하고자 하는 PBD는 외력에 의해 변형된 물체의 움직임을 제약조건에 따라 정점에 대한 직접적인 수정을 가하여 움직임을 제어하는 방법이다. 수술 시 거동을 모방하여 표현하는 의료 시뮬레이션 프로그램은 빠른 속도로 실행되어야 하며, 실제와 같은 대상의 움직임을 표현해야 하므로 속도, 정밀성 그리고 안정성을 모두 고려해야 한다. PBD는 다양한 조건에서 균형을 이루고 있다.

본 연구에서는 위치 변화에 따라 많은 연산을 필요하므로 가속화 방법이 중요하다. 현대 CPU에는 다수의 연산 코어가 장착되어 있어서 하나의 메모리를 공유하며 여러 명령을 동시에 수행할 수 있다. SMP(Shared Memory Parallelism)는 병렬화 기술의 하나이며 SMP 방법의 대표적인 병렬화 라이브러리는 OpenMP이다. OpenMP를 사용하여 CPU에서의 병렬 처리 연구가 많이 이루어졌다[12-14]. [15]에서는 연산 시간이 가장 오래 걸리는 부분을 식별하여 해당 부분을 OpenMP를 사용하여 병렬화하였다.

CPU의 경우 GPU에 비해 코어 수가 적어 순차적인 작업에 효율적이지만, GPU의 경우 많은 코어를 가지고 병렬적으로 연산하므로 효율적이다. 본 연구에서는 변형체 계산 알고리즘인 PBD를 고속으로 움직임을 계산할 수 있도록, Nvidia의 CUDA C/C++를 기반으로 구현하였다.

본 연구의 공헌은 다음과 같다. GPU 병렬처리를 PBD에 적용하여 빠른 속도의 변형체 연산 알고리즘을 개발한다. 그리고, 이에 따라 발생하는 반복되는 커널 실행의 비효율성 문제를 스레드 독립성을 이용하여 해결한다. 마지막으로 영상 해상도에 따른 처리속도를 CPU와 GPU 특성에 따라 비교한다.

## 2. 관련 연구

### 2.1 Position Based Dynamics 개요

PBD는 기존의 힘 기반 엔진에서 사용하는 명시적 적분 기법(explicit integration scheme)으로 인해 발생할 수 있는 오버 슈팅과 같은 안정성 문제를 점을 직접적으로 조작함으로써 해결한다. 구체적으로, 점의 위치는 제약조건을 투영해 얻어진 이동량을 기존 점의 좌표에 누적하여 결정된다. 제약조건($C$)은 각 정점($P$)이 주변 정점과의 관계에 따라 위치해야 하는 안정적인 위치이며, $C(P)=0$을 만족하는 방향으로 정점의 위치를 조정한다. 일반적인 PBD 알고리즘은 다음과 같다.

<표 1> PBD 알고리즘

| **Algorithm Position Based Dynamics** |
|---|
| 1: 정점의 위치P, 속도V, 가속도A, (고정된 점을 다루기 위해) 질량 역수W 값 초기화 |
| 2: 외력(중력 등)이 작용 |
| 3: 모든 정점에 대해 가속도 업데이트 $A$:=FW |
| 4: 모든 정점에 대해 속도 업데이트 V:=V+A•dt |
| 5: 속도 방향으로 정점들 위치 수정 P:=P+V•dt |
| 6: 임의의 반복 횟수 동안 제약조건 투영(Constraint Projection) |
| 7: 모든 정점에 대해 수정된 움직임 반영 |
| 8: 정점의 속도(정점이 움직이기 전과 움직인 후를 이용)와 가속도 업데이트 |

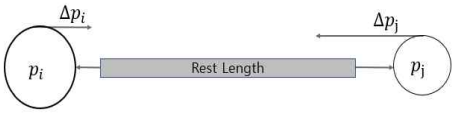
### 2.2 거리 제약

거리 제약은(Distance Constraint) 대표적인 제약조건으로, 정점들 간의 거리에 따라 위치를 조정한다. 거리 제약은 엣지를 이루는 두 개의 정점 사이의 거리가, 미리 설정된 안정한 거리가 되도록 적용한다.

PBD는 $C(P)=0$을 만족하는 방향으로 각 정점 $P$를 직접 조작하면서 움직이도록 한다. 거리 제약에서 외력이 발생할 때, 고정되지 않은 정점은 외력에 따라 움직인다. 하나의 엣지를 이루는 정점 $p_i, p_j$는 $C(p_i,p_j)=|p_i-p_j|-d=0$를 만족하기 위한 방향으로 움직이게 된다. 거리 제약은 정점 간의 거리가 멀어지면 서로 잡아당기며, 반대로 거리가 가까워지면 서로 밀어낸다. 점의 움직임 $\Delta P$를 제약의 그래디언트 방향으로 제한하며, 이것은 제약의 그래디언트 벡터의 방향으로 근을 찾는 가장 빠른 방향이다. 거리 제약에서 $\Delta p_i$과 $\Delta p_j$는 식(1)과 같다.

$$\Delta p_i = -\frac{w_i}{w_i + w_j}(|p_i - p_j| - d)\frac{p_i - p_j}{|p_i - p_j|},$$
$$\Delta p_j = +\frac{w_j}{w_i + w_j}(|p_i - p_j| - d)\frac{p_i - p_j}{|p_i - p_j|} \quad (1)$$

만약 각각 질량이 다르다면 $w$의 영향을 받게 되는데, $w$는 정점 질량의 역수이다. (그림 1)에서 $p_i, p_j$가 가진 질량 $m_i, m_j$에 따라, $m_i > m_j$일 경우 $\Delta p_i$의 값이 작아져 조금 이동하며, 반대로 $\Delta p_j$의 값이 커지므로 $p_j$가 많이 이동한다. $m_i = m_j$의 경우 두 점의 운동량은 같다. 점의 질량은 그림에서 원의 크기로 나타내었다.




(a) $m_i > m_j$

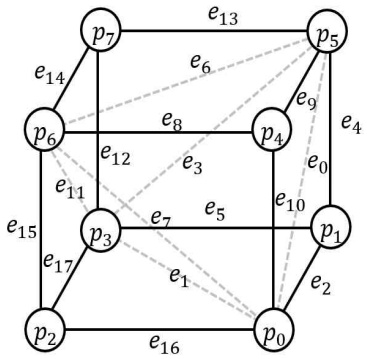

(b) $m_i = m_j$

(그림 1) 질량에 따른 정점의 움직임.

## 3. 본론

### 3.1 SimCell의 정의

엔진에서 사용되는 시뮬레이션 단위를 육면체의 SimCell로 정의한다. (그림 2)와 같이 SimCell은 인접한 정점 8개로 이루어진다. 전체 변형체가 $L \times M \times N$ 개의 정점으로 구성된다면,$(L-1)\times(M-1)\times(N-1)$ 개의 SimCell이 생성된다. 본 연구에서는 시뮬레이션 단위를 사면체로 정의하였고, 육면체인 각 SimCell은 그림과 같이 5개의 사면체로 분해한다. 이때, 사면체의 각 변을 엣지라고 정의한다.




(그림 2) SimCell과 18개의 엣지

많은 개수의 SimCell이 존재할 때, 엣지를 결정하는 방법은 다음과 같다. 하나의 SimCell에서 엣지의 수는 (그림 2)과 같이 18개이다. 이를 기본 단위로 여러 SimCell을 처리하기 위해 다수의 엣지를 생성한다. 여러 개의 SimCell은 서로 면을 공유하여 인접하므로, SimCell들에 중복으로 포함된 엣지는 한 번만 생성한다.

### 3.2 데이터에 따른 엣지의 확장

<표 2> 볼륨 크기에 따른 엣지의 증가

| **볼륨 크기** | 2×2×2 | 3×3×3 | 4×4×4 | 5×5×5 |
|---|---|---|---|---|
| **엣지 개수** | 18 | 90 | 252 | 540 |

정점의 수를 각 축에 대하여 하나씩 확장할 때마다 엣지의 수는 <표 2>와 같이 크게 증가한다. PBD에서 거리 제약을 사용한 투영의 연산 단위는 엣지이기 때문에 엣지의 수가 증가함에 따라 계산량이 늘어나며, 연산이 오래 걸리는 문제가 발생한다. 많은 연산이 필요한 본 연구에서는 GPU 병렬 연산을 통해 처리속도를 향상시키려 한다.

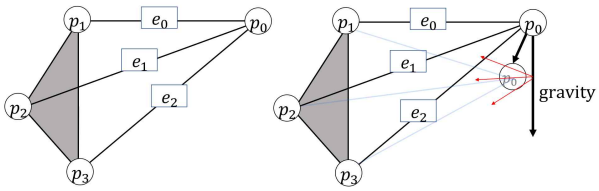
### 3.3 연구목표

GPU 병렬 처리에서는 대규모 스레드를 생성하여 병렬적으로 일을 수행하는 것이 일반적이다. 스레드 간에 데이터를 전달하는 스레드 종속성이 있다면, 올바른 연산을 수행하기 위해 비효율이 발생한다. 따라서 각 스레드는 서로 독립적으로 연산할 수 있도록 고려해야 한다[16].

PBD에서 대부분의 연산 시간은 제약조건에 따라 움직임을 수정하는 제약 투영에서 소요된다. 본 연구에서는 제약 투영 과정에서 GPU 병렬 처리를 효율적으로 수행한다. PBD의 가장 작은 연산 단위인 하나의 제약을 하나의 스레드가 담당한다. 따라서 거리 제약의 경우, 변형체를 구성하는 각 엣지에 대해 연산의 독립성을 확보하는 것이 본 연구의 목표이다.

### 3.4 엣지 연산 독립성과 종속성

엣지가 종속되어 있다는 것은 여러 개의 엣지가 하나의 정점을 공유하는 상황을 의미한다. 예를 들어 (그림 3)과 같이 사면체의 이동을 가정할 때, 삼각형면 하나는 벽에 고정되어 있고, 중력의 작용으로 점 $p_0$만 움직인다. $p_0$에 연결된 3개의 엣지 $e_0,e_1,e_2$가 병렬 처리되어 한 번에 움직이게 되면, 각 엣지 연산의 결과로 정점 $p_0$의 이동이 동시에 계산되고 동일한 지점 $p_0$에 계산 결과가 저장된다.




(그림 3) 엣지 연관성

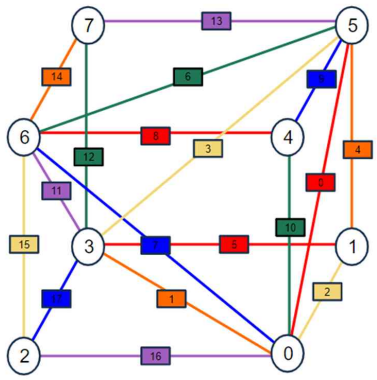
따라서 세 엣지 중 하나의 결과 값이 무작위로 저장되며 나머지 2개의 결과 값은 무시된다. 한편, 3개의 움직임을 모두 반영하기 위해 GPU의 원자 연산을 이용한다면, 세 엣지 사이에는 순차성이 생겨 병렬화의 효율이 크게 감소한다. 이때, 독립적으로 처리할 수 있는 엣지들을 서로 다른 그룹으로 분리하여 별도로 처리하면, 연산 효율을 유지하며 병렬적으로 변형체 연산을 수행할 수 있다. 각 엣지 연산의 독립성을 확보하는 방법은 다음과 같다.

<표 3> 엣지 독립성 확보

**Algorithm 엣지 독립성 확보**

```
for all Constraints C do.
    for all vertex V in C do
        Test V 원소표시 groups
    end for
    if Group에 V가 존재하지 않음 then
        엣지를 그룹에 추가
    end if
    if Group에 V 하나라도 존재 then
        다른 Group에 대해서 검사
    end if
    if 어떤 그룹에서도 해당 엣지가 추가되지 않음 then
        새로운 Group을 만들어 해당 엣지를 추가
    end if
end for
```

<표 3>과 같이 엣지 그룹을 만들면, 같은 정점을 공유하는 엣지는 서로 다른 그룹에 속하며, 동일 그룹 내에서는 같은 정점을 공유하는 엣지가 존재하지 않는다. 해당 과정을 통해 만들어진 엣지 배열은 한 그룹 내에 존재하는 모든 엣지에 대해 서로 독립적이며 해당 그룹 내부에 대해서 자유롭게 병렬적으로 처리할 수 있다. 크기가 2x2x2인 정육면체의 예를 (그림 4)에서 보인다. 엣지 연산 독립성을 확보하기 위해, 총 18개의 엣지에 대해 같은 그룹에 속하는 엣지는 동일한 색으로 표시하여 6개의 그룹으로 분리할 수 있다.



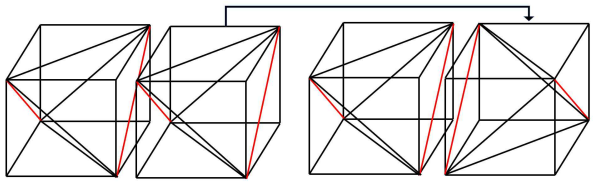

(그림 4) 엣지독립성이 확보된 SimCell.

### 3.5 ParallelizableEdgeGroups 정의

<표 4> ParallelizableEdgeGroups 2×2×2

| 엣지 / 그룹 | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 0 | 5 | 8 |
| 1 | 1 | 4 | 14 |
| 2 | 2 | 3 | 15 |
| 3 | 6 | 10 | 12 |
| 4 | 7 | 9 | 17 |
| 5 | 11 | 13 | 16 |

이렇게 만들어진 그룹들의 구조를 <표 4>와 같이 2차원 배열의 형태로 보였다. 하나의 행에 독립적으로 처리 가능한 엣지들이 모여 있는 배열을 ParallelizableEdgeGroups(PEG)로 정의한다. 병렬처리가 가능한 단위인 PEG의 한 행이 하나의 그룹이 된다. 병렬 처리하는 과정에서 같은 행의 원소들이 동시에 병렬적으로 처리된다. SimCell의 개수가 늘어날수록 하나의 그룹에 포함된 엣지 수가 늘어난다. 예를 들어, $15^3$ SimCell을 사용하는 경우 그룹의 수는 18개 불과하지만 하나의 그룹에 포함된 엣지의 수는 평균 1240개로 크게 늘어나기 때문에 병렬화 효율이 향상된다. 더 큰 데이터에 대해서는 더 큰 규모의 PEG가 생성된다. 다만, 셀의 경계 부분을 효과적으로 처리하기 위해, 인접한 셀의 경계 엣지는 일치되도록 거울 효과를 적용하여 (그림 5)와 같이 인접시킨다.



(그림 5) 거울 효과가 적용된 SimCell

### 3.6 PBD 구현

해당 장에서는 PBD를 구현하기 위해, 정점과 엣지를 정의한 방법에 대해 설명한다.

#### 3.6.1 정점 초기화 과정

우리는 정점을 위치, 속도, 가속도, 질량, 질량 역수 값을 가지고 있는 구조체로 구현하였다. 위치를 초기화하기 위해 각 $x,y,z$ 방향으로 일정 간격의 좌표를 생성하였다. 실험을 위해 속도에 대한 초기값은 0으로 설정하고, 가속도는 중력($g=-9.81$)만을 고려하여 $y$방향으로 부여하였다. 질량은 모두 같은 값으로 초기화하였으나, 특정 고정된 점을 표현하기 위해 질량의 역수값을 무한한 크기를 나타내도록 0으로 초기화하고, 가속도 또한 0으로 설정하였다.

#### 3.6.2 엣지 초기화 과정

<표 5> 엣지 초기화

**Algorithm 엣지 초기화**

**Input:**
SimCell을 이루는 사면체의 Edge Vertices를 담은 배열 V
SimCell을 이루는 모든 정점을 담은 배열 P
Edge 배열 E

```
Bool checking := false
for all SimCell을 이루는 각 사면체 do
   for all 사면체의 간선 n do
        v1 := V[n].first_vertexId
        v2 := V[n].second_vertexId
        for all P배열의 크기 do
           E의 원소에 v1,v2가 있는지 확인
                checking :=  true
                break
        end for
        if checking = false then
           E := E.add( v1, v2 )
           checking := true
         end if
    end for
end for
```

엣지는 두 개의 정점, 경도(stiffness), 두 정점 사이의 안정한 거리를 나타내는 값을 저장하고 있는 구조체로 구현하였다. 안정한 거리는 각 엣지를 구성하는 초기화된 정점 사이의 거리로 정의된다. 엣지를 초기화 하는 과정은 다음과 같다. 하나의 볼륨 데이터는 SimCell이 인접한 구조이므로, 전체 엣지를 생성하려면 하나의 SimCell을 구성하는 엣지들에 대해 좌표를 이동하여 반복해 생성해야 한다.

이때 SimCell의 인접한 면에서 엣지들이 교차하므로, (그림 5)와 같이 거울 효과를 주어 대칭 이동을 수행한다. 이는 일반적인 텍스처 매핑에서 경계면 처리에 대한 거울 효과와 유사하다. 실제 구현에서는 정점을 거울 효과로 대칭 이동한 좌표들을 미리 생성하여 저장한다.

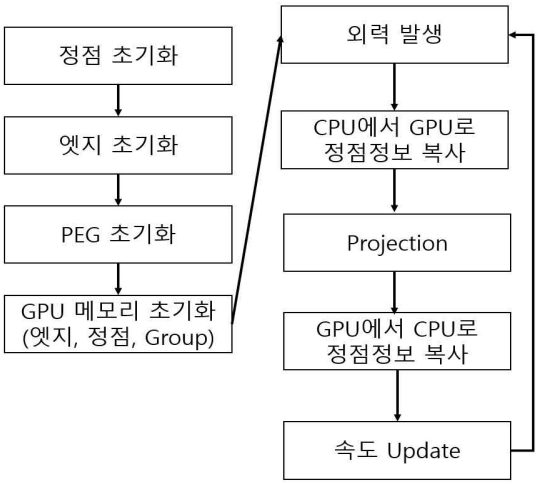
## 4. GPU 구현

해당 장에서는 투영 과정의 병렬화를 위한 GPU 구현에 대해 설명한다. 앞서 PEG의 n번째 행의 모든 원소가 동시에 처리될 수 있도록 그룹을 만들었다. 우리의 프로그램에서는 Nvidia의 CUDA C/C++을 사용하여, 투영 과정에 대해 한 그룹 단위로 병렬적인 연산을 수행하였다.

### 4.1 GPU 메모리 관리

서론에서 어려움으로 언급된 메모리 관리에 대하여, 다음과 같은 처리가 필요하다. GPU와 CPU에서 주고받을 데이터는 1. 엣지, 2. 정점 3. 그룹 세가지가 있다. 이 데이터들은 GPU를 활용하기 이전 (그림 6)과 같이 CPU 내에서 모두 초기화된 상태이다.

먼저, GPU 공간은 한 번만 초기화하면 되는 GPU 메모리 부분, CPU 메모리와 정보를 교환하고 계속 크기를 바꾸어 할당해 줘야 하는 GPU 메모리 부분으로 나뉜다. 엔진의 정점과 엣지를 초기화하는 과정은 투영 반복(projection iteration)의 실행 전 CPU 내에서 한 번만 수행하면 된다. 또한, GPU 내부에서도 정점, 엣지에 대한 메모리 할당은 한 번만 수행하여 CPU 메모리에 저장된 초기값을 받아 할당을 진행하고 그 후에는 CPU와 GPU 사이에서의 복사만 이루어진다.




(그림 6) PBD 구성도

반면, PEG는 서로 다른 크기의 그룹들을 가지고 있다. 매번 그룹을 변경할 때마다 CPU에서 GPU로 메모리 할당과 해제를 반복하면 비효율적이다. 본 연구는 CPU에서 2차원 배열인 PEG의 행의 크기 만큼 반복하면서 가장 큰 그룹의 크기만큼만 동적 할당하고 GPU 공간으로는 복사만 효율적으로 수행하였다.

동적 할당 시, 필요한 그룹과 가장 큰 그룹의 크기는 인자로 받아 사용한다. GPU 내에서 그룹의 크기만큼 복사가 이루어진다. 이 과정은 투영마다 PEG의 행의 개수만큼 진행된다. 연산이 끝나면, 해당 값을 GPU 내부에 존재하는 갱신된 정점에 반영한다. 본 연구에서는 속도 값의 갱신이 CPU에서 일어난다. 이 경우 GPU에 있는 값들을 CPU로 복사하고, 속도 갱신이 완료된 정점들은 다음 반복에서 GPU에 복사된다.

### 4.2 스레드 생성과 커널 실행

커널 실행를 위한 스레드는 다음과 같은 과정을 통해 생성된다. 사용하는 GPU 장치에 따라 한 번에 호출할 수 있는 스레드의 개수가 다르다. 엣지 그룹에 속한 원소의 개수가 최대 생성 가능한 스레드 수보다 작은 경우, 커널 실행 시 스레드는 원소의 개수만큼 생성하여 커널 실행 하면 병렬적으로 연산이 이루어진다. 만약, 어떤 그룹에 속한 원소의 개수가 최대 생성 가능한 스레드 수를 초과한 경우 여러 개의 블록을 생성하여 나누어 처리한다.

커널 함수의 파라미터는 GPU 메모리에 복사된 그룹, 엣지, 정점이다. 커널 함수가 불리면 스레드의 각 원소마다 연산이 이루어진다. 인자로 전달된 그룹의 원소는 엣지이며, 연산에 필요한 엣지 구조체를 이루는 정점과 안정한 거리, 그리고 경도를 추출한다. 외력에 의해 움직인 정점에 대해서 직접적인 수정을 가하기 위해 거리 제약을 적용하여 외력에 의해 변형된 좌표를 수정하기 위한 제약 투영을 실행한다.

#### 4.2.1 거리 제약의 투영 과정

2.2절에서 설명한 거리 제약을 구체적인 예시를 들어 설명한다. 이를 계산하면, 정점의 위치가 직접적으로 수정된다.

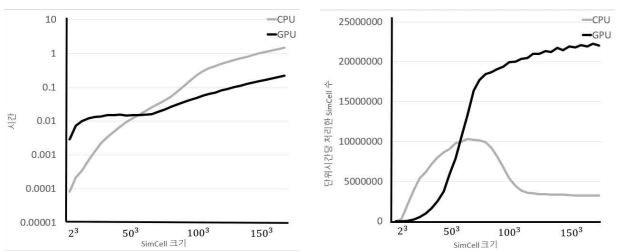
이동을 계산하기 위해서는 입력값으로 질량의 역수 $w$, 정점의 좌표 $p$, 정점 간의 안정적인 거리 $d$가 필요하다. 각 $w$, $p$는 정점에 부여된 정보이므로, 정점 버퍼에 있고, $d$는 엣지에 부여된 정보이므로 엣지 버퍼에 존재한다. 정점 버퍼는 매 프레임마다 시스템 메모리에서 GPU 전역 메모리로 복사되며, 프로그램에서 변화가 없는 엣지 버퍼는 GPU의 전역 메모리에 초기화 시 한번 복사된 상태에서 유지된다. 이렇게 얻은 값을 식 (1)을 적용하여 기존 좌표값 $p$를 갱신한다. 이렇게 이동량을 반영한 정점은 다시 GPU 메모리로 반영시키고, GPU 메모리에서 CPU 메모리로 복사하는 과정이 필요하다. CPU로 연산된 정점이 돌아오게 되면 한 번의 시뮬레이션 단위의 반복이 끝나며 그 후에 GPU에 저장된 정점 정보를 통해 렌더링이 이루어진다. 이 과정을 모든 엣지가 안정한 상태를 유지할 때까지 반복한다. 안정한 상태라는 것은 $C(P)=0$에 가까운 상태이다.

## 5. 실험 결과

이번 장에서는 구체적인 구현 방법과 실험 결과를 설명한다. 실험에서는 NVIDIA GeForce RTX 3090 GPU와, AMD Ryzen 9 3900X 12-Core Processor @ 3.80GHz CPU (64.0GB RAM)을 사용하였다.

우선, CPU와 GPU의 성능을 SimCell의 개수에 따라 비교하였다. SimCell의 개수를 $x$, $y$, $z$ 축으로 5씩 증가시키며 성능 실험을 진행하였다. 한 번의 물리 연산 후에 5회의 투영을 수행하는 일련의 작업을 30회 반복하여 평균 내었다. 우리의 프로그램은 실험을 위해 PBD에서 대표적인 거리 제약을 제약조건으로 채택하였다.

실험 결과로 얻은 (그림 7 (a))은 SimCell의 개수 증가에 따른 처리시간의 변화를 보이고 있고, (그림 7 (b))는 단위 시간당 처리한 SimCell의 개수를 도시하였다.




(a) (b)

(그림 7) (a) SimCell 처리시간
(b) 단위시간 당 처리 SimCell 수

(그림 7 (a))에서 볼 수 있듯 SimCell의 개수가 상대적으로 작을 때는 GPU 병렬화 보다 오히려 CPU의 병렬화를 거친 시스템의 성능이 좋은 것을 확인할 수 있으나, SimCell의 개수가 점차 많아지면서 $55^3$의 구간에서 역전되는 것을 확인하였다. SimCell의 개수가 작을 때 CPU가 더 효율적인 이유는 GPU 병렬화 과정 중, (그림 6)의 CPU, GPU 사이의 데이터 전달과정에서 오버헤드가 발생하기 때문이다. 이것이 병렬화로 인한 이득을 상쇄한다. 이후 SimCell의 개수가 증가하고, 각 병렬화 그룹의 크기가 커지면서 병렬화의 효율이 증가한다. 따라서 CPU와 GPU 간 데이터 전달에 따른 오버헤드가 병렬화의 효율에 비해 무시할 정도로 작아지므로, 일정 구간 이후 GPU 병렬화가 CPU 병렬화보다 성능이 확연히 빠르게 나타난다.

매우 작은 규모의 시뮬레이션을 수행할 때는, CPU 병렬화로도 좋은 성능을 보인다는 것이 본 연구를 통해 설명된다. 그러나 일반적인 의료 시뮬레이션은 큰 크기의 볼륨 데이터를 사용하는 것이 보통이므로, SimCell의 개수가 많아 GPU 병렬화가 더 효과적이다.

구체적으로 (그림 7 (b))에서 1초에 처리하는 SimCell의 개수를 계산하여 도시하였다. 즉, (그림 7 (b))는 병렬화의 효율을 의미한다. CPU의 경우, OpenMP 병렬화 규모에 따른 성능 향상이 존재하므로 일정 정도의 SimCell의 개수가 증가하는 동안 효율이 증가한다. 그런데, SimCell의 개수가 $60^3$를 넘어서면서 효율이 급격하게 감소하는 것을 확인할 수 있다. 다루는 SimCell 개수가 충분히 많아지면 CPU 하드웨어의 병렬화 기능을 모두 사용하면서 더 이상의 효율 향상을 얻을 수 없고, 메모리 용량이 커짐에 따라 메모리 참조의 효율성이 감소하기 때문으로 추정된다.

한편 GPU는 초기 오버헤드가 존재하지만, 실험에 사용한 GeForce RTX 3090은 10496개의 코어를 장착하고 있어서 수만 개의 스레드를 동시에 동작할 수 있을 정도로 병렬화 가능 정도가 크다. 따라서 전반적인 효율은 $70^3$까지는 급격하게 증가하며, 그 이후로도 완만하게 효율이 상승하는 것을 확인할 수 있다. 그 이유는 다음과 같이 추정된다. SimCell 개수가 증가할수록 병렬화되는 단위인 하나의 그룹(<표 4> PEG의 한 행)의 크기가 커지게 되며, 그룹에 속한 엣지의 수가 GPU 코어 또는 한 번에 생성할 수 있는 스레드 수보다 (본 연구에서는 수만 개) 커지게 되면, 결국 GPU 내부에서도 순차 처리가 일어나게 되므로 효율 향상이 멈추게 된다. 최종적으로 SimCell의 개수가 $169^3$이 되는 큰 데이터의 경우 GPU의 성능이 병렬처리된 CPU 보다 약 6.8배까지 우수한 것을 확인하였다.

본 연구를 통해 GPU 병렬화의 경우 원본 볼륨 데이터의 1/4 정밀도인 $128^3$ 크기의 변형체까지 대화적 속도(0.1초)로 시뮬레이션이 수행되었다. 즉, GPU 병렬화를 통해 높은 정밀도를 가진 변형체 시뮬레이션이 가능하였다.

## 6. 결론

본 논문에서는 기존 PBD의 성능을 개선하여 투영 과정에 대한 GPU 병렬화를 수행하였다. 엣지 연산 독립성 확보를 위하여 SimCell을 구성하는 독립적인 엣지들을 선별하여 그룹으로 만든 PEG를 생성하였다. PEG의 행 원소인 그룹의 각 원소에 대해 대규모 GPU 커널 실행을 통해 GPU 병렬화하였다.

CPU 병렬화와 체계적인 속도 비교를 수행하였으며, 매우 작은 데이터의 경우 CPU의 병렬화도 충분히 빠른 속도를 얻었다. 다만, 데이터의 크기가 증가하면서 CPU에서의 효율이 급격하게 하락하는 점을 발견하였다. 일반적인 시뮬레이션 크기에서는 GPU 병렬 처리를 이용하였을 때 CPU 병렬 처리보다 6배 이상의 성능 향상을 얻었고, $128^3$ 크기의 큰 변형체에 대해 대화적 시간의 시뮬레이션이 가능하였다.

따라서 본 연구는 상대적으로 큰 변형체에 대해 대화적 시간의 시뮬레이션을 수행하였다. 이를 통해 의료 시뮬레이션에서 대화적 시간으로 더욱 정확하게 변형체를 연산할 수 있다. 향후 연구로는 공유 메모리와 같은 GPU의 기능을 활용하여 더욱 향상된 속도로 시뮬레이션을 수행하려 한다.

## Ⅱ 참고문헌


[1] Badash, I., Burtt, K., Solorzano, C. A., & Carey, J. N. (2016). Innovations in surgery simulation: a review of past, current and future techniques. Annals of translational medicine, 4(23).

[2] Spyrakos, C. C. (1994). Finite Element Modeling. Morgantown, WV, USA: West Virginia Univ. Press.

[3] Duan, Y., Huang, W., Chang, H., Chen, W., Zhou, J., Teo, S. K., ... & Chang, S. (2014). Volume preserved mass-spring model with novel constraints for soft tissue deformation. IEEE journal of biomedical and health informatics, 20(1), 268-280.

[4] S. F. Gibson, “3D Chainmail: A Fast Algorithm for Deforming Volumetric Objects,” in Proceedings of the 1997Symposium on Interactive 3D Graphics. ACM, 1997, p. 149. 49

[5] Rodríguez, A., León, A., & Arroyo, G. (2016). Parallel deformation of heterogeneous ChainMail models: Application to interactive deformation of large medical volumes. Computers in Biology and Medicine, 79, 222-232.

[6] Bartelheimer, K., Teske, H., Bendl, R., & Giske, K. (2017). Tissue-specific transformation model for CT-images. Current Directions in Biomedical Engineering, 3(2), 525-528.

[7] Camara, M., Mayer, E., Darzi, A., & Pratt, P. (2016). Soft tissue deformation for surgical simulation: a position-based dynamics approach. International journal of computer assisted radiology and surgery, 11(6), 919-928.

[8] Bender, J., Müller, M., & Macklin, M. (2017). A survey on position based dynamics, 2017. Proceedings of the European Association for Computer Graphics: Tutorials, 1-31.

[9] Segato, A., Di Vece, C., Zucchelli, S., Di Marzo, M., Wendler, T., Azampour, M. F., ... & De Momi, E. (2021). Position-based dynamics simulator of brain deformations for path planning and intra-operative control in keyhole neurosurgery. IEEE Robotics and Automation Letters, 6(3), 6061-6067.

[10] Nienhuys, H. W., & Frank van der Stappen, A. (2001, October). A surgery simulation supporting cuts and finite element deformation. In International conference on medical image computing and computer-assisted intervention (pp. 145-152). Springer, Berlin, Heidelberg.

[11] Baraff, D., & Witkin, A. (1998, July). Large steps in cloth simulation. In Proceedings of the 25th annual conference on Computer graphics and interactive techniques (pp. 43-54).

[12] Couturier, R., & Chipot, C. (2000). Parallel molecular dynamics using OpenMP on a shared memory machine. Computer physics communications, 124(1), 49-59.

[13] Goedecker, S. (2002). Optimization and parallelization of a force field for silicon using OpenMP. Computer physics communications, 148(1), 124-135.

[14] Ayguade, E., Gonzalez, M., Martorell, X., & Jost, G. (2004, April). Employing nested OpenMP for the parallelization of multi-zone computational fluid dynamics applications. In 18th International Parallel and Distributed Processing Symposium, 2004. Proceedings. (p. 6). IEEE.

[15] Huang, P., Zhang, X., Ma, S., & Wang, H. K. (2008). Shared memory OpenMP parallelization of explicit MPM and its application to hypervelocity impact. Computer Modeling in Engineering & Sciences, 38(2), 119-148.

[16] Pan, J., Zhang, L., Yu, P., Shen, Y., Wang, H., Hao, H., & Qin, H. (2020, March). Real-time VR simulation of laparoscopic cholecystectomy based on parallel position-based dynamics in GPU. In 2020 IEEE Conference on Virtual Reality and 3D User Interfaces (VR) (pp. 548-556). IEEE.