Journal of Digital Contents Society Vol. 17 No. 3 Jun. 2016(pp. 165-172)
http://dx.doi.org/10.9728/dcs.2016.17.3.165

IEEE 754 부동 소수점 32비트 float 변수의 Morton Code 변환 분석

박태정*

요 약

GPU 기반 병렬처리에서 대규모 데이터의 인접 정보 검색(nearest neighbor search)에서 Morton code의 역할이 점점 더 중요하게 부각되고 있으며 그 응용 사례도 점차 증가하고 있다. 본 논문에서는 Tero Karras가 제안한 float 형 변수에 기반한 $[0,1]^3$ 공간 내의 3차원 기하 정보를 32비트 unsigned int형 Morton code로 변경하는 기존의 방법을 논의하고 그 기하학적인 의미를 분석함으로써, 보다 높은 해상도를 구현할 수 있는 64비트 unsigned long long형의 Morton code 변환 알고리듬을 제안한다. 제안하는 알고리듬은 GPU에서 구현되었을 때 CPU에서 실행하는 것보다 약 1000배 수준의 성능 향상을 달성한다.

키워드 : CUDA, IEEE 754, 64 비트 모튼 코드 kANN, GPU

Analysis of Morton Code Conversion for 32 Bit IEEE 754 Floating Point Variables

Taejung Park*

Abstract

Morton codes play important roles in many parallel GPU applications for the nearest neighbor (NN) search in huge data and queries with its applications growing. This paper discusses and analyzes the meaning of Tero Karras's 32-bit 'unsigned int' Morton code algorithm for three-dimensional spatial information in $[0,1]^3$ and its geometric implications. Based on this, this paper proposes 64-bit 'unsigned long long' version of Morton code and compares the results in both CPU vs. GPU and 32-bit vs. 64-bit versions. The proposed GPU algorithm runs around 1000 times faster than the CPU version.

Keywords : CUDA, IEEE 754, 64-bit Morton code, kANN, GPU

1. 서론

1.1 연구 배경

최근 그래픽 처리 프로세서(GPU)의 용도가 2차원 또는 3차원 그래픽 정보의 처리를 수행하는 용도를 넘어서 GPGPU (General-Purpose computing on Graphics Processing Units)로 확장됨에 따라 CPU 몇 개 코어만으로는 불가능했던 대규모 병렬 연산으로 인해 과거에는 상상하기 힘들었던 새로운 결과가 실현되고 있다. 이러한 사례로는 과거에는 실시간 연산이 불가능하다고 알려 졌던 ray-tracing의 실시간 구현[1], 여러 물리 시뮬레이션의 실시간화 또는 고속화 등이 있으며 특히 주목을 받고 있는 분야는 오랜 기간 동안 연구 측면에서 침체를

※ Corresponding Author : Taejung Park
Received : May 09, 2016
Revised : June 24, 2016
Accepted : June 29, 2016
* Dept. of Digital Media, Duksung Women's University
Tel: +82-2-901-8339, Fax: +82-2-901-8646
email: tjpark@duksung.ac.kr
▣ 이 논문은 2015년도 정부(교육부)의 재원으로 한국연구재단의 지원을 받아 수행된 기초연구사업임(NRF-2013R1A1A2064147).

겪어 오다가 GPGPU의 등장으로 deep learning을 통해 새로운 전기를 맞고 있는 인공 지능 분야라고 할 수 있다.

패턴 인식, 공간 탐색 등에서 대규모 데이터의 병렬 처리, 특히 최인접(the nearest neighbor) 데이터 요소의 대규모 병렬 검색 목적으로 Morton code[2], Hilbert curve[3] 등 space filling 방법이 병렬 처리 맥락에서 새롭게 조명되고 있다. 단일 CPU 코어 기반에서 최인접 데이터를 찾기 위한 목적으로 사용하는 kd-tree 등과 같은 공간 탐색 자료 구조는 SIMT(Single Instruction Multiple Threads) 구조의 GPU에 적용할 경우 warp divergence [4]와 같은 현상으로 인해 성능 저하가 불가피한 단점이 있다. 이러한 문제에 대한 대안으로, Shengren Li et. al[5]이 제안한 space filling curve 기반 최인접 검색 기법은 warp divergence로 인한 성능 저하 없이 대규모 데이터에 대한 병렬 kANN(k Approximate Nearest Neighbors) 문제를 해결할 수 있다. 따라서 Morton code와 같은 space-filling curve의 효율적인 구현이 점차 더 중요해지고 있다.

1.2 연구의 목적

앞서 논의한 Morton code의 중요성에도 불구하고 현재 공개된 대부분의 Morton code 변환 알고리듬은 32비트 unsigned int 변수형(또는 64비트 확장형)에 기초한 bitwise 연산을 이용한다. 그러나 unsigned int 변수형에 기초한 알고리듬은 실제 공간 정보에서의 여러 문제들을 다루기에는 적합하지 않으며 이 경우에는 실수 변수형 데이터를 Morton code로 직접 변환할 수 있는 방법이 필요하다. Tero Karras는 이 문제를 해결하기 위해서 각각 0.0f와 1.0f 사이의 32비트 float형 x, y, z 변수(즉, $[0,1]^3$ 공간 내의 임의의 점의 좌표)에서 Most Significant Bit(MSB)부터 10비트만을 취하고 나머지는 버림으로써 총 30비트(10 x 3)를 조합해서 32비트 unsigned int로 Morton code로 출력하는 코드를 제시했다[6]. 그러나 이 방식에서는 제시된 bitwise 연산들에 대한 설명 없이 이 연산들에 사용되는 값들이 magic

<표 1> 32비트 unsigned int 형 Morton code 생성 코드[6]

```
// Expands a 10-bit integer into 30 bits
// by inserting 2 zeros after each bit.
unsigned int expandBits(unsigned int v)
{
    v =(v * 0x00010001u) & 0xFF0000FFu;  // Ⓗ
    v =(v * 0x00000101u) & 0x0F00F00Fu;  // Ⓘ
    v =(v * 0x00000011u) & 0xC30C30C3u;  // Ⓙ
    v =(v * 0x00000005u) & 0x49249249u;  // Ⓚ
    return v;
}

// Calculates a 30-bit Morton code for the
// given 3D point located within the unit cube [0,1].
unsigned int morton3D(float x, float y, float z)
{
    x = min( max(x*1024.0f, 0.0f), 1023.0f); // Ⓐ
    y = min( max(y*1024.0f, 0.0f), 1023.0f); // Ⓑ
    z = min( max(z*1024.0f, 0.0f), 1023.0f); // Ⓒ
    unsigned int xx = expandBits((unsigned int)x); // Ⓓ
    unsigned int yy = expandBits((unsigned int)y); // Ⓔ
    unsigned int zz = expandBits((unsigned int)z); // Ⓕ
    return xx *4+ yy *2+ zz; // Ⓖ
}
```

<Table 1> Code for generating 32-bit unsigned int Morton codes

number로 명시되었고 원래 32비트 float형이었던 각 축의 값을 10비트만 취함으로써(즉, 각 축당 22비트를 버림으로써 전체적으로는 66비트를 버림) 상당한 정보 손실이 발생하는 문제가 있다.

따라서 x, y, z 각 좌표값이 원래의 32비트 정보(혹은 double형일 경우 64비트)를 보다 많이 유지하는 정밀한 Morton code 알고리듬을 구현하기 위해서는 1) Tero Karras의 bitwise 연산의 의미를 정확하게 이해해야 하고 2) Tero Karras가 제시한 알고리듬으로 생성된 Morton code가 shifted sort[5] 같은 병렬 nearest neighbor 알고리듬에 적용하기 위해, 공간 상에서 실제 근접성을 가지고 있는지, 기하학적인 특성을 검토해야 한다.

본 논문에서는 이 두 가지 문제를 고찰하고 실제 CUDA 환경에서 구현한 64비트 Morton code 코드를 제시한다.

2. Morton Code의 분석 및 확장

2.1 기존 Morton Code의 분석

<표 1>에서는 Tero Karras가 제시한 코드를 정리한다. 이 코드는 $[0,1]^3$ 공간 내에 위치

하는 점 (x, y, z)를 세 개의 32비트 float형으로 나타낸 후 각 축마다 10비트를 취해서 32비트 unsigned int형으로 Morton code를 생성하는 함수(morton3D)의 실제 C/C+ 코드이다.

<표 1>에서 제시한 Tero Karras의 코드를 이해하기 위해서는 IEEE 754 float 변수 형식[7]을 이해할 필요가 있다.

이 표준에 의하면 32비트 float 변수는 부호를 표현하기 위한 1비트(sign bit), 지수(exponent)를 표현하는 8비트. 가수(fraction)를 표현하는 23비트로 구성된다. 본 논문에서 다루고자 하는 좌표의 범위는 $[0,1]^3$ 공간 내 실수 좌표값으로 한정하기 때문에 각 float 형의 좌표값은 다음과 같은 특징을 가진다.

- sign bit == 0, (∵ x, y, z ≥0)

Sign bit의 경우 위에서 명시한 대로 각 좌표값이 0 이상이기 때문에 항상 0으로 설정된다. fraction bits의 경우는 0과 1사이의 소수값을 2진수로 나타내는 일반적인 과정(그림 1)을 통해 결정되는데 특정한 경우를 제외하고는 일반적으로 무한하게 비트 시퀀스가 확장될 수 있으며 그중 23비트만 취하게 된다. 이 비트 시퀀스의 의미에 대해서는 2.2절에서 자세하게 분석한다. 예를 들어 0과 1사이의 소수 $0.672_{(10)}$를 2진수로 나타내면 다음과 같다.

$0.672_{(10)} = 0.10101100000001000001100011..._{(2)}$

또한 float형으로 정의된 변수에 0.672의 float값으로 0.672f을 대입하면 다음과 같은 비트 패턴을 가진다(편의상 sign, exponent, fraction을 빈칸으로 구분).

→ 0 01111110 01011000000100000110001

<표 1>에 제시한 코드에서 Ⓐ, Ⓑ, Ⓒ로 표시한 라인에서는 입력받은 float형 변수 x, y, z에 먼저 1024.0f를 곱한다. int형 변수의 연산이었다면 int형 변수가 가진 값을 MSB(most significant bit) 쪽으로 10번 shift하는 연산(즉, a가 int형이라고 할 때 a * 1024 == a << 10)으로 쉽게 이해할 수 있으나 <표 1>의 코드 연산은 float형 연산이기 때문에 단순히 bitwise shift 연산이라고 단정하기는 어렵다. 위에서 제시한 예(0.672f)를 적용해서 실제 코드로 구현한 후 값을 확인해 보면 다음과 같다.

0.672f * 1024.0f = 688.127991f
→ 0 10001000 01011000000100000110001

이 결과를 위에서 제시한 0.672f의 비트 패턴과 비교해 보면, 1024.0f를 곱한 후에는 sign, fraction 부분은 변화가 없고 exponent 부분만 변경되었음을 볼 수 있다. exponent 값은 원래 exponent 비트 시퀀스(즉, 01111110)를 2진수로 간주하고 여기에 $\log_2 1024.0f$=10을 더한 값이 된다는 사실을 알 수 있다. 즉,

0.672f의 exponent : $01111110_{(2)} = 126_{(10)}$
$126_{(10)}+10_{(10)} = 136_{(10)} = 10001000_{(2)}$

따라서 float 연산에서는 int형에서처럼, 2^n을 곱했을 때($n \in Z$), 단순한 bitwise shift 연산이 아니라 exponent 비트 패턴이 정수처럼 해석되고 덧셈/뺄셈 연산을 거쳐서 변환된다는 사실을 확인할 수 있다.

<표 1> 코드의 Ⓓ, Ⓔ, Ⓕ 라인에서는 그전까지 계산한 float형 변수 x, y, z를 unsigned int형으로 type casting한 후 expandBits( ) 함수로 전달함을 볼 수 있다. float형 변수를 int 혹은 int unsigned형으로 type casting할 경우, 소수 부분을 버림 연산으로 제거한다는 사실은 널리 알려져 있다. 예를 들어 0.672f * 1024.0f = 688.127991f를 unsigned int형으로 type casting하면 그 결과는 다음과 같다.

(unsigned int) 0.672f * 1024.0f = 688u

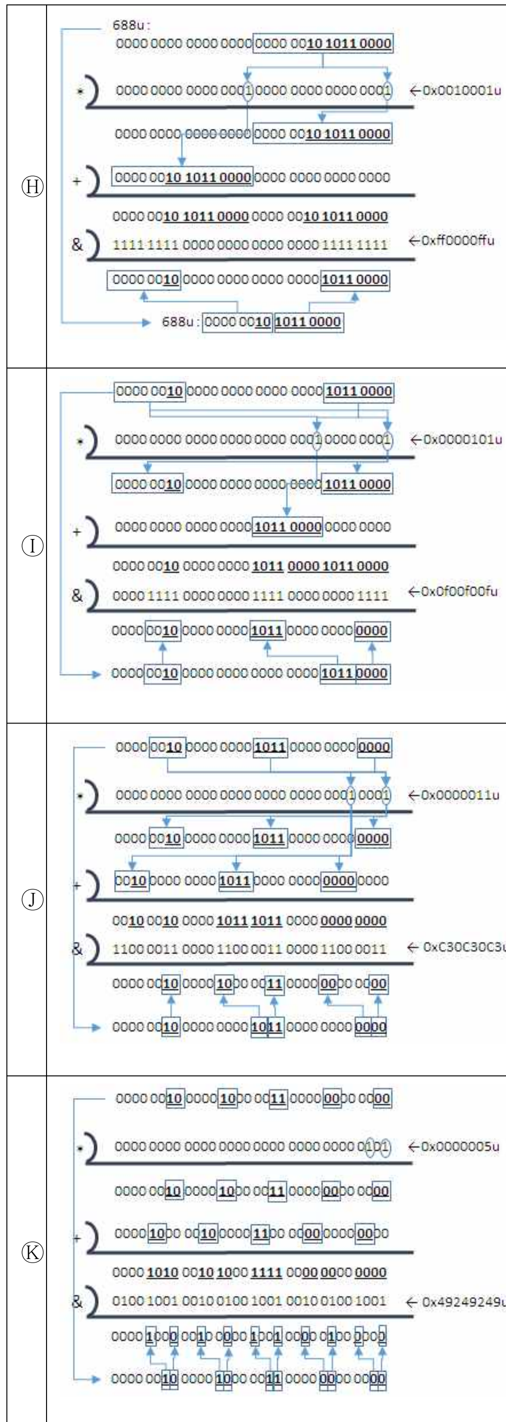
이 때 688u의 비트 패턴은 다음과 같다.
00000000000000000000001010110000

이 값을 원래 값인 $0.672_{(10)}$의 2진수 표현과 비교해 보면 2진수로 표현한 소수점 10자리가 그대로 이 비트 패턴에 반영되어 있음을 볼 수 있다. 즉,

0.672(10)= 0.10101100000001000001100011...(2)

688u : 00000000000000000000001010110000

<표 2> v = 688u일 때 Ⓗ~Ⓚ 루틴 분석



<Table 2> From Ⓗ to Ⓚ when v = 688u

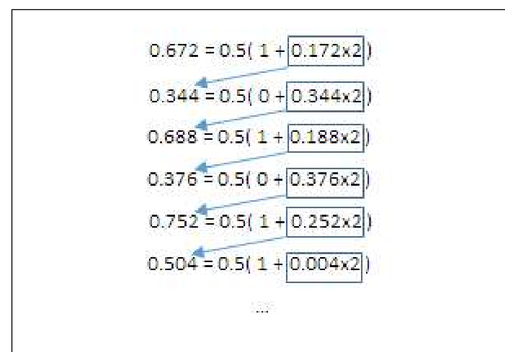
따라서, <표 1> 코드의 Ⓐ, Ⓑ, Ⓒ 라인과 Ⓓ, Ⓔ, Ⓕ 라인에서 expandBits( ) 함수로 전달될 때까지의 내용을 정리하면 min, max로 float 변수 x, y, z의 범위를 한정하고, 2진수로 나타낸 소수를 앞에서부터 10개($\log_2 1024.0f=10$) 복사해서 unsigned int 변수에 비트 패턴으로 보관하는 작업을 수행한다.

<표 2>에서는 v = 688u일 때 Ⓗ~Ⓚ 루틴의 실행 중간 결과를 분석, 정리한다.

이 루틴에서 변수 v에 곱하는 16진수 숫자(0x00010001u, 0x00000101u, 0x00000011u, 0x00000005u)는 변수 v 자신을 각각 16칸, 8칸, 4칸, 2칸 왼쪽으로 bitwise 이동(shift)시킨 값과 원래 값을 더하는 연산을 수행한다.

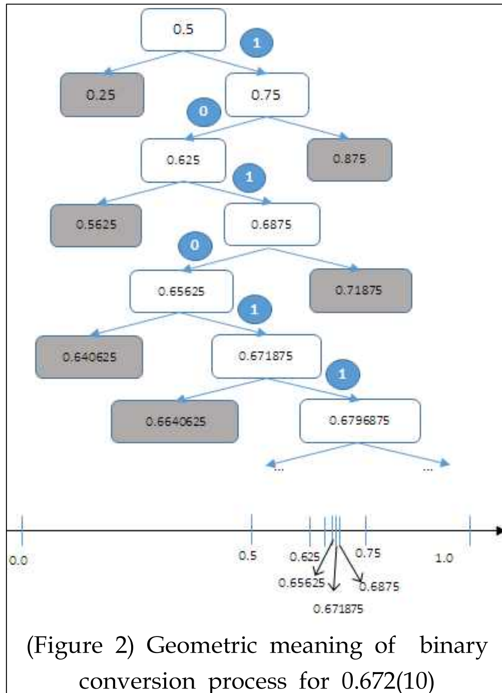
그리고 bitwise AND 연산(&)을 수행하는 16진수 숫자(0xFF0000FFu, 0x0F00F00Fu, 0xC30C30C3u , 0x49249249u)는 <표 2>에서 볼 수 있는 것처럼, 1의 개수를 이용해서 너비가 8, 4, 2, 1인 필터링 역할을 수행한다. 최종 단계인 Ⓚ에서 볼 수 있는 것처럼, 결국 입력 비트 패턴에 0 비트 두 개 (즉, “00”)를 삽입(padding)한 결과를 얻게 된다. 이렇게 x, y, z 좌표값을 각각 계산하고 그 결과를 unsigned int형 xx, yy, zz 좌표에 저장한 후 (Ⓓ, Ⓔ, Ⓕ) x 좌표값은 4, y 좌표값은 2를 곱해서 각각 2비트, 1비트 왼쪽으로 shift 시킨 후 x, y, z 좌표값을 더해서([6의 ‘Exploiting the Z-Order Curve’ 항목 그림 참고) Morton code를 완성한다.

(그림 1) 0.672(10)의 2진수 계산 과정



(Figure 1) Conversion process for 0.672(10) to a binary number

(그림 2) 0.672(10)의 2진수 계산 과정의 기하학적 의미



(Figure 2) Geometric meaning of binary conversion process for 0.672(10)

2.2 float 기반 3차원 정보 기반 Morton Code의 기하학적인 의미

2.1절에서는 Tero Karras가 제안한 $[0,1]^3$ 공간 내로 scale된 float형 변수 기반 3차원 공간 정보 (x, y, z)를 Morton code로 표현하는 알고리듬을 살펴보고 원래 문서[6]에서 설명하지 않은 작동 원리를 분석했다. 결과적으로 <표 1>에서 제시된 알고리듬은 논의한 과정을 거쳐서 0과 1사이의 소수로 표현되는 각 좌표값을 2진수로 나타내고 각 좌표마다 10비트씩을 읽어서 xyzxyz... 순서로 총 30비트를 구한 후 32비트 unsigned int 형 변수로 출력한다.

그러나 이 Morton code 알고리듬을 shifted sort[5] 같은 병렬 nearest neighbor 알고리듬이나 병렬 공간 분할/탐색 문제[8]에 적용하기 위해서는 기하학적인 의미를 검토할 필요가 있다.

본 논문에서 다루고 있는 Morton code 알고리듬의 기하학적인 의미는 [0,1] 구간 내의 실수를 2진수로 변환하는 과정에서 찾을 수 있다. 2.1절에서 제시한 예제에서처럼 10진수 0.672는 (그림 1)의 과정을 거쳐 다음과 같이 2진수로 표현할 수 있다.

$$0.672_{(10)} = 0.10101100000001000001100011..._{(2)}$$
$$=0 \cdot 2^{0}+1 \cdot 2^{-1}+0 \cdot 2^{-2}+1 \cdot 2^{-3}+0 \cdot 2^{-4}+1 \cdot 2^{-5}+...$$

이 과정은 (그림 2)에서 제시한 것처럼, 이진 트리로 표현할 수 있다. 즉, 0과 1사이의 실수 x에 대해 이 실수 x가 항상 포함되는 초기 구간 I_0=[0,1]을 생각한다. 이때, n번째 구간을 I_n이라고 하고 이 구간을 가운데 값을 기준으로 절반으로 나누면, 왼쪽과 오른쪽의 두 개 구간으로 나눌 수 있고 그 중 실수 x가 포함되는 구간을 I_{n+1}이라고 하자. 만일 실수 x가 I_n의 왼쪽에 위치할 경우 0을 출력하고 오른쪽에 위치할 경우 1을 출력하면 이 구간이 분할되면서 출력되는 0/1로 구성되는 수열을 얻을 수 있는데 이 수열은 실수 x의 2진수값임을 알 수 있다. 특히 구간 I_n의 중간값에 해당되는 트리 노드를 생성하고 왼쪽 자식 노드는 다음 번 분할했을 때의 왼쪽 구간, 오른쪽 자식 노드는 오른쪽 구간을 할당하면 (그림 2)에서 제시한 것과 같은 이진 트리를 구성할 수 있다.

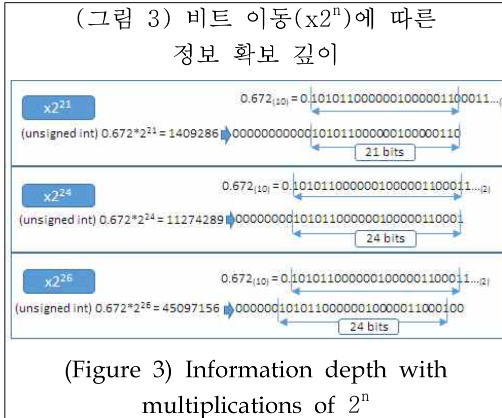
이러한 관찰을 통해서 (그림 1)에서 제시한 구간 [0,1] 내의 실수를 2진수로 변환하는 과정은 이 구간에 대한 2진 트리 분할을 수행하는 과정이라고 결론을 내릴 수 있다. 따라서 <표 1>에서 살펴 본 것처럼, xyz 순서대로 이 정보를 padding 형태로 표시하면 3차원 폐공간 $[0,1]^3$에서 Octree 분할[9]을 수행하는 것과 동일하다. 따라서 <표 1>에서 제시한 코드로 3차원 점에 대한 Morton code를 생성하고 shifted sort[5]를 수행하면 MSB(Most Significant Bit) 쪽의 비트 패턴이 동일하고 LSB(Least Significant Bit)의 비트가 다르면 공간상에서 Morton code의 traversal 순서에 대해 인접해 있다고 결론을 내릴 수 있다.

3. 구현

3.1 32비트 Morton code의 한계

<표 1>에서 제시한 코드는 x, y, z축으로 각각 10비트를 할당해서 총 30비트의 의미있는 정보를 32비트 unsigned int형 변수에 저장한

(그림 3) 비트 이동($x2^n$)에 따른 정보 확보 깊이



(Figure 3) Information depth with multiplications of 2^n

다. 따라서 각 축 당 [0, 1] 범위를 최대 1024(==2^{10})개로 분할하며 구분 가능한 최소 간격은 2^{-10}=0.000977에 불과하다. 이 정도의 해상도는 임의의 3차원 공간을 $[0,1]^3$ 단위 공간 내로 scale하는 상황임을 고려해 보면 의미있는 애플리케이션을 구현하기에는 매우 부족한 수준이라고 할 수 있다. 따라서 본 논문에서는 지금까지 파악한 내용을 바탕으로 각 축 당 21비트를 할당하고 그 결과로 얻은 Morton code를 64비트 int에 저장하는 알고리듬을 구현하고 그 결과를 비교한다. 각 축 당 21비트를 할당할 경우 각 축 당 [0, 1] 범위를 최대 2097152(==2^{21})개로 분할하며 구분 가능한 최소 간격은 2^{-21}=4.77x10^{-7}으로 일반적인 3차원 공간 탐색 및 그래픽 구현을 위한 해상도로 충분한 수준으로 향상된다.

<표 3> 64 비트 Morton code 생성 코드

```
// 64-bit version.
unsigned long long expandBits64(unsigned long long v)
{
    v =(v * 0x0000000100000001ui64)
      & 0xFFFF00000000FFFFui64;
    v =(v * 0x0000000000010001ui64)
      & 0x00FF0000FF0000FFui64;
    v =(v * 0x0000000000000101ui64)
      & 0xF00F00F00F00F00Fui64;
    v =(v * 0x0000000000000011ui64)
      & 0x30C30C30C30C30C3ui64;
    v =(v * 0x0000000000000005ui64)
      & 0x1249249249249249ui64;

    return v;
}

// Calculates a 63-bit Morton code for the
// given 3D point located within the unit cube [0,1].
unsigned long long morton3D64(float x, float y, float z)
{
    x = min( max(x*2097152.0f, 0.0f), 2097151.0f);
    y = min( max(y*2097152.0f, 0.0f), 2097151.0f);
    z = min( max(z*2097152.0f, 0.0f), 2097151.0f);
    unsigned long long xx
          = expandBits64((unsigned long long)x);
    unsigned long long  yy
          = expandBits64((unsigned long long)y);
    unsigned long long zz
          = expandBits64((unsigned long long)z);
    return xx *4+ yy *2+ zz;
}
```

<Table 3> Code for generating 64-bit Morton codes

<표 4> $[0,1]^3$ 공간 내 무작위 3차원 점 324437 개에 대한 Morton code 생성 시간 비교 (단위 μs)

	32bits	64bits
GPU	4.0	5.0
CPU	3828.0	5186.0

<Table 4> Time comparison to generate 324437 random points in $[0,1]^3$ space

3.2 64비트 Morton code의 구현

2.1절에서 파악한 원리를 응용하면, 원하는 개수의 비트만큼 unsigned int형 변수에 기록할 수 있다. 예를 들어 20개 비트를 저장하려면,

(unsigned int) (0.672f * 2^{20}) = 704643u
0.672(10) = 0.10101100000010000011100011...(2)
704643u:
000000000000101011000000100000 11

(그림 3)에서는 $x2^n$ 연산을 통해서 확보할 수 있는 비트 개수를 정리한다. 그림에서 확인할 수 있듯이 n이 커질수록 [0, 1] 사이의 실수를 2진수로 나타냈을 때 얻게 되는 비트 개수가 많아지지만, float형 변수의 IEEE 754 표준에 의해서 fraction 비트 길이가 23비트이고 소수의 경우 왼쪽으로 한 칸 이동시키는 방식[7, 10] 때문에 최대 24비트까지 확보 가능하다. 또한 이러한 특징 때문에 (그림 3)의 가장 아래 예에서 볼 수 있듯이 2^{26}을 곱해서 26비트를 확보하려고 해도 최대 24비트까지만 의미있는 숫자로 확보 가능하고 뒤쪽에 0이 남은 비트만큼 들어오는 사실을 확인할 수 있다.

이러한 관찰을 토대로 64비트 버전 확장을

위해 x, y, z축 각각 21비트씩 확보해서 <표 1>에서 제시한 코드와 유사하게 총 63비트를 순차적으로 구성한다. 나머지 1개 비트는 애플리케이션별로 추가 정보를 기록하는 용도(예를 들어 data point와 query point의 구분 등)로 활용한다. 최종 코드는 <표 3>에서 제시한다.

3.3 실행 시간 비교

논의한 방식의 성능을 확인하기 위해서 [0,1]3 공간 내에서 임의의 3차원 점 324437개에 대해 GPU, CPU 환경에서 각각 32비트, 64비트 Morton code 생성 코드를 작성한 후 <표 4>에서 실행 시간 비교를 정리했다. GPU, CPU 모든 경우 생성되는 실행 파일(exe) 코드 자체는 64비트 코드로 컴파일했으며 GPU의 경우 NVIDIA GTX TITAN X에서 block 당 thread 개수는 1024개에서 실행했으며 CPU의 경우 I7-4790 CPU (3.6GHz), RAM 16GB, Windows 7 (64bits) 환경에서 실행했다.

GPU 코드의 경우, NIVIDA CUDA kernel을 구현하고 실행 시간 측정을 위해서 host 측에서 cudaDeviceSynchronize() 함수[4로 동기화를 했으며 kernel의 초기 실행 시 발생하는 overhead를 제거하기 위해서 warmup 루틴[4]을 먼저 실행한 후 시간을 측정하였다. GPU의 병렬 처리 특성 상 실행 시간 결과의 변동폭이 큰 특성이 있기 때문에 <표 4>에서 제시된 시간은 10 회 수행 후 평균값으로 제시되었다.

CPU와 GPU를 비교하면, 동일한 결과를 얻기 위해 32/64비트 Morton 코드에서 모두 약 1000배 정도의 성능 향상이 있음을 확인할 수 있다. 또한 CPU와 GPU 모두, 32비트보다 64비트 Morton code 생성 시에 조금 더 시간이 소요된다는 사실을 확인할 수 있다.

4. 결론 및 논의

1.2절에서 언급한 대로 본 논문에서 1) Tero Karras의 bitwise 연산의 의미를 분석하고 2) Tero Karras가 제시한 알고리듬으로 생성된 Morton code가 공간 상에서 실제 근접성을 가지는 기하학적인 특성이 있는지 두 가지 문제에 대한 논의와 분석을 진행했다. 먼저 IEEE 754 float 형식 표준을 통해 <표 1>을 통해 제시된 코드 내의 type casting의 의미를 고찰하고 bitwise 연산을 분석함으로써 float 유형의 값을 64비트 int (unsigned long long) 유형으로 변경하기 위한 기본 연산의 의미를 파악하였다. 그리고 <표 1>에서 제시한 알고리듬의 의미를 기하학적으로 분석하고 그 결과 Octree 분할 과정과 동일하다는 결론을 도출하였다. 마지막으로 GPU로 64비트 Morton code를 실제로 구현한 결과를 제시하였는데 Tero Karras의 32비트 Morton code와 성능 차이가 크지 않은 결과를 얻었다<표 4>. 제안하는 64비트 Morton code는 보다 해상도가 개선된 3차원 기하 애플리케이션에서 유용하게 사용될 수 있을 것으로 기대한다.

References

[1] NVIDIA OptiX, https://developer.nvidia.com/optix

[2] https://en.wikipedia.org/wiki/Z-order_curve

[3] https://en.wikipedia.org/wiki/Hilbert_curve

[4] John Cheng, Max Grossman, and Ty McKercher, “Professional CUDA C Programming”, p. 84, 1st Edition, Wrox, 2014.

[5] Shengren Li, Lance Simons, Jagadeesh Bhaskar Pakaravoor, Fatemeh Abbasinejad, John D. Owens and Nina Amenta, “kANN on the GPU with Shifted Sorting”, In Proceedings of Eurographics/ACM SIGGRAPH Symposium on High Performance Graphics, pp. 039-047, 2012.

[6] https://devblogs.nvidia.com/parallelforall/thinking-parallel-part-iii-tree-construction-gpu/

[7] https://en.wikipedia.org/wiki/IEEE_floating_point

[8] Tero Karras and Timo Aila, “Fast Parallel Construction of High-quality Bounding Volume Hierarchies”, In Proceedings of the 5th High-Performance Graphics Conference, pp.89-99, 2013.

[9] https://en.wikipedia.org/wiki/Octree

[10] https://ko.wikipedia.org/wiki/IEEE_754

박 태 정

1997년 : 서울대 전기공학부 (공학사)
1999년 : 서울대 전기공학부 대학원(공학석사 반도체 전공)
2006년 : 서울대 전기컴퓨터공학부 대학원 (공학박사, 컴퓨터 그래픽스 전공)
2006년~2013년: 고려대학교 연구교수
2013년~현재 : 덕성여자대학교 디지털미디어학과 조교수
관심분야 : 컴퓨터그래픽스, 병렬처리, 게임 물리, 수치해석, 3차원 모델링