CUDA를 게임 프로젝트에 적용하기

유영천

https://megayuchi.com

tw: @dgtman

GPGPU(General-Purpose computing on GPU

- GPU를 사용하여 CPU가 전통적으로 취급했던 응용 프로그램들의 계산을 수행하는 기술
- GPU 코어 1개의 효율은 CPU 코어 1개에 비해 많이 떨어지지만 코어의 개수가 엄청나게 많다.
- 많은 수의 코어를 사용하면 산술술 연산 성능(Throughput)은 CPU의 성능보다 훨씬 뛰어나다.

GPGPU의 특징

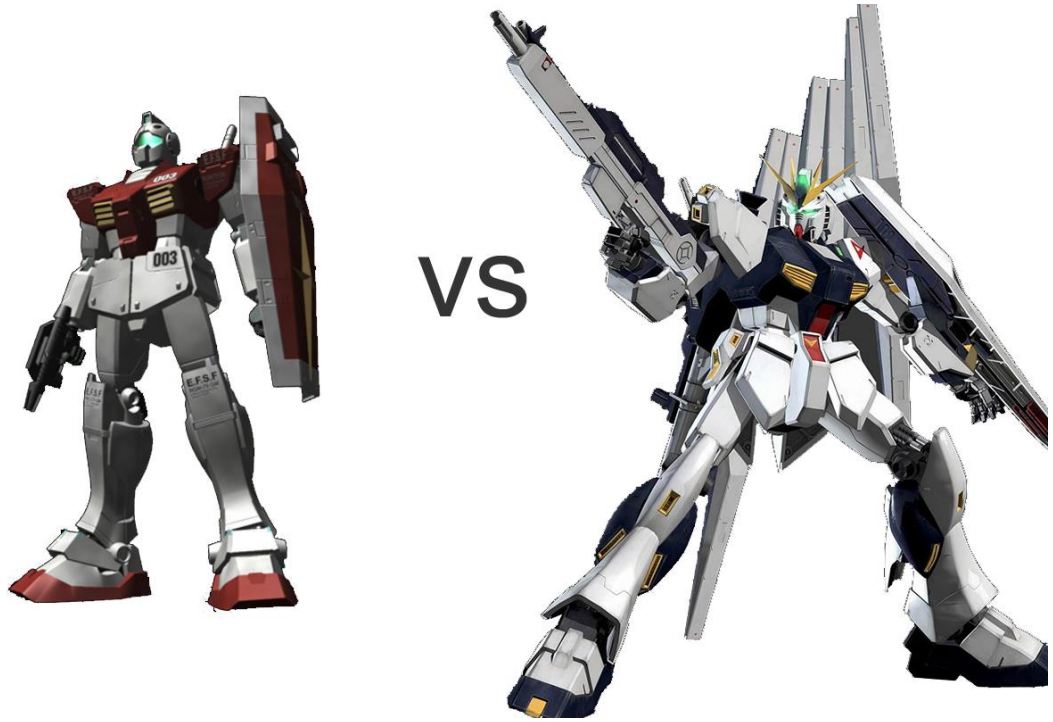
강점

- 엄청난 수의 스레드를 사용할 수 있다.
- 부동소수점 연산이 엄청 빠르다.
- 컨텍스트 스위칭이 엄청 빠르다.

약점

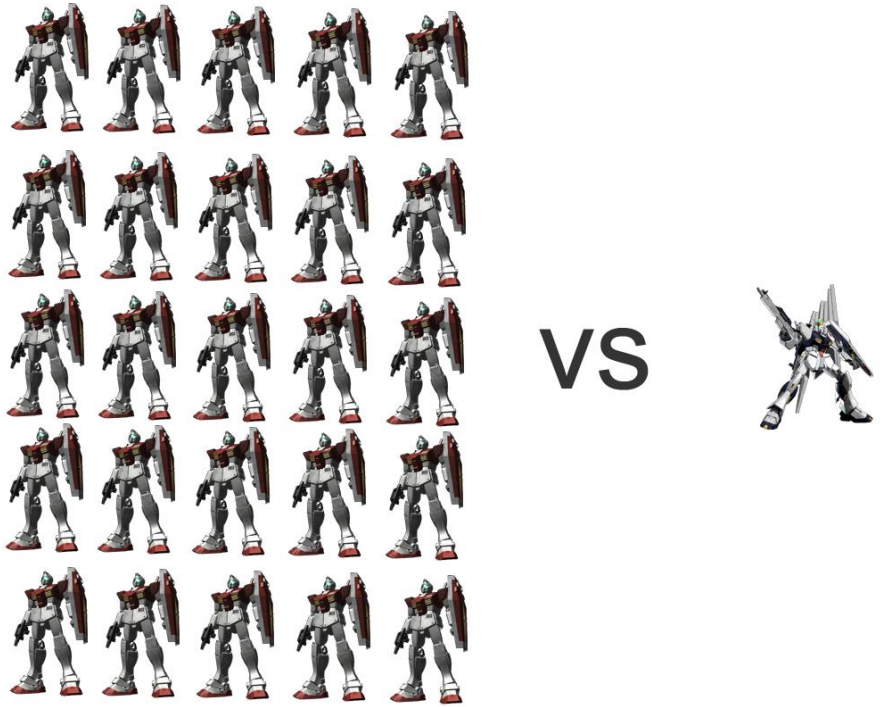
- 흐름제어 기능 자체가 빈약하다.
- 프로그래밍 기법의 제약(재귀호출 등)
- Core당 클럭이 CPU에 비해 많이 느리다(1/3 – 1/2수준)

GPU vs CPU



GM < ν Gundam - 택도없는 승부. 이것은 학살.

GPU vs CPU

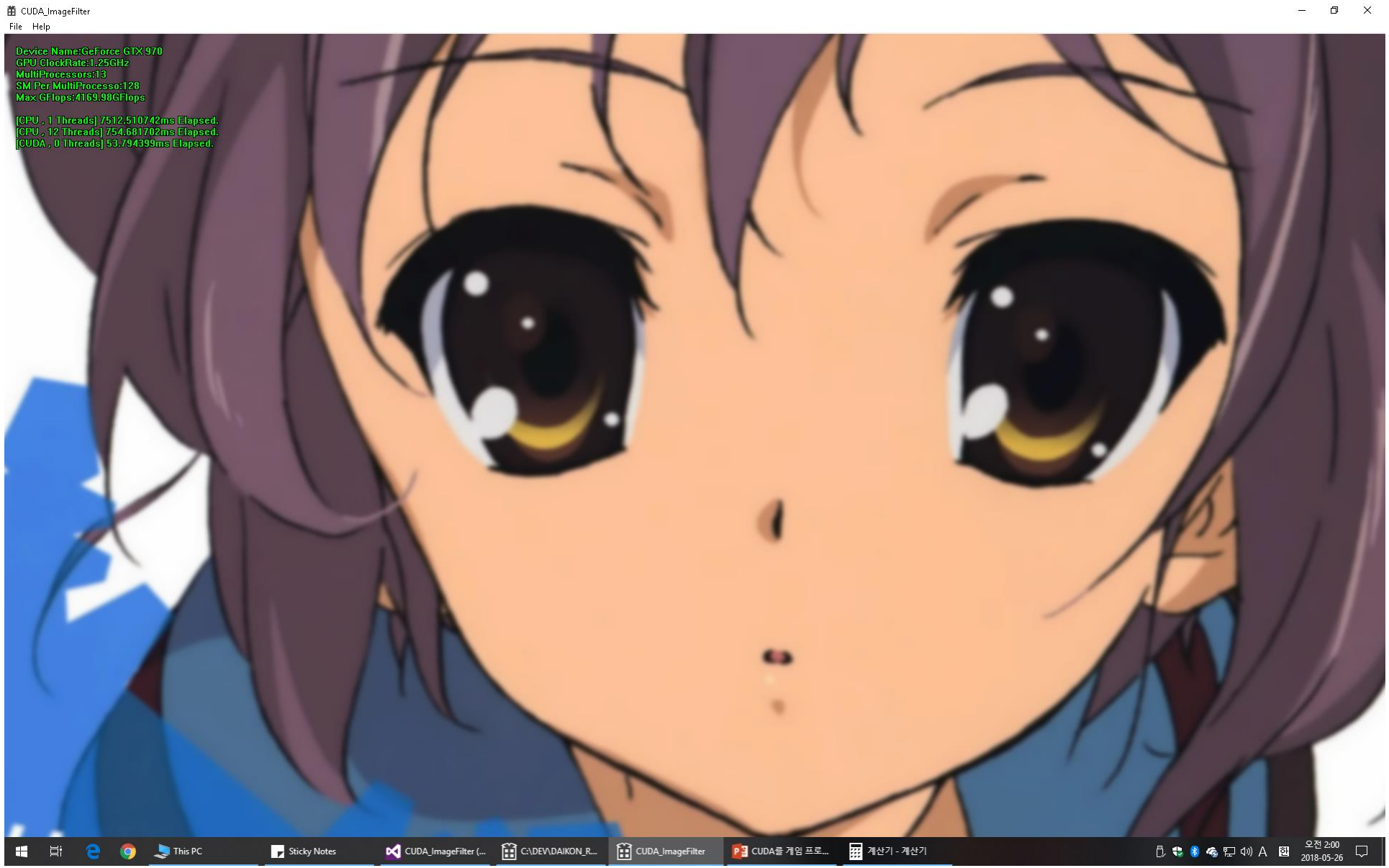


GM > ν Gundam – 한판 붙자!!!

적용분야

- Physics Simulation
- Video Processing
- Image Processing
- Astrophysics
- Medical Imaging
- More…

Image Processing 예제



2885x4102크기의 이미지에 5x5 Gaussian Blur를 3회 적용

성능 비교

1Threads - Intel i7 8700K @4.5GHz

-> 7512.5ms

12Threads - Intel Intel i7 8700K @4.5GHz

-> 754.6ms

CUDA – GTX970 (13 SM x 128 Core = 1664 Core)

-> 53.7ms

참고)

이미지 필터링 코드는 CUDA, CPU 완전 동일.

알고리즘상의 최적화는 없음.

병렬처리 효율의 비교를 위한 테스트임.

사용가능한 S/W제품들

- nvidia CUDA
- Direct X Direct Compute Shader(실질적으로 게임전용)
- ~~Open CL~~
- ~~Microsoft AMP C++~~

CUDA

CUDA (Compute Unified Device Architecture)

- C언어 등 산업 표준 언어를 사용하여 GPU에서 작동하는 병렬처리 코드를 작성할 수 있도록 하는 GPGPU기술
- nVidia가 개발,배포.그래서 nVidia GPU만 가능.
- 비슷한 기술로 OpenCL, Direct Compute Shader가 있음.

CUDA의 장점

- 자료가 많다.
- GPU상에서의 디버깅이 가능하다.
- C/C++(CPU코드)에서의 포팅이 쉽다. 포인터 사용 가능!!!!
- 경쟁 제품들이 다 사망....

CUDA 프로그래밍

용어 및 기본 요소들

GPU구조의 이해 (GP104)

*CPU의 Core에 해당하는 것은 GPU의 SM이다.

따라서 흐름제어의 측면으로 보면 2560 Core CPU가 아니라 20 Core CPU에 상응한다.

Thread

- CPU의 Thread와 비슷…하다(같지 않다).
- GPU의 Core(혹은 SP)에 맵핑
- 독립적인 컨텍스트를 가지고 있다(라고 해봐야 연산에 필요한 컨텍스트 뿐).
- 컨텍스트 스위칭이 빠르다.
- 독립적인 흐름제어는 불가능하다. Warp(32개 스레드) 단위로 묶여서 움직임).

Block

- Block -> N개의 Thread집합. SM에 맵핑
- 동일 Block의 스레드는 L1 cache와 Shared Memory를 공유한다.
- 스레드만 잔뜩 있으면 될것 같은데 왜 이런게 있냐면...그것은 아마도

  CPU의 core !≒ GPU의 core

  CPU의 core ≒ GPU의 SM

  이기 때문일 것이다.

Grid

- Block들의 집합
- 그냥 쉽게 생각해서 그래픽 카드 -> Grid

Thread, Block, Grid

- Thread -> Core
- Block -> SM
- Grid -> GPU

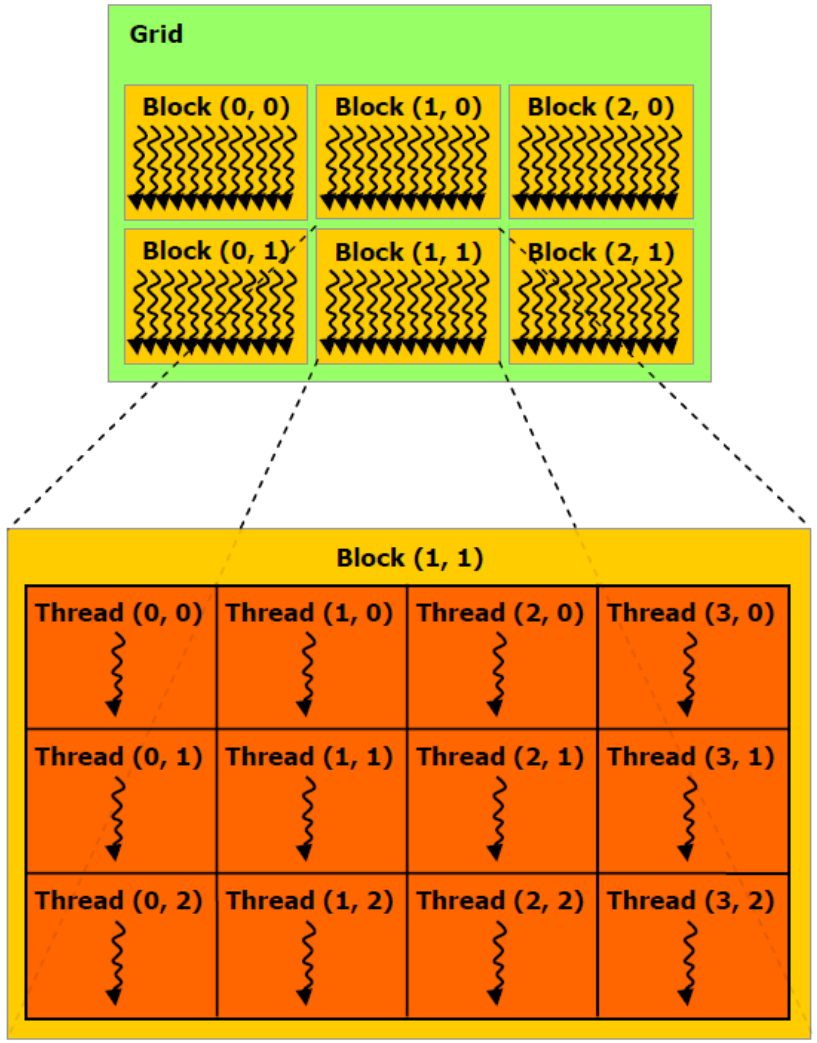


Figure 2-1. Grid of Thread Blocks

Kernel

- GPU에서 돌아가는 C함수
- CPU멀티스레드 프로그래밍에서의 Thread 함수와 같다.
- 복수의 Thread에 의해 실행된다.
- CPU측(host)에서 호출되며 GPU측(device)코드를 호출할 수 있다.

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                       float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];

}

int main()
{
    ...
    // Kernel invocation
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

N x N 매트릭스의 합을 구하는 CUDA커널함수

	CUDA	Compute Shader	대응 H/W
연산 단위	Thread	Thread	SP or CUDA Core
HW점유 단위	Block	Group	SM
Shared Memory	48KB, __shared__	16KB , groupshared	L1 Cache (16KB + 48KB)
Barrier	__syncthreads()	GroupMemoryBarrierWithGroupSync()	
언어	C/C++ ,ptr사용 가능	HLSL, similar to C	

CUDA 프로그래밍

Memory

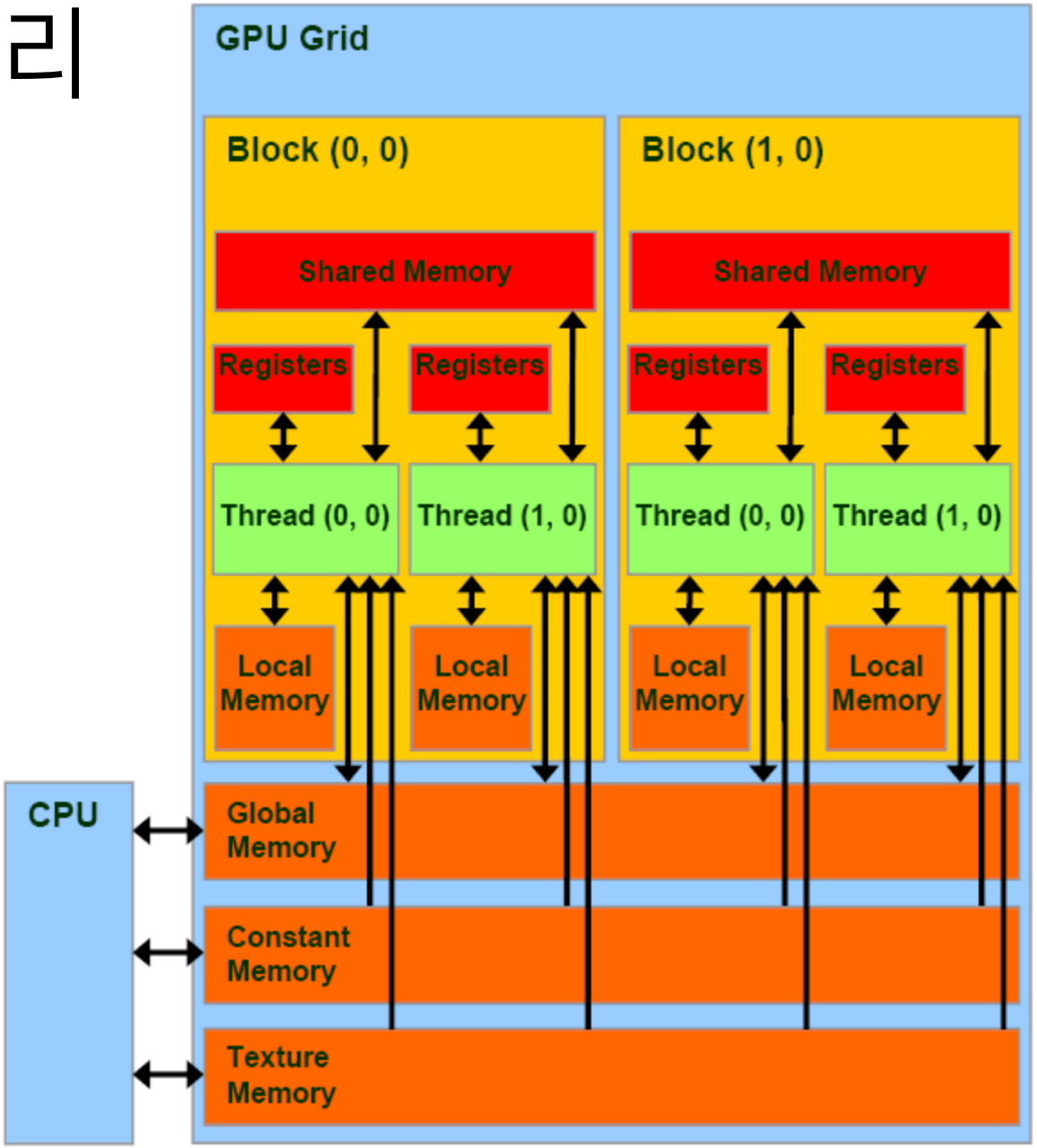
CUDA Memory

- Global Memory – 보통 말하는 Video Memory. 크다. 느리다.
- Register – CPU의 레지스터와 같다. 로컬변수, 인덱싱하지 않고 크기가 작은 로컬배열은 레지스터에 맵핑된다. 작다. 빠르다.
- Shared Memory – Block의 스레드들이 공유하는 메모리. 블럭당 48KB. 작다. 빠르다.
- Constant memory – GPU 하드웨어에 구현된 읽기 전용 64KB 메모리. 물리적으로는 Global Memory로와 같음. 캐싱됨. 상황에 따라 L1 Cache와 동일 속도.
- Texture Memory – 읽기전용. Global Memory위에 구현되지만 Texture로서 읽는 경우 공간지역성을 가짐. 캐싱됨.

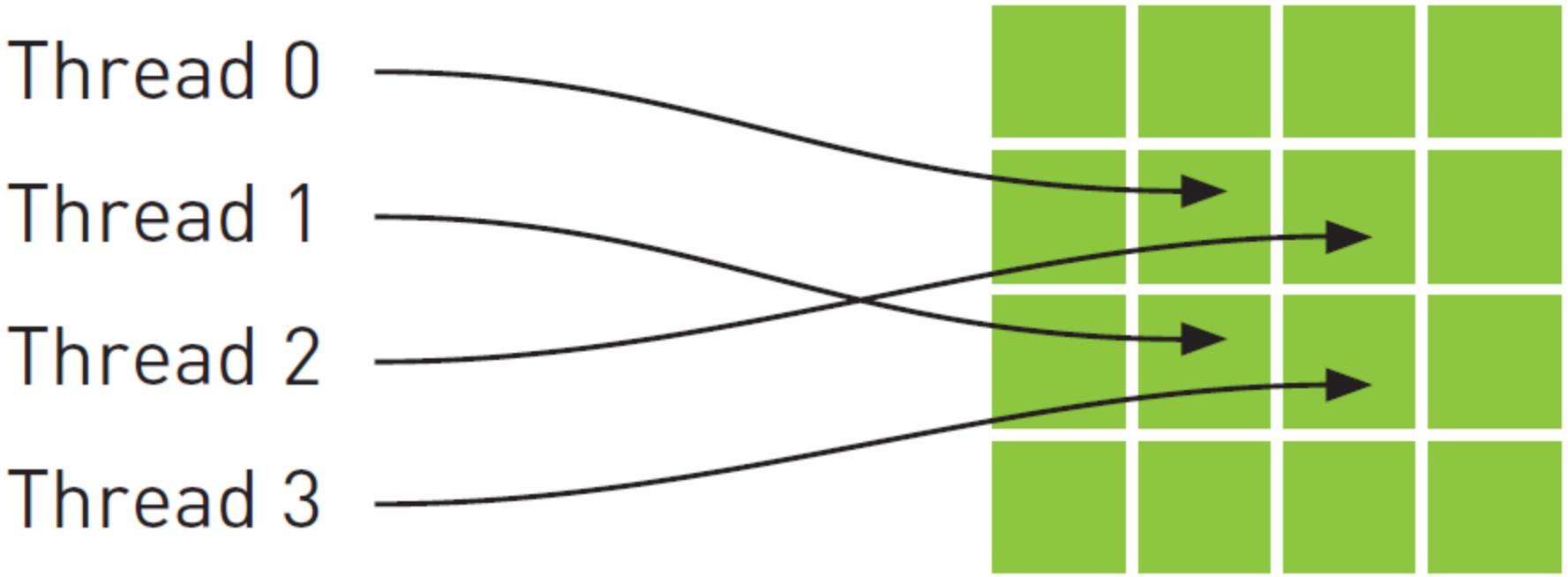
참고)

SM에 하나씩 존재하는 64KB의 cache를 16KB L1캐시와 48KB Shared Memory로 사용

CUDA메모리


GPU Grid
Block (0, 0)
Block (1, 0)
Shared Memory
Shared Memory
Registers
Registers
Registers
Registers
Thread (0, 0)
Thread (1, 0)
Thread (0, 0)
Thread (1, 0)
Local Memory
Local Memory
Local Memory
Local Memory
CPU
Global Memory
Constant Memory
Texture Memory

Texture memory caching



메모리 억세스 범위

- Thread ->
  Local Memory ,
  Shared Memory , Global Memory
- Block ->
  Shared Memory, Global Memory
- Grid -> Global Memory(Launch Kernel)

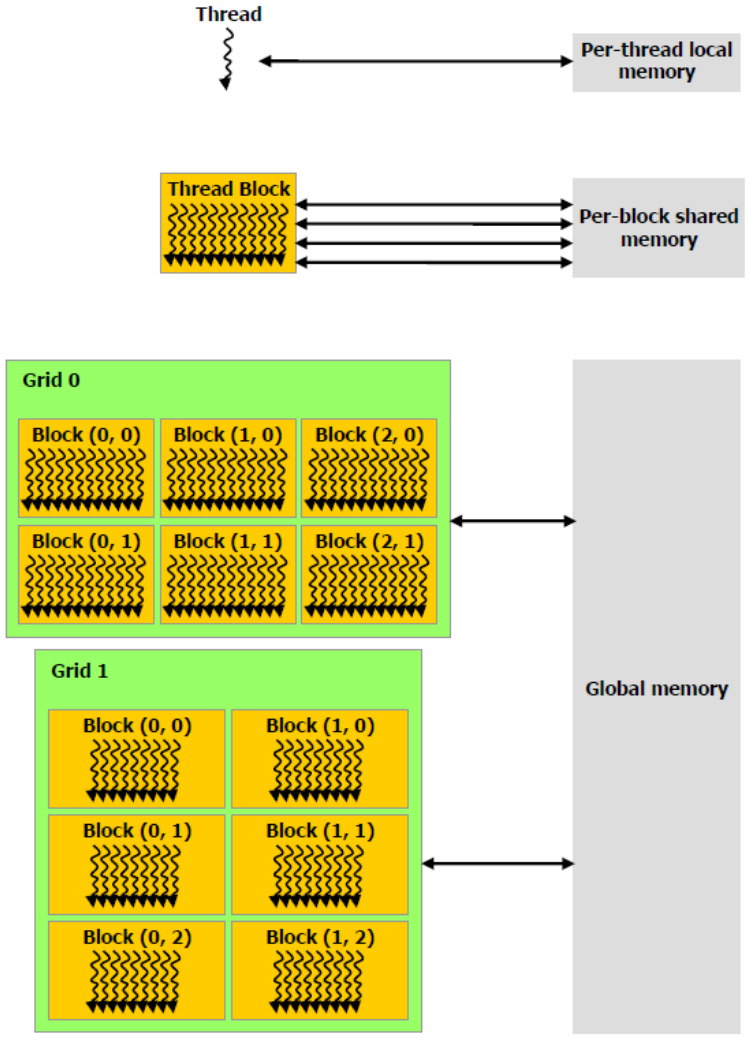


Figure 2-2. Memory Hierarchy

CUDA 프로그래밍

GPU 코드 스케쥴링

SIMT(Single Instruction Multiple Threads)

- 동일 명령어를 여러개의 Thread가 동시 실행
- GPU는 기본적으로 SIMT로 작동
- N차원 벡터를 다루는 산술처리에선 적합
- Thread중 일부가 분기를 하거나 루프를 하면 나머지 Thread들은 대기 -> 병렬처리의 의미가 없어짐.

WARP의 이해

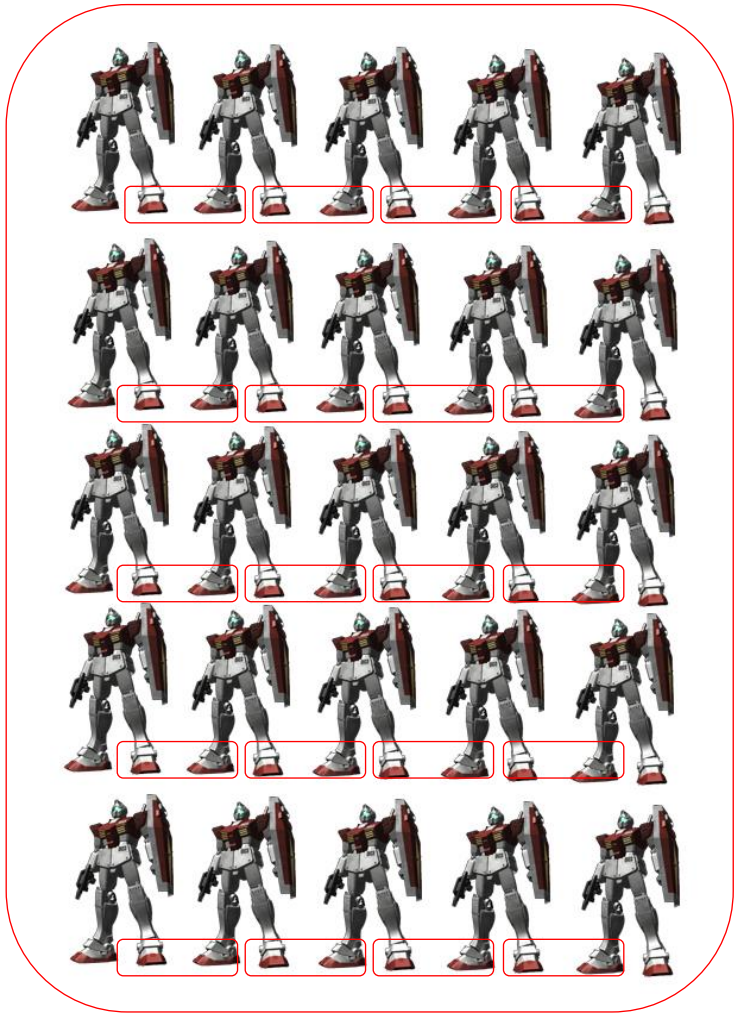
- Thread들을 묶어서 스케쥴링하는 단위
- 현재까지는 모든 nVidia GPU의 1 Warp는 32 Thread. 즉 32 Thread는 늘 같은 같은 코드 어드레스를 지나감.
- 동시 수행 가능한 Warp 수는 SM의 Warp 스케쥴러 개수에 의존. SM당 2-4개의 Warp를 동시에 실행가능

WARP의 이해

- 32 Thread중 1개의 Thread만이 분기해서 Loop를 돌고 있으면 31개의 Thread는 1개의 Thread가 Loop를 마치고 같은 코드 어드레스를 수행할 수 있을 때까지 대기.
- 각 Thread가 다른 루프, 다른 흐름을 타는 것은 엄청난 성능 저하를 부른다.

2인 3각 경기

SM



VS

CPU Core

CUDA 기준 GPU세대별 하드웨어 특성

Compute Capability	2.0	2.1	3.0	3.2	3.5	3.7	5.0	5.2	5.3	6.0	6.1	6.2
SM Version	sm_20	sm_21	sm_30	sm_32	sm_35	sm_37	sm_50	sm_52	sm_53	sm_60	sm_61	sm_62
Threads / Warp	32	32	32	32	32	32	32	32	32	32	32	32
Warps / Multiprocessor	48	48	64	64	64	64	64	64	64	64	64	128
Threads / Multiprocessor	1536	1536	2048	2048	2048	2048	2048	2048	2048	2048	2048	4096
Thread Blocks / Multiprocessor	8	8	16	16	16	16	32	32	32	32	32	32
Shared Memory / Multiprocessor (bytes)	49152	49152	49152	49152	49152	114688	65536	98304	65536	65536	98304	65536
Max Shared Memory / Block (bytes)	49152	49152	49152	49152	49152	49152	49152	49152	49152	49152	49152	49152
Register File Size / Multiprocessor (32-bit registers)	32768	32768	65536	65536	65536	131072	65536	65536	65536	65536	65536	65536
Max Registers / Block	32768	32768	65536	65536	65536	65536	65536	65536	32768	65536	65536	65536
Register Allocation Unit Size	64	64	256	256	256	256	256	256	256	256	256	256
Register Allocation Granularity	warp	warp	warp	warp	warp	warp	warp	warp	warp	warp	warp	warp
Max Registers / Thread	63	63	63	255	255	255	255	255	255	255	255	255
Shared Memory Allocation Unit Size	128	128	256	256	256	256	256	256	256	256	256	256
Warp Allocation Granularity	2	2	4	4	4	4	4	4	4	2	4	4
Max Thread Block Size	1024	1024	1024	1024	1024	1024	1024	1024	1024	1024	1024	1024
Shared Memory Size Configurations (bytes)	49152	49152	49152	49152	49152	114688	65536	98304	65536	65536	98304	65536
[note: default at top of list]	16384	16384	32768	32768	32768	98304	#N/A	#N/A				
	#N/A	#N/A	16384	16384	16384	81920	#N/A	#N/A				
Warp register allocation granularities	64	64	256	256	256	256	256	256	256	256	256	256

CUDA 프로그래밍

코딩

단순포팅예제

```
}
void CPU_Edge_Filter(char* pDest,char* pSrc,DWORD dwWidth,DWORD dwHeight,
{
    if (ThreadIndex >= ThreadNum)
        return;

    DWORD   dwPitch = dwWidth*4;

    DWORD   dwHeightPerThread = dwHeight / ThreadNum;
    DWORD   dwReservedHeight = dwHeight % ThreadNum;

    DWORD   start_y = dwHeightPerThread*ThreadIndex;
    DWORD   end_y = start_y + dwHeightPerThread;

    if (ThreadIndex == (ThreadNum-1))
    {
        end_y += dwReservedHeight;
    }
    for (DWORD y=start_y; y<end_y; y++)
    {
        for (DWORD x=0; x<dwWidth; x++)
        {
            DWORD   dwPixel = SampleEdgePixel32_CPU(pSrc,dwWidth,dwHeight
            DWORD*  pDestColor = (DWORD*)pDest + x + (y*dwWidth);
            *pDestColor = dwPixel;
        }
    }

}
```

```
__global__ void CUDA_Edge_Filter(char* pDest,char* pSrc,DWORD dwWidth,DWO
{
    int px = blockIdx.x * blockDim.x + threadIdx.x;
    int py = blockIdx.y * blockDim.y + threadIdx.y;

    if (px >= (int)dwWidth)
        return;

    if (py >= (int)dwHeight)
        return;


    DWORD   dwPixel = SampleEdgePixel32(pSrc,dwWidth,dwHeight,dwWidth*4,p
    DWORD*  pDestColor = (DWORD*)pDest + px + (py*dwWidth);
    *pDestColor = dwPixel;

}
#endif



__global__ void CUDA_BW_Filter(char* pDest,char* pSrc,DWORD dwWidth,DWORD
{

    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    if (x >= dwWidth)
```

구현전략

- 싱글스레드 C코드로 작성한다
- 자료구조상으로 최적화한다.
- 멀티스레드로 바꾼다.
- CUDA로 (단순) 포팅한다.
- 퍼포먼스를 측정한다.
- CUDA에 적합하도록 일부의 설계를 변경 한다.
- 최적화한다.

CPU코드로 작성

- CUDA로 포팅할 것을 염두해 두고 작성한다
- CUDA 6 이전에는 메모리 할당 및 카피가 주요 이슈였지만 CUDA 6 / Kepler 아키텍처 이후부터 Unified Memory System 사용가능. 명시적으로 시스템 메모리 <-> GPU메모리 전송이 필요없음.
- 일단 CPU상에서 멀쩡히 돌아가는게 가장 중요하다.
- SSE,AVX등 SIMD최적화는 나중에 생각하자.CUDA포팅에 걸림돌이 된다.

Multi-Thread코드 작성

- 잘 돌아가는 Single-Thread코드를 작성했으면 Multi-Thread버전을 만든다.
- 충돌처리할 오브젝트들을 Thread개수로 나눠서 처리.

CUDA코드로 포팅

- 표준적인 문법만 사용했다면 별로 수정할게 없다.
- Multi-Thread코드와 유사하다. CPU Thread -> CUDA Thread로 바로 맵핑하면 된다.
- __device__ 지시어만 사용하지말고 최대한 __host__ __device__ 지시어를 함께 지정해서 CPU에서 미리 돌려볼수 있도록 한다.

단순포팅

- 처리하고 하는 [원소 1개 > thread 1개] 직접 맵핑
- 1 pixel -> 1 thread, 1 오브젝트-> 1스레드
- C로 짠 코드는 거의 100% 그대로 돌릴 수 있다.
- 이미지 프로세싱 매트릭스 연산 등에선 충분히 효과가 있다.

CUDA 프로그래밍

CPU프로그래밍과 다른 점 - GPU는 I/O 디바이스

시스템 메모리 <-> GPU 메모리

- GPU에 일을 시키려면 GPU에서 사용할 데이터는 GPU메모리에 전송해야한다.
- cudaMallocHost()로 시스템 메모리 할당
- cudaMalloc()으로 GPU메모리 할당
- cudaMemcpy()로 전송

시스템 메모리 <-> GPU 메모리

- GPU에 일을 시키려면 GPU에서 사용할 데이터는 GPU메모리에 전송해야한다.
- cudaMallocHost()로 시스템 메모리 할당
- cudaMalloc()으로 GPU메모리 할당
- cudaMemcpy()로 전송

Unified Memory System

- cudaMallocManaged()로 할당하면 CPU<->GPU간 명시적 전송 없이 자동처리
- GPU에서의 Page fault처리의 마법
- 그러나 테스트 결과 명시적으로 host측, device측 메모리 할당 및 전송이 빠르다.
- 또한 일부 GPU와 드라이버에서 Unified Memory System이 문제를 일으킬 수 있다.

CUDA 프로그래밍

최적화

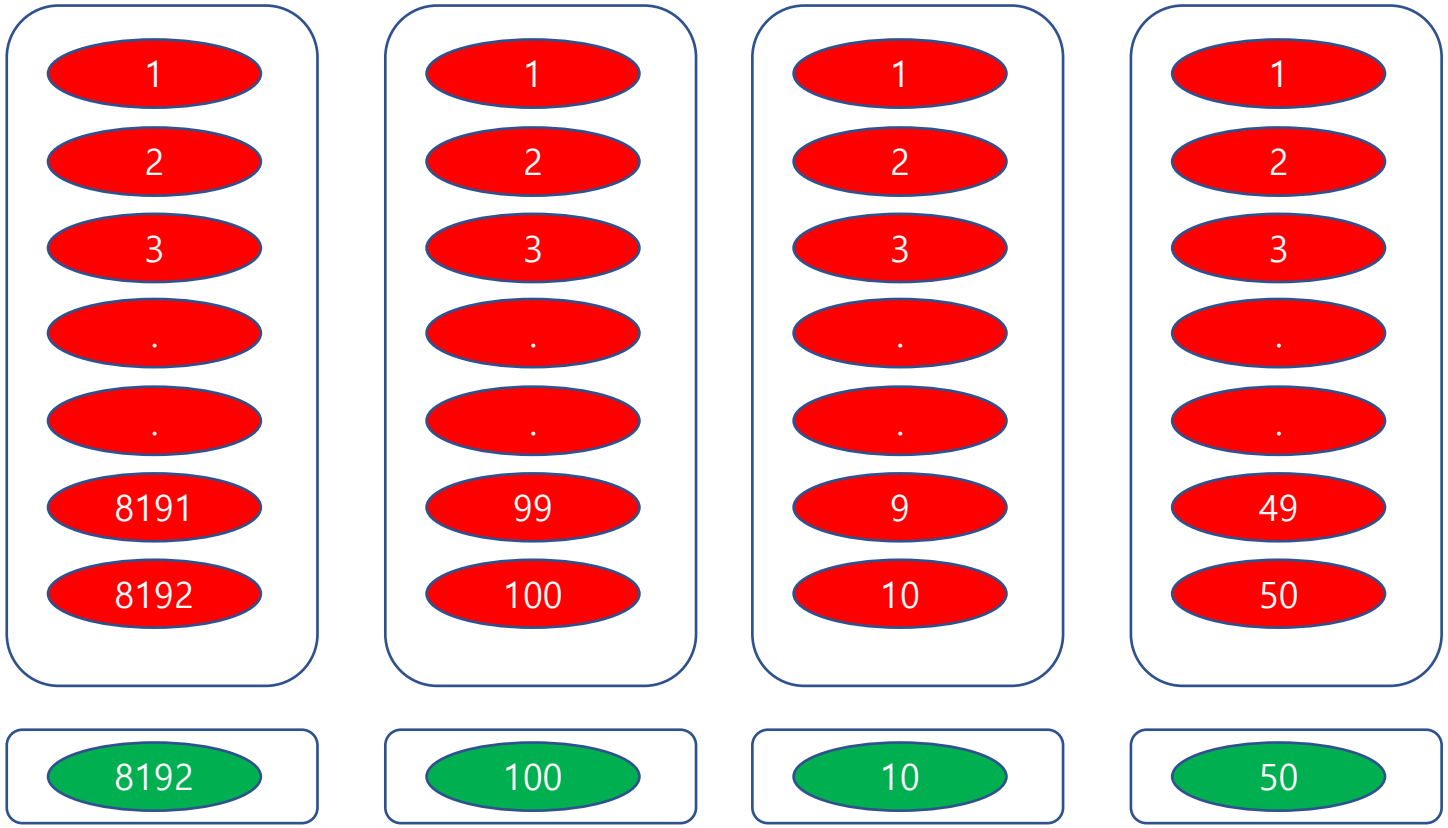
Block와 Thread를 어떻게 사용하는게 효과적일까?

- Per thread처리
- Per block처리
- 몇 개의 block? 몇 개의 thread?

Per block vs per thread예제

- 8192개의 CELL이 있다.
- 각 CELL은 랜덤하게 1 ~ 8192범위의 숫자를 1 ~ 8192 개 가지고 있다.
- 각 CELL에 대해서 최대값을 구한다.
  1) CPU Single Thread
  2) CUDA , 1 Cell -> 1 Thread
  3) CUDA , 1 Cell -> 1 Block

최대값 구하기

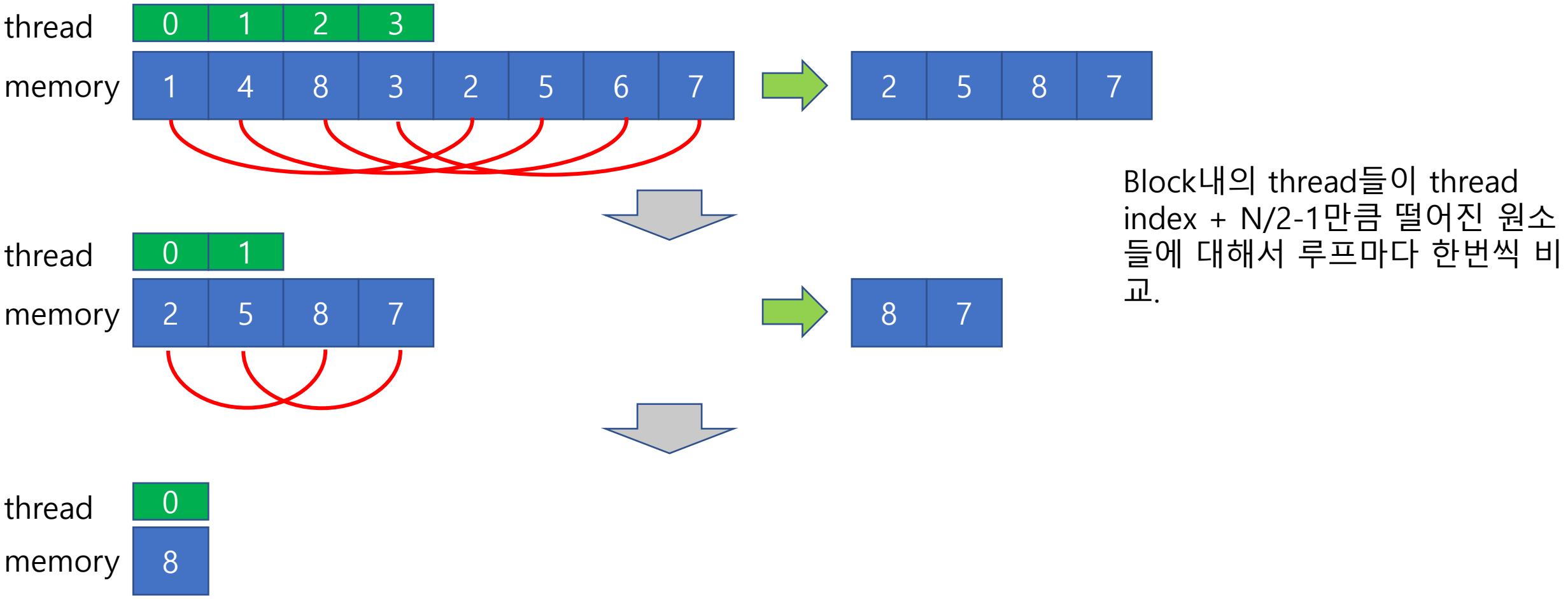


1 Cell -> 1 Thread

```
__global__ void SearchMaxValue_A(CELL* pCellList,DWORD dwCellNum)
{
    // 블럭당 스레드 총 개수
    DWORD ThreadNumPerBlock = blockDim.x * blockDim.y;

    // 그리드당 블럭 총 개수
    DWORD BlockNumpPerGrid = gridDim.x * gridDim.y;

    // 블럭 인덱스
    DWORD UniqueBlockIndex = blockIdx.y * gridDim.x + blockIdx.x;

    // 블럭 내에서의 스레드 인덱스
    DWORD ThreadIndexInBlock = threadIdx.y * blockDim.x + threadIdx.x;

    // 그리드에서의 스레드 인덱스
    DWORD UniqueThreadIndex = UniqueBlockIndex * ThreadNumPerBlock + ThreadIndexInBlock;


    if (UniqueThreadIndex >= dwCellNum)
    {
        return;
    }

    CELL*   pCell = pCellList+UniqueThreadIndex;

    float   MaxValue = -999999.0;

    for (DWORD i=0; i<pCell->CountOfValue; i++)
    {
        if (MaxValue < pCell->Value[i])
        {
            MaxValue = pCell->Value[i];
        }
    }
    pCell->MaxValue = MaxValue;
}
```

일반적인 비교 및 치환

조금 더 똑똑하게 최대값 구하기



Block내의 thread들이 thread index + N/2-1만큼 떨어진 원소들에 대해서 루프마다 한번씩 비교.

1 Cell -> 1 Block

```
__global__ void SearchMaxValue_B(CELL* pCellList,DWORD dwCellNum)
{
    // 스레드별로 중간 결과값을 담을 shared memory
    __shared__  float   g_SharedMaxValue[THREAD_NUM_PER_BLOCK];

    // 블럭당 스레드 총 개수
    DWORD ThreadNumPerBlock = blockDim.x * blockDim.y;

    // 블럭 인덱스
    DWORD UniqueBlockIndex = blockIdx.y * gridDim.x + blockIdx.x;

    // 블럭 내에서의 스레드 인덱스
    DWORD ThreadIndexInBlock = threadIdx.y * blockDim.x + threadIdx.x;

    if (ThreadIndexInBlock >= dwCellNum)
    {
        return;
    }

    // block당 하나의 cell을 처리한다.
    CELL*   pCell = pCellList+UniqueBlockIndex;

    // 각 스레드가 자신이 억세스할 수 있는 인덱스들 중에서 최대값을 구한다.
    float   MaxValue = -999999.0;
    g_SharedMaxValue[ThreadIndexInBlock] = 999999.0;

    DWORD   CurIndex = ThreadIndexInBlock;
    while (CurIndex < pCell->CountOfValue)
    {
        if (MaxValue < pCell->Value[CurIndex])
        {
            MaxValue = pCell->Value[CurIndex];
        }

        CurIndex += ThreadNumPerBlock;
    }
    // 각 스레드들이 억세스 범위 내에서는 최대값을 찾았고, 그 중간값을 shared memory에 저장해둔다.
    g_SharedMaxValue[ThreadIndexInBlock] = MaxValue;

    __syncthreads();    // 여기까지 스레드 동기화
```

Shared Memory사용.
Block내의 thread들이 병렬로 데이터를 shared memory로 로드한다.

1 Cell -> 1 Block

```
    // shared memory안에서 최대값을 찾는다.
    // 한번의 루프를 돌때마다 절반씩 줄여나간다.
    DWORD   dwHalfThreadCount = THREAD_NUM_PER_BLOCK>>1;
    while (dwHalfThreadCount)
    {
        if (ThreadIndexInBlock < dwHalfThreadCount)
        {
            DWORD   dwTargetIndex = ThreadIndexInBlock+dwHalfThreadCount;

            if (g_SharedMaxValue[ThreadIndexInBlock] < g_SharedMaxValue[dwTargetIndex])
            {
                g_SharedMaxValue[ThreadIndexInBlock] = g_SharedMaxValue[dwTargetIndex];
            }
        }
        __syncthreads();

        dwHalfThreadCount = dwHalfThreadCount>>1;
    }

    // 루프를 마치고 나면 Shared Memory의 배열 0번에 가장 큰 값이 저장된다.
    if (0 == ThreadIndexInBlock)
    {
        pCell->MaxValue = g_SharedMaxValue[0];
    }
}
```

Block내의 thread들이 thread index + N/2-1만큼 떨어진 원소들에 대해서 루프마다 한번씩 비교.

Occupancy

- SM당 활성화 Warp 수 / SM당 최대 Warp 수
- GPU의 처리능력을 얼마나 사용하는지 척도
- 대부분의 경우 Occupancy가 증가하면 효율(시간당 처리량)도 증가함
- Global Memory 억세스로 latency가 발생할때 유휴 Warp를 수행함.
- GPU 디바이스의 스펙에 상관없이 WARP를 많이 유지할 수 있으면 처리성능이 높아진다.

Occupancy Calculator

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click):	5.2	(Help)
1.b) Select Shared Memory Size Config (bytes)	98304	
1.c) Select Global Load Caching Mode	L2 only (cg)	

2.) Enter your resource usage:		
Threads Per Block	512	(Help)
Registers Per Thread	32	
Shared Memory Per Block (bytes)	16384	

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:		
Active Threads per Multiprocessor	2048	(Help)
Active Warps per Multiprocessor	64	
Active Thread Blocks per Multiprocessor	4	
Occupancy of each Multiprocessor	100%	

Physical Limits for GPU Compute Capability:	5.2
Threads per Warp	32
Max Warps per Multiprocessor	64
Max Thread Blocks per Multiprocessor	32
Max Threads per Multiprocessor	2048
Maximum Thread Block Size	1024
Registers per Multiprocessor	65536
Max Registers per Thread Block	65536
Max Registers per Thread	255
Shared Memory per Multiprocessor (bytes)	98304
Max Shared Memory per Block	49152
Register allocation unit size	256
Register allocation granularity	warp
Shared Memory allocation unit size	256
Warp allocation granularity	4

Allocated Resources	Per Block	Limit Per SM	= Allocatable Blocks Per SM
Warps (Threads Per Block / Threads Per Warp)	16	64	4
Registers (Warp limit per SM due to per-warp reg count)	16	64	4
Shared Memory (Bytes)	16384	49152	6

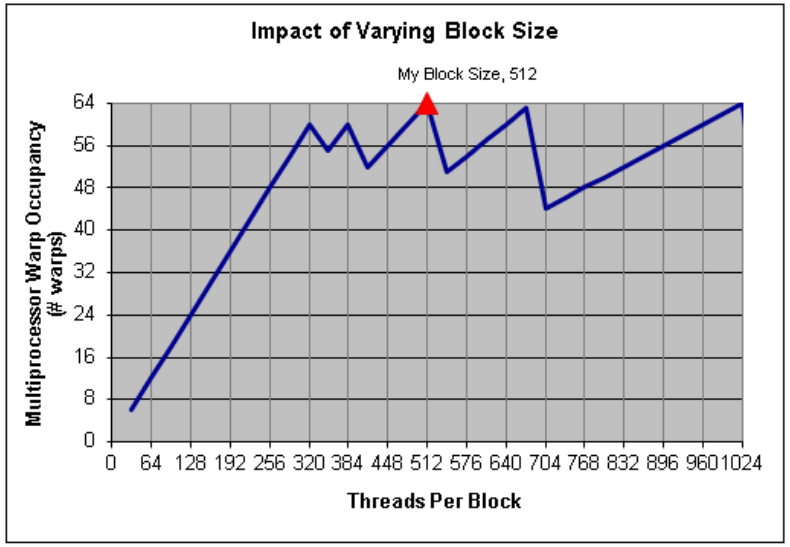
Note: SM is an abbreviation for (Streaming) Multiprocessor

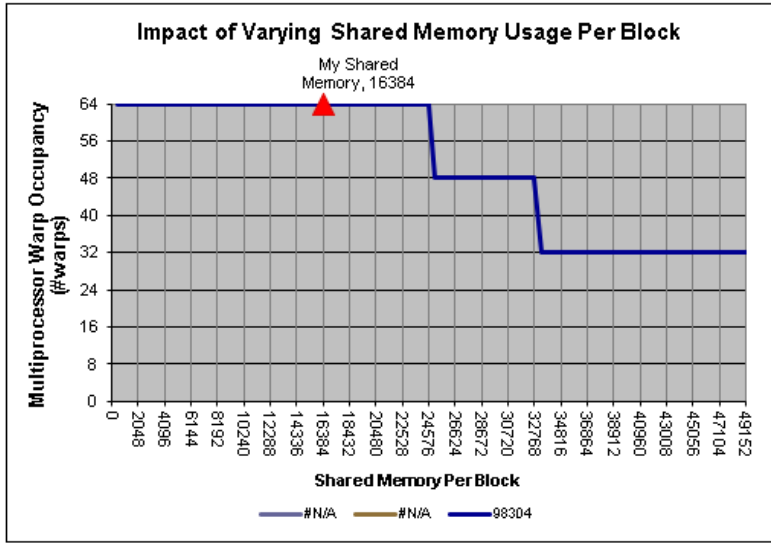
Maximum Thread Blocks Per Multiprocessor	Blocks/SM	* Warps/Block	= Warps/SM
Limited by Max Warps or Max Blocks per Multiprocessor	4	16	64
Limited by Registers per Multiprocessor	4	16	64
Limited by Shared Memory per Multiprocessor	6		

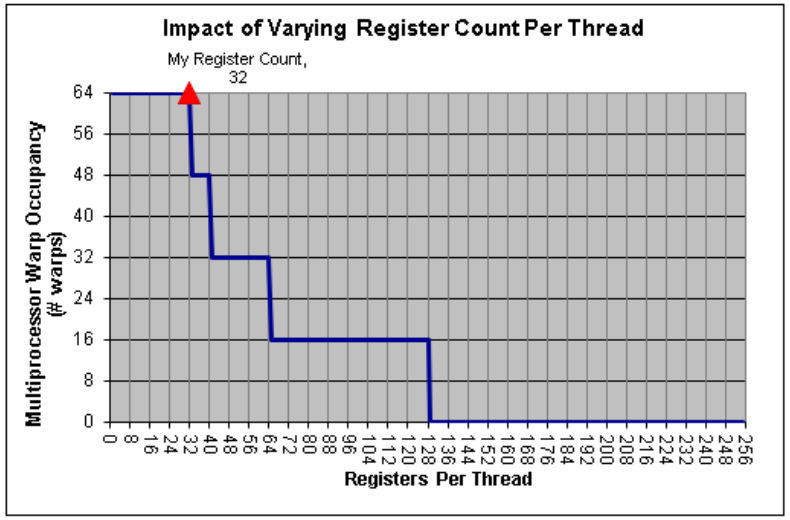
Note: Occupancy limiter is shown in orange

Physical Max Warps/SM = 64
Occupancy = 64 / 64 = 100%

Your chosen resource usage is indicated by the red triangle on the graphs. The other data points represent the range of possible block sizes, register counts, and shared memory allocation.







Occupancy늘리기

- 많은 워프를 사용하여 Latency 숨기기
- 레지스터 수를 줄이도록 노력한다.
- Shared Memory사용을 아낀다.(줄이는 것이 아니다)
- Shared Memory사용량 때문에 Occupancy를 떨어뜨린다면 Shared Memory대신 Global Memory를 사용하는 것이 나을수 있다.(L2캐시 때문에 생각보단 느리지 않다)

Register 사용량 줄이기

- 사실 뾰족한 방법이 있는건 아니다.
- 알고리즘 수준에서의 최적화 -> 결과적으로 작은 코드.
- 레지스터 개수를 컴파일 타임에 제한할 수 있다.

Shared Memory아껴쓰기

- 자주 억세스할 경우 Shared Memory에 올려놓으면 빠르다.
- 무작정 마구 쓰면 SM에 맵핑할 수 있는 블럭 수가 크게 떨어진다.
- Shared Memory절약 ->Active Block수 증가 -> 성능 향상

cudaStream을 이용한 비동기 처리

- CUDA디바이스는 기본적으로 Copy engine을 2개 가지고 있다.
- 시스템 메모리 <-> GPU메모리 전송 도중에도 CUDA Kernel코드 실행 가능
- 따라서 여러 개의 copy 작업과 연산 작업을 동시에 처리 가능
- 각 작업의 컨텍스트를 유지하고 순서를 맞추기 위해 cudaStream을 사용.
- 여러 개의 cudaStream과 Multi-Thread와 함께 사용하여 성능을 극대화한다.

한번에 많은 데이터 처리하기.

- 내부적으론 PCI버스를 타고 GPU메모리로 전송됨.
- CUDA Kernel함수 한번 호출하고 결과를 받아오는 작업은 굉장히 긴 시간이 필요함. – 비행기 탑승에 걸리는 시간은?
- 작은 데이터를 여러 번 처리하는 대신 모아서 한번에 처리한다.

데이터 사이즈 줄이기

- CUDA Kernel함수 실행중에 가장 많은 시간을 소모하는 작업은 global memory 읽기/쓰기
- 데이터 사이즈가 작으면 global memory억세스가 줄어듦 -> 성능향상
- 데이터 사이즈가 작으면 캐시 히트율이 높아짐. -> global memory 억세스가 줄어듦 -> 성능 향상.

게임 프로젝트에 CUDA적용

GPGPU 적용 판단 기준

부적합한 경우

- 분기가 많다.
- 각 요소들의 의존성이 높다.(병렬화 하기 나쁘다)
- Throughput보다 Responsibility가 중요하다.(10ms에 목숨을 건다!)

적합한 경우

- 코드에서 분기가 적다.
- Image Processing처럼 각 요소들의 의존성이 낮다. – (병렬화 하기 좋다.)
- Throughput이 중요한 경우.(3일걸리던 작업을 1일로 줄였다!!!)

게임프로젝트에 CUDA적용

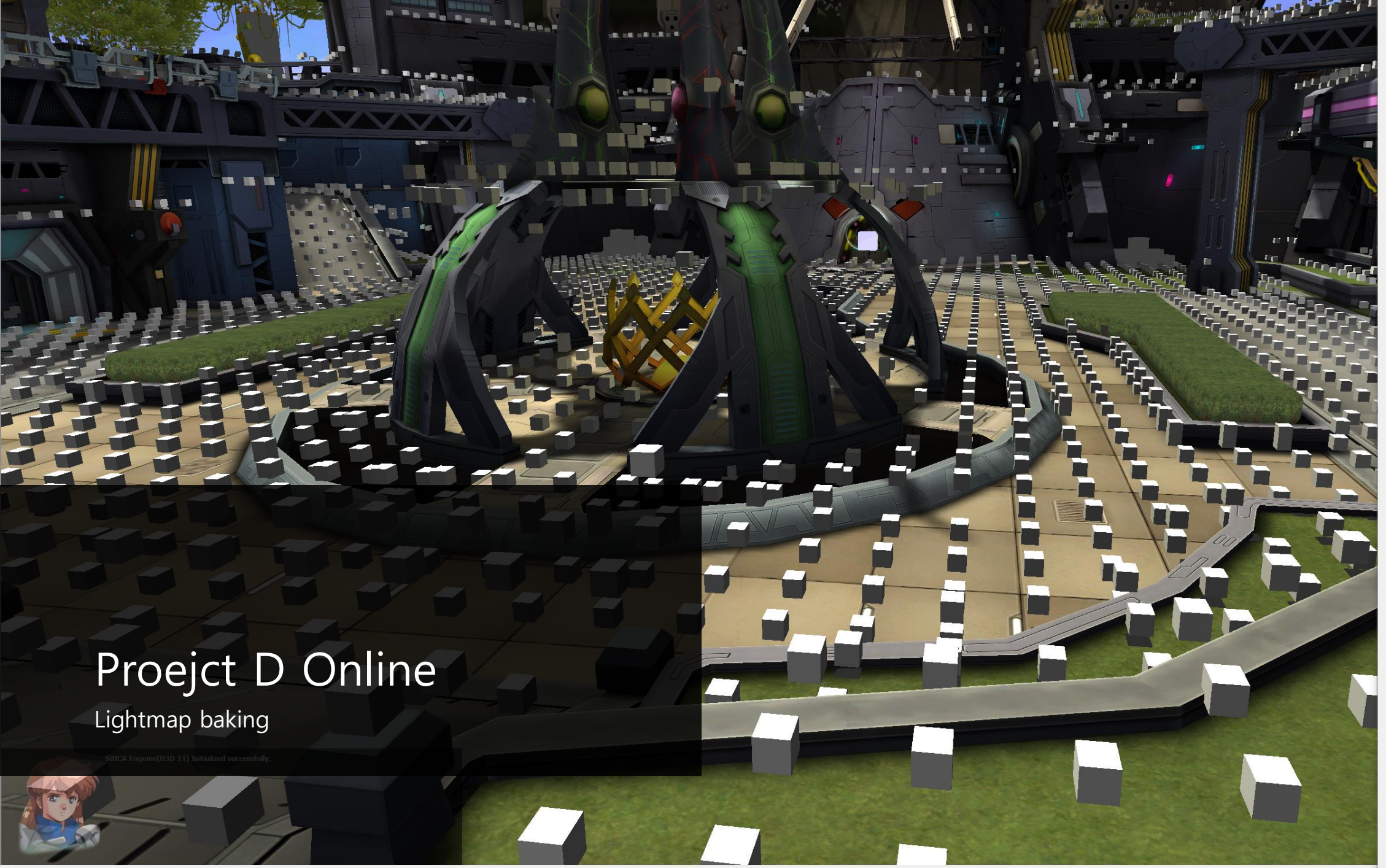
- nvidia GPU만 지원하잖아
  - 어차피 게이머들 GPU는 75%이상 nvidia제품. 절반 이상의 플레이어에게 도움이 됨.
  - 에디터등 개발중에만 사용해도 충분히 쓸모가 많음.
  - 코드 논리와 흐름은 같으므로 Compute Shader로 포팅할때 도움이 됨.
  - Multi-thread 최적화에도 도움이 됨.

일반적으로 가능한 작업들

- 물리처리등– PhysX (실제로 널리 사용)
- 개발 단계에서 정적 라이트 계산
- 게임 플레이중 라이트 계산
- 압축된 텍스처나 지형의 디코딩

Rage - id soft

- https://www.geforce.com/whats-new/articles/how-to-unlock-rages-high-resolution-textures-with-a-few-simple-tweaks



Align Y | Align X | Lock

Polygon Room

- WALL — CUT
- POLYGON
- Draw Portal — Pick Po
- Inverse All | Inverse Roo
- Auto Interior | Auto Extric

Portal Link

- Select Portal
- Portal
- Room A
- Room B

Build

- Build BSP | Insert M

Test

- Portal Culling
- Real Camera
- Virtual Camera

Display

- Only Selected Room
- WIREFRAME
- Hidden Polygon
- Inverse

Load

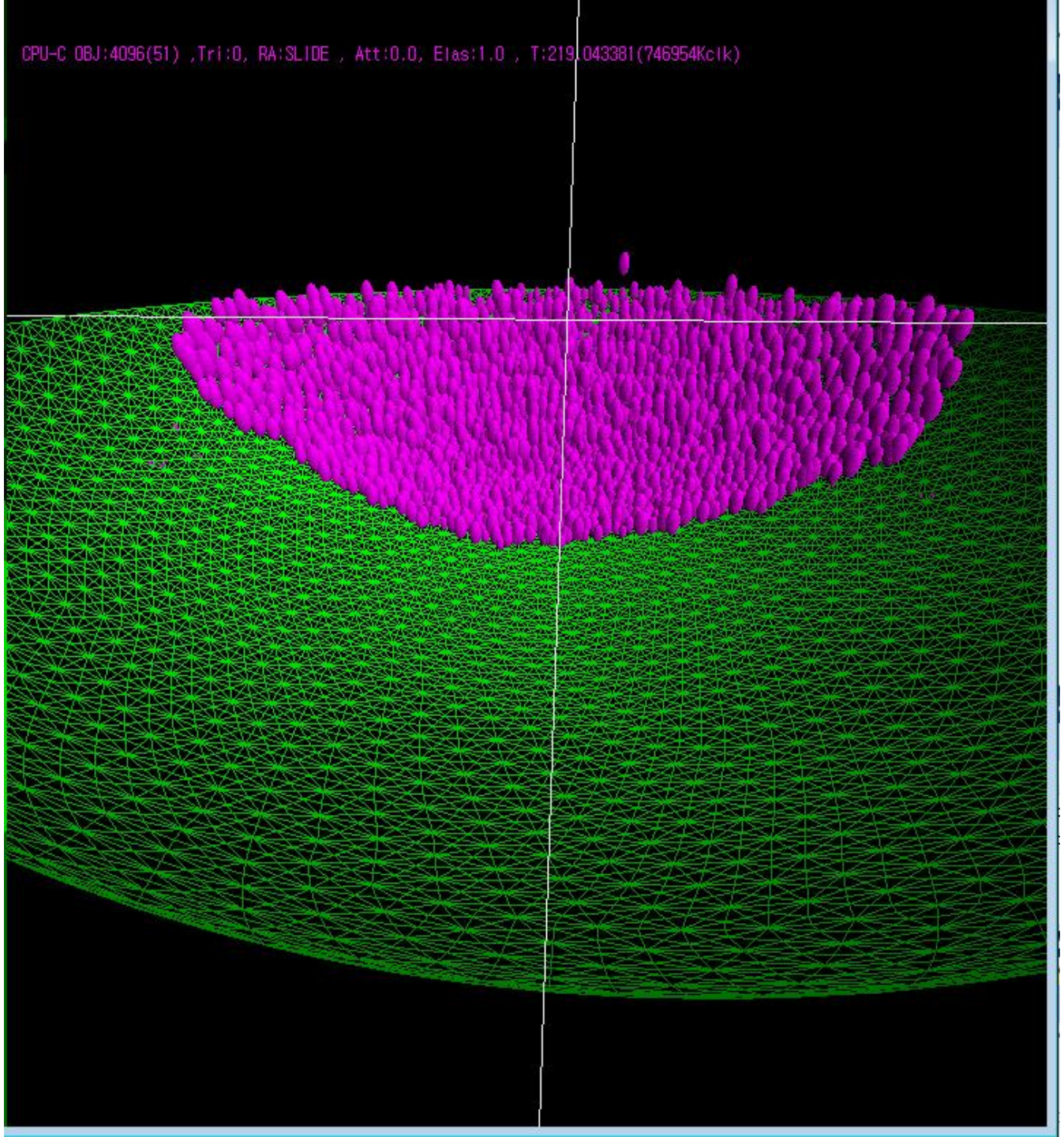
준비

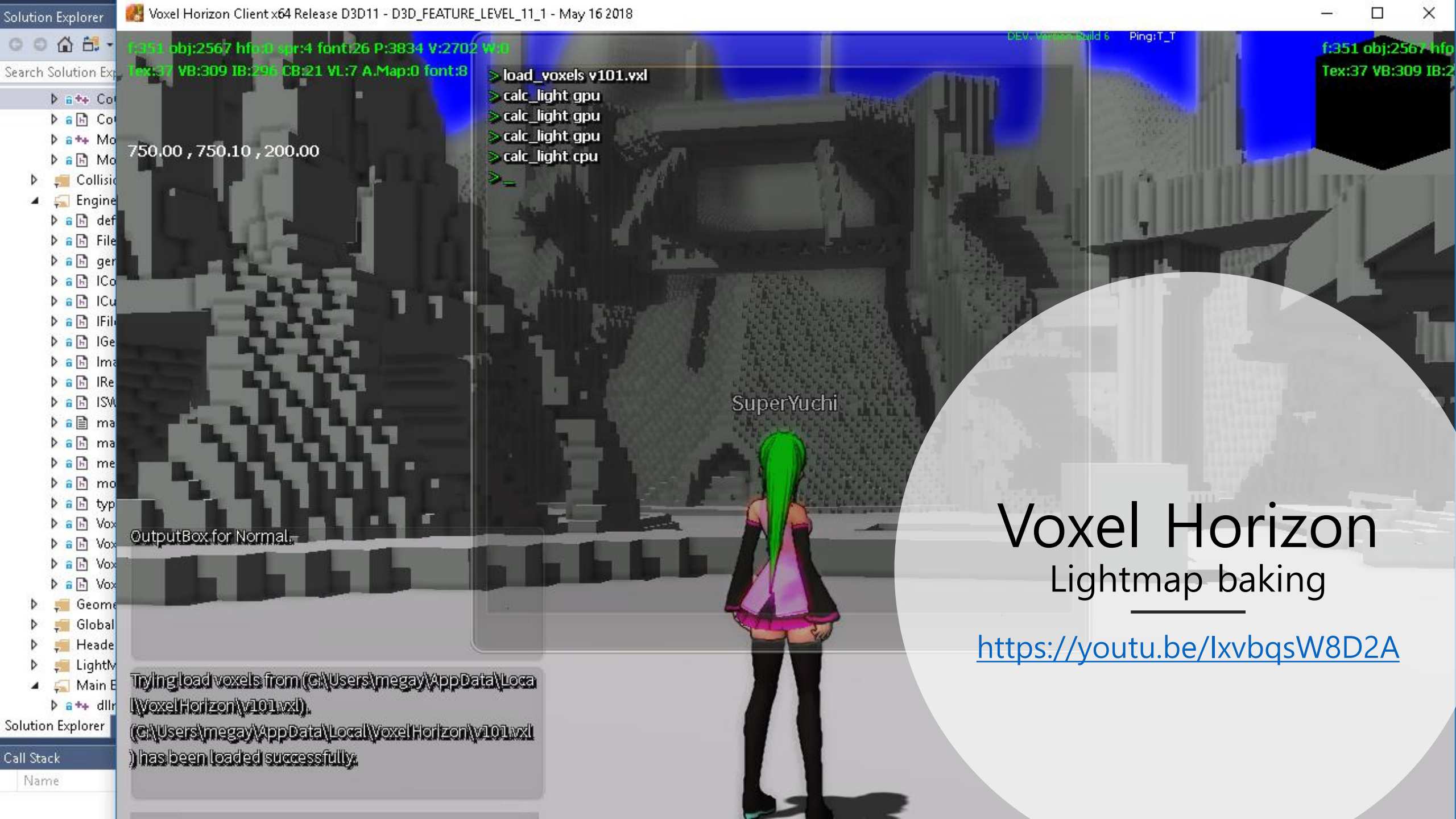
서버/클라이언트를 위한 충돌처리

- https://youtu.be/QHv3tVF0qxc
- https://youtu.be/cY2sa_dGbMc

- http://youtu.be/towb280Qlgo
- http://youtu.be/egWQUxvTThs

- 성능은 2배-4배 정도
- 몇가지 이유로 상용서비스로의 적용은 하지 않았음.




Solution Explorer
Search Solution Exp
Call Stack
Name
Voxel Horizon Client x64 Release D3D11 - D3D_FEATURE_LEVEL_11_1 - May 16 2018
f:351 obj:2567 hfo:0 spr:4 font:26 P:3834 V:2702 W:0
Tex:37 VB:309 IB:296 CB:21 VL:7 A.Map:0 font:8
DEV. Version Build 6
Ping:T_T
f:351 obj:2567 hfo
Tex:37 VB:309 IB:2
> load_voxels v101.vxl
> calc_light gpu
> calc_light gpu
> calc_light gpu
> calc_light cpu
>_
750.00 , 750.10 , 200.00
SuperYuchi
OutputBox for Normal.
Trying load voxels from (C:\Users\megay\AppData\Local\VoxelHorizon\v101.vxl).
(C:\Users\megay\AppData\Local\VoxelHorizon\v101.vxl
) has been loaded successfully.
Voxel Horizon
Lightmap baking
https://youtu.be/IxvbqsW8D2A

Reference

- https://megayuchi.com
- https://developer.nvidia.com/cuda-zone