〈양식 2〉 결과 보고서_2023학년도 졸업심화연구 결과 보고서

연구 횟수	1	2	3	4	5
참여 날짜	12.6	12.7	12.8	12.9	12.13

지도교사 확인
김현철 (인)

GPGPU의 선택적 활용과 코드 최적화를 통한 3차원 볼록 껍질 계산의 가속

3304 박찬웅
지도 교사: 김현철
경기북과학고등학교

Accelerate the 3D convex hull computation through selective GPGPU and code optimization

3120 Park ChanUng
GyeonggiBuk Science High School

초록(Abstract)

이전 연구에서 3차원 볼록 껍질 알고리즘을 OpenGL의 Compute Shader를 이용해 GPGPU로 계산할 수 있도록 구현하였다. 허나 GPU의 연산이 의미있어지는 지점에서의 처리시간이 너무 길어 렌더링 사이클 내에 삽입할 수 없었는데, 이를 해결하기 위해 본 연구에서는 코드 최적화와 선택적 GPU 활용을 통하여 실시간 렌더링에서 의미 있는 시간 내 (0.1초)로 연산이 가능한 정점 개수에 대해 GPGPU가 CPU에 대해 유의미한 시간적 이점을 가질 수 있도록 최적화하였다.

주제어: Computer Graphics, OpenGL, Optimization, GPGPU, Convex Hull

Ⅰ. 서론

이전 연구에서, 3차원 볼록 껍질(Convex Hull)을 Quick Hull 알고리즘을 응용하여 OpenGL Compute Shader의 GPGPU로 구성하는 연구를 진행하였다. 허나, 일전의 연구에서는 GPU의 연산이 CPU의 연산에 대해 유의미한 이점을 지니는 100만개 이상의 정점에 대해서, 실행 시간이 몇 초 정도 소모되었는데, 이는 최대 1/10s 이내의 시간으로 연산을 끝마쳐야 하는 실시간 렌더링에 대해서는 사용할 수 없는 속도이고, 따라서 해당 연구 결과를 응용하여 새로운 연구를 진행하기도 어려움이 있었다.

따라서 본 연구에서는 알고리즘의 오버헤드를 줄이고, GPU를 연산 단위에 따라 선택적으로 활용할 수 있게 하여 연구를 실용적인 분야에서 활용할 수 있도록 개선하였다.

추가로, 일전의 연구 코드의 경우, 정점의 개수가 충분히 크지 않은 경우 랜덤하게 Convex Hull이 잘못 구성되는 문제가 발생하였는데, 이 문제의 발생 원인을 찾고 해결하였다.

Ⅱ. 이론적 배경

1. Quick Hull 알고리즘

1) 알고리즘의 설명

QuickHull 알고리즘은 N 차원에 흩뿌려져 있는 정점들에 대해, 일반적인 경우 O(nlogn)의 시간복잡도

로 볼록 껍질 (주어진 점이나 영역을 포함하는 가장 작은 볼록 집합)을 구하는 알고리즘이다.

2) 코드에서 알고리즘의 구성

1) CreateSimplex : QuickHull 알고리즘을 시작하기 위한 사면체를 구성한다. 이 사면체의 경우 포인트 클라우드에서 가장 외곽에 있는 것이 효율적이다.

2) GetFurthestPoint : 다면체를 구성하는 평면들 외곽에 있는 점들 중, 가장 멀리 떨어져 있는 점을 구한다. 이 정점을 다면체에 추가시키게 된다.

3) NextPolyhedron : QuickHull 다면체와 GetFurthestPoint로 구한 가장 먼 점을 이용하여 이 점을 다면체에 편입시킨다. 코드 특성상 O(1)에 가깝게 동작하므로 CPU만으로 동작시킨다.

4) DivideOutside : 다면체에 바깥쪽에 있었던 점에 대해서, 새로 추가된 점으로 인해 내부로 편입되게 되는 점을 계산한다.

2. GPGPU

GPGPU는 General-Purpose computing on Graphics Processing Units의 약자로, GPU가 그래픽 렌더링 파이프라인의 처리뿐만이 아니라, 원래는 CPU가 맡았던 연산에도 활용해 연산 속도를 향상시키는 기술이다.

GPGPU로 병렬 처리를 할 경우, 병렬도가 높은 프로그램에서는 높은 속도 향상을 이룰 수 있지만, 병렬도가 낮은 프로그램에서는 오히려 CPU로 연산하는 것보다 느려질 수 있다. GPU 코어의 성능은 CPU 코어보다 보통 떨어지기 때문이다.

Ⅲ. 연구 내용 및 방법

연구를 진행하기에 앞서, 일전에 급하게 코드를 작성하여 발생한 기술부채를 해결하였다.

1. 프로젝트가 Convex와 ConvexGPU로 나누어져 Cmake를 수정하여 실행해야 했던 것을 GPU 사용 여부를 내부 상수로 소프트 코딩하여 프로젝트를 정리하였다.
2. 점, 선, 평면, 배경의 색상 지정을 하드 코딩하지 않고 유연하게 변경할 수 있도록 코드를 수정하였다.

또한, 이전 코드에서 있었던 문제점도 해결하였다.

1. 적은 수의 점에 대해, CreateSimplex 과정에서 4개 미만의 후보 점이 설정되어 Simplex가 제대로 형성되지 않던 문제를 해결하였다.
2. Simplex 안의 정점이 적을 경우 NextPolyhedron으로 형성된 새로운 평면의 법선 벡터 방향이 제대로 설정되지 않던 문제를 해결하였다.

그 후, 아래와 같은 개선을 통하여 계산의 성능을 향상하였다.

A. 원래는 DivideOutside의 계산 과정에서, N개의 평면과 M개의 정점이 있다고 하면, CPU로는 N*M번 연산을 수행하였고, GPU로는 M개의 정점을 계산하는 병렬 연산을 N번 수행하여 DivideOutside 연산을 수행하였다. 허나, 셰이더 코드에 수정을 가해 GPU가 N*M 사이즈의 행렬을 한번에 출력할 수 있게 하여 오버헤드를 크게 감소시켰다.

B. 약식 테스트 결과, DivideOutside 연산은 2000미만의 계산 단위, (계산 단위는, 연산 과정에서 병목이 일어나는 특정 연산을 수행해야하는 횟수를 의미한다. 이를테면, DivideOutside에서는 평면의 개수 * Outside 정점의 개수가 계산 단위가 된다.) FurthestPoint 연산은 모든 계산 단위에서 CPU가 우세하였다. 따라서 계산 단위를 미리 계산하여 CPU와 GPU를 번갈아가며 사용하게 함으로써 최적화하였다.

C. CreateSImplex, GetFurthest, NextPolyhedron, DivideOutside 4개의 핵심 연산들의 소요 시간을 측정

함으로써 어떤 연산이 가장 큰 시간을 소요하는지 검사하였다.
D. C.의 결과 CPU로만 연산하는 CreateSimplex가 연산 시간에 큰 지분을 차지한다는 것을 확인하고, GPU를 이용하여 병렬로 연산할 수 있도록 개선하였다.

전체 소스코드는 https://github.com/zlfn/ConvexGL 에서 다운로드할 수 있다.

Ⅳ. 연구 결과

CPU, GPU0, GPU1의 사양과 환경은 이전 연구와 같다.

CPU : i7-10750H, 단일 스레드 활용

GPU0 : Intel(R) UHD Graphics

작업 그룹 최대 개수 X, Y, Z 모두 2147483646

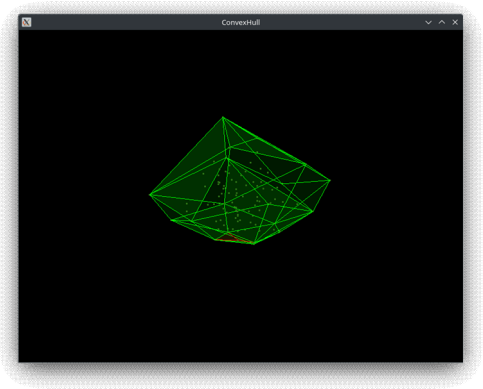
SSBO 최대 크기 128MB

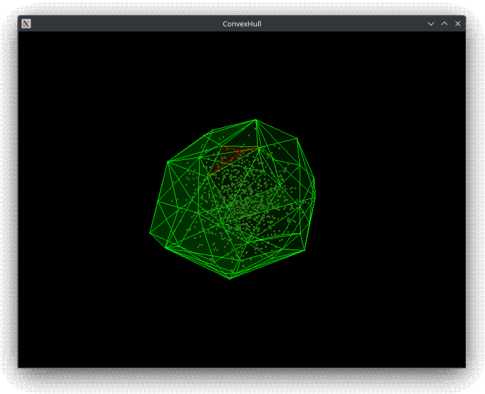
GPU1 : Nvidia RTX 2070 with Max-Q Design

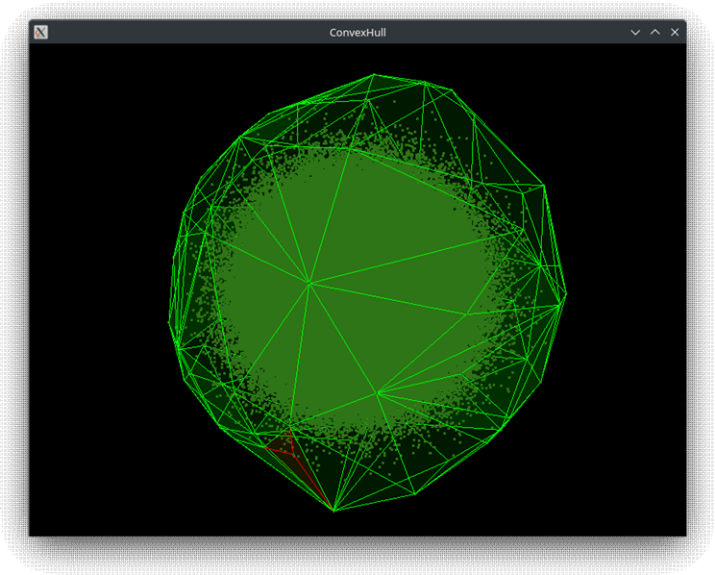
작업 그룹 최대 개수 X=2147483636, Y, Z=65535

SSBO 최대 크기 2048MB

Env : Arch LInux 64bit, X11, KDE



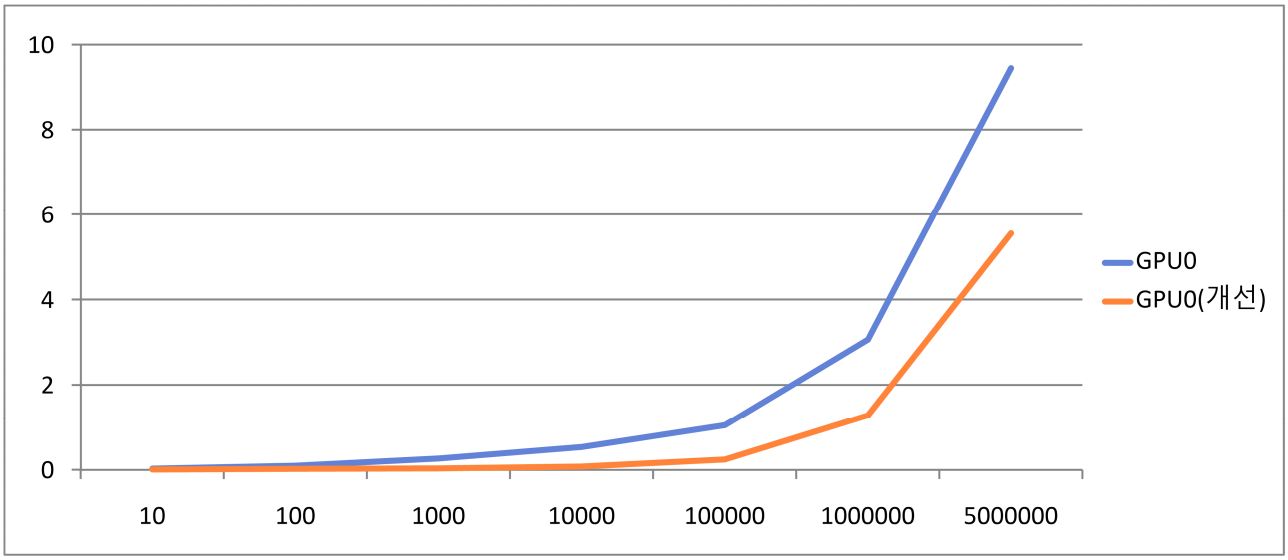




위 연구 과정에서 A, B를 진행하고, GPU0을 이용해 연산을 시행하였을 때 정점의 개수에 따른 소요 시간은 아래와 같다. (단위는 초이며, 5회 시행 후 평균 값이다.

)

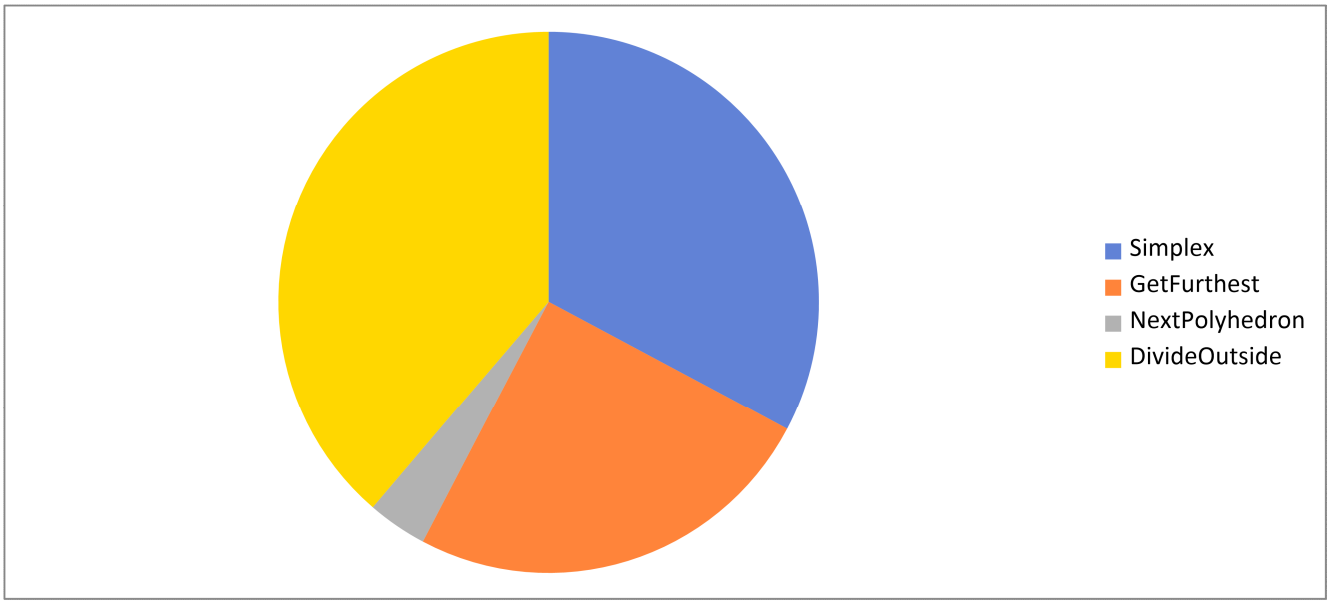
정점 개수	10	100	1000	10000	100000	1000000	5000000
개선 전	0.026	0.094	0.264	0.534	1.048	3.062	9.448
개선 후	0.0064	0.0226	0.0327	0.0762	0.2426	1.2825	5.5553



충분한 오버헤드 개선 효과가 있었음을 확인할 수 있다.

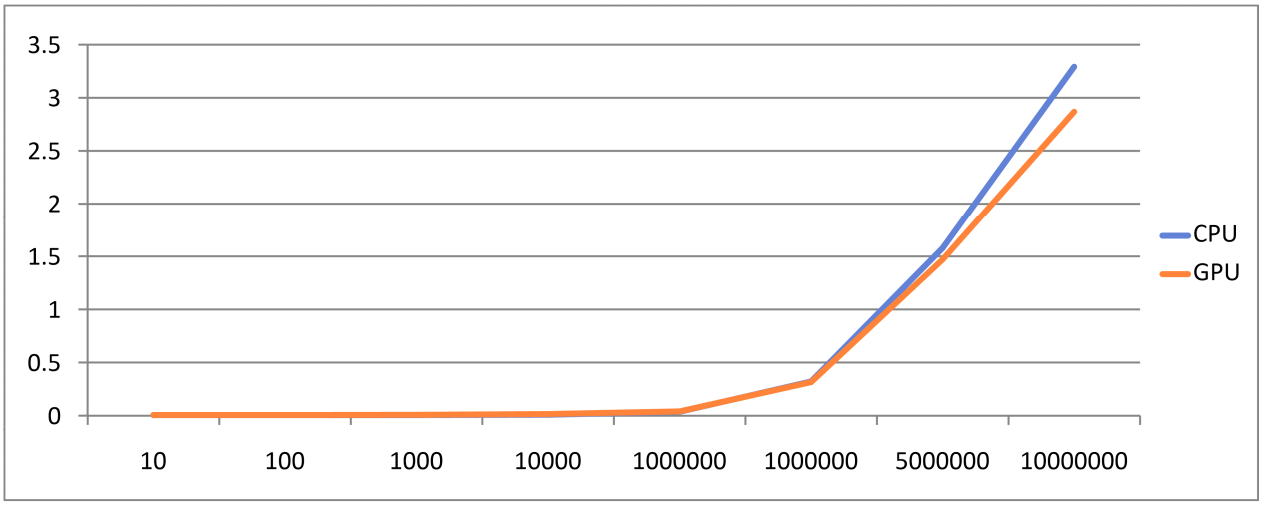
C에서 측정한 각 과정 소요 시간은 아래와 같다.

Simplex	GetFurthest	NextPolyhedron	DivideOutside
0.342476	0.260038	0.037309	0.40433



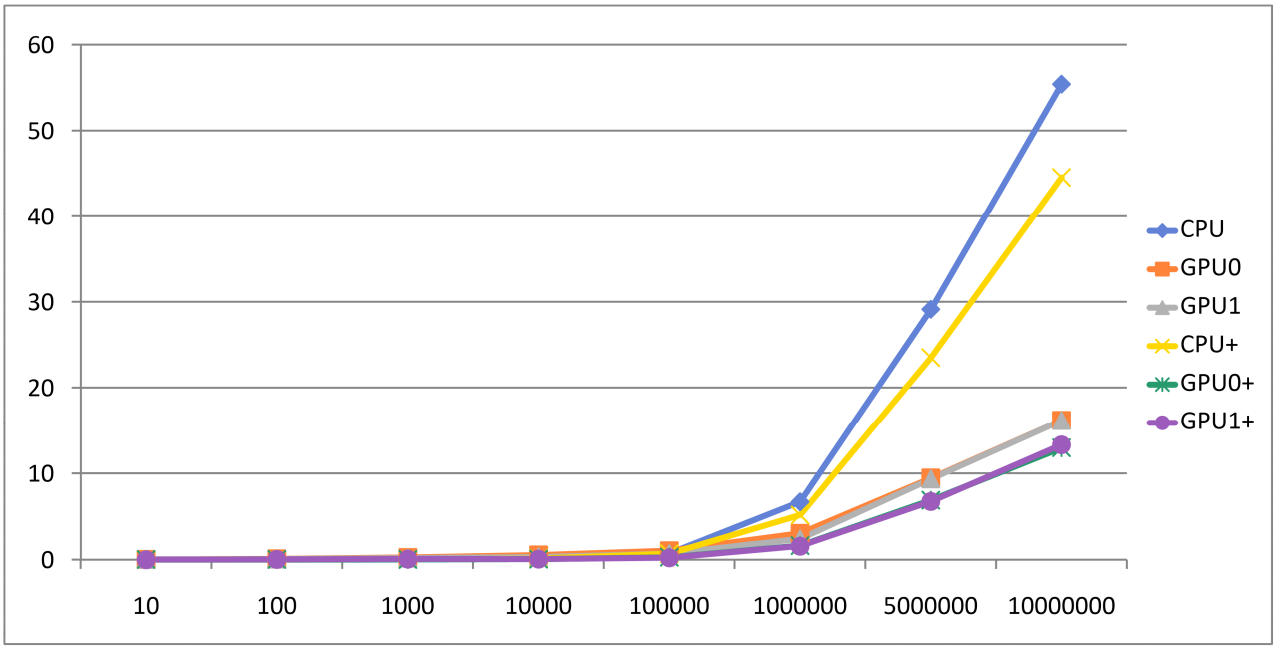
D에서 개선을 시행한 CreateSimplex 연산의 정점 개수에 따른 연산 시간은 아래와 같다.

	10	100	1000	10000	100000	1000000	5000000	10000000
CPU	0.0046	0.0047	0.0049	0.0084	0.036	0.3209	1.5789	3.2917
GPU	0.0058	0.0054	0.0075	0.0147	0.0395	0.3152	1.4711	2.868



CPU, GPU0, GPU1으로 측정한 최종 개선 값은 아래와 같다.

	10	100	1000	10000	100000	1000000	5000000	10000000
CPU	0.0007	0.0053	0.026	0.126	0.734	6.716	29.12	55.4
GPU0	0.026	0.094	0.264	0.534	1.048	3.062	9.448	16.17
GPU1	0.025	0.062	0.180	0.302	0.698	2.412	9.348	16.16
CPU+	0.0050	0.0109	0.0318	0.118	0.687	5.173	23.486	44.56
GPU0+	0.0049	0.0102	0.0306	0.0705	0.2735	1.633	6.900	12.958
GPU1+	0.0022	0.0158	0.0746	0.0507	0.2121	1.570	6.7307	13.3101



Ⅴ. 결론 및 제언

이전의 코드에 비해 확실한 개선 효과를 볼 수 있었으며, 특히 모든 정점 개수 구간에 대해 GPGPU 연산이 CPU 연산 속도와 거의 비슷하거나 더 빠르다는 결과를 얻을 수 있었다.

이를 통해 볼록 껍질을 실시간 렌더링에 사용할 때 확실한 이점을 얻을 수 있을 것이며, 특히 1만개 이상의 점에 대해서 GPU가 확실한 우위가 있으므로, Convex Hull이 필요한 렌더링 상황에서 유용하게 이용할 수 있을 것으로 기대된다.

■ 참고 문헌

[1] OpenGL 컴퓨터 셰이더를 활용한 3차원 볼록 껍질 계산의 최적화 (경기북과학고, 박찬웅)