NVIDIA Jetson TK1 과 CUDA™ 프로그래밍

[필자소개]

서영진 | http://valentis.pe.kr, valentis@chollian.net

IT 분야에서 90년대부터 십여년이 넘게 프로그래밍을 하고 있으며, IT 전문강사와 컨설턴트, 관광 TC로도 일하고 있다. 리눅스용 다이얼패드, SKY 6400/6500 모바일 캠코더, 원자력 발전소 CPS 시스템, 신도리고 NEST UI, 삼성전자 VOIP 전화기 등 리눅스/UI(Qt)/임베디드/모바일/스마트폰 쪽에서 프로그램을 개발했으며, 이집트 SECC, 삼성전자, LG전자 등의 정부 기관/업체와 대구/DIP/인하/ 원광/전북/조선대학교 앱창작터와 3DFIA, 전자부품연구원(KETI), KEA, RAPA, KOSTA 등의 협회에서 강의 및 세미나를 진행하였다. 주요 저술로는 “타이젠으로 웨어러블 앱 개발하기”, "사물인터넷 - 우리가 꿈꾸는 스마트한 세상", "Tizen 애플리케이션 프로그래밍", "[열혈강의] Qt 프로그래밍" 등이 있다.

1990 년대만 하더라도 주머니에 PC 를 한 대씩 넣고 다니는 것은 꿈에 불과했지만 현재 대부분의 개인이 그 시절보다 더 빠르고 좋은 스마트 디바이스들을 사용하고 있다. 스마트 디바이스들은 점점 발달하고 있고 속도도 빨라지고 있다. 하지만 현재의 반도체 기술은 재질과 노이즈 등의 문제로 더 이상 클럭 주파수를 놀려서 속도를 증대시키는데 한계점을 보이고 있다. 이러한 속도 문제의 해결을 위해서 등장한 것이 멀티 프로세서, 멀티 코어 시스템이다.
1990 년대 들면서 3D 게임과 MS 윈도우와 같은 GUI 시스템의 등장 그리고 웹에서의 멀티미디어 지원 등으로 컴퓨터 그래픽이 발달하면서 3D 그래픽 카드가 점점 발달하였다. 범용 연산을 위해서 사용되는 CPU 와 다르게 GPU 는 단순한 행렬 연산에 최적화되어 있다. 이러한 GPU 를 이용해서 프로그래밍하기 위해 등장한 개념이 GPGPU(General-Purpose computing on Graphics Processing Units) 프로그래밍이다.
CPU 에 GPU 나 DSP 등을 이용해서 병렬 연산을 수행하는 속도를 높일 수 있는데, 산업계의 개발형 표준인 OpenCL(Open Computing Language)과 함께 NVIDIA 의 CUDA(Compute Unified Device Architecture)가 많이 사용되고 있다.
지난 달에는 그래픽 카드의 발달과 3D 의 등장에 대해서 살펴보고 GPGPU 프로그래밍이 등장한 배경에 대해서 알아보았다. NVIDIA 의 JetsonTK1 에 대해서 살펴보았고 개발 환경을 구성하는 방법에 대해 배웠다. 이번 달에는 지난 달에 이어 간단한 “Hello CUDA!” 프로그램을 작성해보고 CUDA 에 대해 보다 깊게 살펴보도록 하겠다.

8 월호 : 사물인터넷을 위한 모바일 슈퍼컴퓨터 : NVIDIA Jetson TK1
9 월호 : NVIDIA Jetson TK1 과 CUDA™ 프로그래밍
10 월호 : **CUDA™ 병렬 프로그래밍**과 OpenCV 프로그래밍

[목차]

1. Hello CUDA 프로그래밍

Jetson TK1을 사용할 준비가 끝나면 간단한 코드를 만들어 애플리케이션을 개발하는 방법을 살펴보도록 하자. 지난 달에도 설명했지만 CUDA는 NVIDIA에서 GPGPU 프로그래밍을 위해서 만들었다.

3-1. Hello World 프로그래밍

Jetson TK1은 ARM 기반으로 되어 있으며, 전형적인 리눅스의 개발 도구인 GCC(the GNU Compiler Collection)를 지원하고 있고 CUDA 개발을 위한 개발 도구들도 역시 제공한다. 일반적인 개발은 데스크탑 리눅스와 같이 C 언어를 사용할 수 있다.

CUDA는 표준 C 언어를 기반으로 병령처리를 위한 함수(명령어)들을 제공하고 있다. 하나의 C 언어에서 CPU에서 실행되는 C 언어의 함수와 GPU에서 실행하기 위한 CUDA 함수들을 함께 사용한다.

CUDA의 개발 환경으로 vi/Emacs 외에도 Eclipse, KDevelop, Qt Creator, Gedit, Geany, NetBeans, Code::Blocks 등을 사용할 수 있다. 원하는 텍스트 에디터를 이용해서 다음의 코드를 입력해보자. <코드 1>과 같이 C 언어만을 사용한 함수는 CPU에서 실행된다.

<코드 1> helloworld.c

```
#include <stdio.h>

int main(int argc, char** argv)
{
    printf("Hello World!\n");

    return 0;
}
```

앞의 코드는 C 언어를 이용해서 간단히 터미널로 “Hello World!”를 출력하였다. 소스 코드의 빌드는 터미널에서 gcc 명령어를 이용하면 된다.

```
ubuntu@tegra-ubuntu:~$ gcc -o helloworld helloworld.c
```

유닉스에서 gcc를 이용한 소스 코드의 빌드 시 아무런 옵션이 없으면 a.out이라는 이름의 실행 파일을 생성하는데, 이때 gcc에 –o 옵션을 이용해 실행 파일의 이름을 부여하면 된다. 소스 코드의 빌드가 완료되면 유닉스에서 현재 디렉터리 내에 있는 실행 파일은 './'을 붙여서 실행하면 된다.

```
ubuntu@tegra-ubuntu:~$ ./helloworld
Hello World!
```

1-2. Hello CUDA 프로그래밍

이제 앞의 Hello World 프로그램을 수정해서 간단한 CUDA 프로그램을 작성해보자. 앞의 코드를 다음과 같이 수정한다. CUDA의 소스코드는 .cu의 확장자를 가지고 있다.

<코드 2> hellocuda.cu

```
#include <stdio.h>

__global__ void kernel()
{
}

int main(int argc, char** argv)
{
    /* 디바이스에서 싱글 스레드를 갖는 커널을 실행(Launch) */
    kernel<<<1,1>>>();

    /* Hello CUDA! 출력 */
    printf("Hello CUDA!\n");
    printf("CUDA error: %s\n", cudaGetErrorString(cudaGetLastError()));

    return 0;
}
```

앞의 코드는 CUDA를 이용한 코드로 앞의 C 언어와 같이 printf( ) 함수를 이용해서 간단히 터미널로 “Hello CUDA!”를 출력하였다.

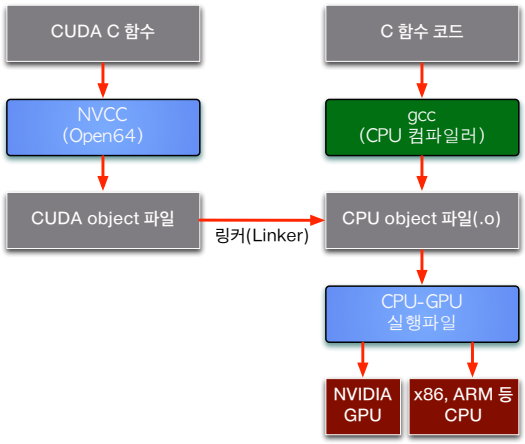
먼저 코드를 빌드하기 위해서는 gcc가 아닌 nvcc(Nvidia CUDA Compiler)[1]를 사용하는데, 기본 옵션은 gcc와 비슷하다. 앞의 코드를 빌드해서 실행해보면 다음과 같은 결과를 볼 수 있다.

```
ubuntu@tegra-ubuntu:~$ nvcc -o hellocuda hellocuda.cu
ubuntu@tegra-ubuntu:~$ ./hellocuda
```

[1] http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc 참조

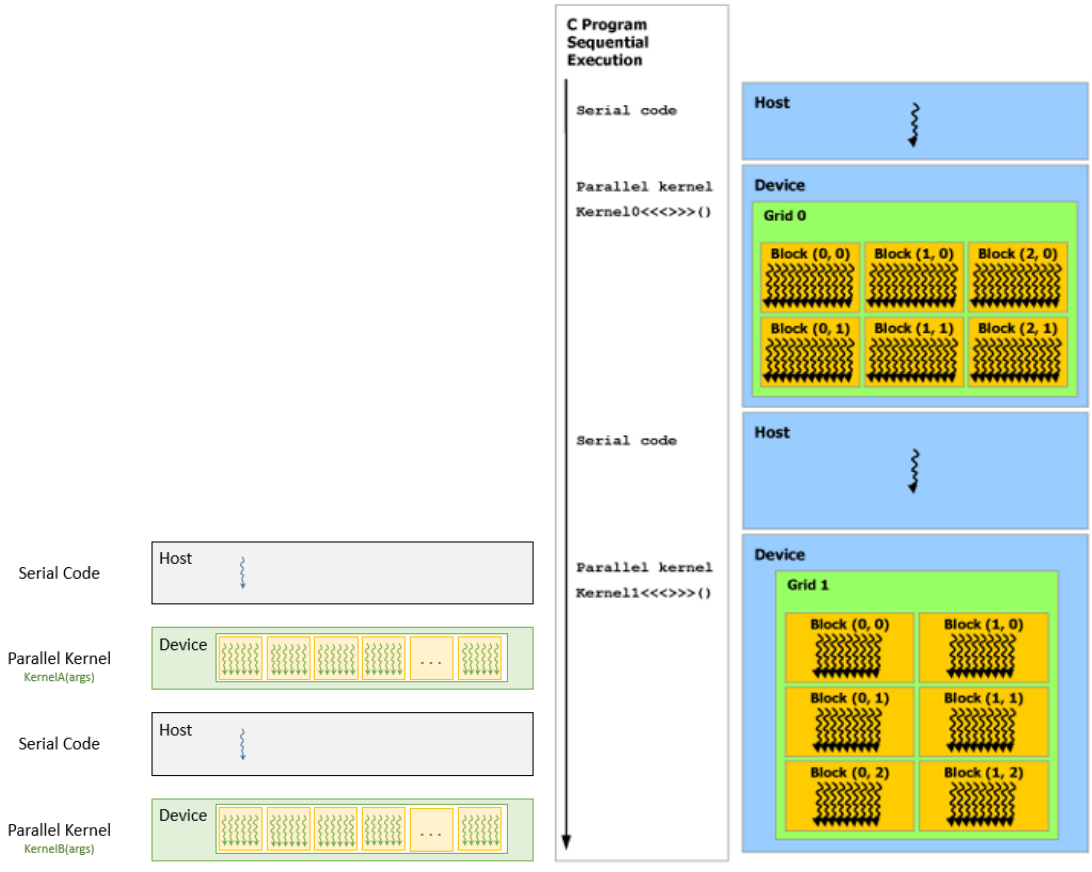
```
Hello CUDA!
CUDA error: no error
```

기본적인 코드는 C 언어와 비슷하지만 CUDA의 디바이스를 사용할 수 있도록 함수를 추가하였다. nvcc는 소스코드를 CPU에서 실행될 코드와 GPU에서 실행될 코드로 분리해서 각자의 컴파일러가 컴파일할 수 있도록 해준다.



<그림 1> CUDA의 컴파일 환경

이렇게 빌드된 실행 파일은 CPU와 GPU에서 실행되게 되는데, CPU를 호스트(Host)라고 하고 GPU를 디바이스(Device)라고 하는데, 호스트에서 호출되고 디바이스에서 실행되는 함수를 커널(kenrel) 함수라고 부른다. CPU에서는 호스트 스레드를 직렬로 처리하고 GPU에서는 병렬로 처리한다.



<그림 2> GPU와 CPU에서의 스레드 처리

앞의 코드에 대해서는 뒤의 "CUDA의 기본 문법"에서 보다 자세히 살펴보도록 하겠다.

1-3. GPU 사양 가져오기[2]

현재 CUDA는 NVIDIA의 그래픽 카드나 Jetson TK1과 같은 NVIDIA의 GPU를 사용하고 있는 하드웨어에서밖에 구동이 되지 않는다. NVIDIA의 그래픽 카드는 지포스(GeForce) 시리즈, 고성능의 그래픽 프로세스의 쿼드로(Quadro) 시리즈, 그리고 GPGPU용의 고성능 그래픽 프로세스인 테슬라(Tesla) 시리즈로 구분할 수 있는데 각 그래픽스 카드 군에 따라서 지원되는 기능들이 다르다.

현재 사용하고 있는 시스템의 GPU에 대한 정보를 알아와야하는 경우가 있는데, CUDA에서는 이러한 API를 지원하고 있다. 이제 디바이스의 정보를 가져오는 간단한 코드를 작성해보자.

<코드 3> cudaProperty.cu

```
#include <stdio.h>

int main(int argc, char** argv)
{
    int nDevices;

    cudaGetDeviceCount(&nDevices);
    for (int i = 0; i < nDevices; i++) {
        cudaDeviceProp prop;
        cudaGetDeviceProperties(&prop, i);
        printf("Device Number: %d\n", i);
        printf("  Device name: %s\n", prop.name);
        printf("  Major revision number: %d\n", prop.major);
        printf("  Minor revision number: %d\n", prop.minor);
        printf("  Total global memory: %u\n", prop.totalGlobalMem);
        printf("  Total shared memory per block: %u\n", prop.sharedMemPerBlock);
        printf("  Memory Clock Rate(KHz): %d\n", prop.memoryClockRate);
        printf("  Memory Bus Width(bits): %d\n", prop.memoryBusWidth);
        printf("  Peak Memory Bandwidth(GB/s): %f\n",
                                2.0*prop.memoryClockRate*(prop.memoryBusWidth/8)/1.0e6);
        printf("  Total registers per block:  %d\n", prop.regsPerBlock);
        printf("  Warp size: %d\n", prop.warpSize);
        printf("  Maximum memory pitch: %u\n", prop.memPitch);
        printf("  Maximum threads per block: %d\n", prop.maxThreadsPerBlock);
        printf("  Clock rate: %d\n", prop.clockRate);
        printf("  Total constant memory: %u\n", prop.totalConstMem);
        printf("  Texture alignment: %u\n", prop.textureAlignment);
        printf("  Concurrent copy and execution: %s\n", (prop.deviceOverlap ? "Yes" : "No"));
        printf("  Number of multiprocessors: %d\n", prop.multiProcessorCount);
        printf("  Kernel execution timeout: %s\n",
                                            (prop.kernelExecTimeoutEnabled ?"Yes" : "No"));
        for (int i = 0; i < 3; ++i) {
            printf("    Maximum dimension %d of block: %d\n", i, prop.maxThreadsDim[i]);
            printf("    Maximum dimension %d of grid: %d\n", i, prop.maxGridSize[i]);
        };
    };
}
```

NVidia Jetson TK1에서 앞의 코드를 실행해보면 다음과 같이 디바이스의 이름과 버전 등의 정보를 결과를 볼 수 있다.

```
ubuntu@tegra-ubuntu:~$ nvcc -o cudaProperty cudaProperty.cu
```

[2] http://docs.nvidia.com/cuda/cuda-samples 참조

```
ubuntu@tegra-ubuntu:~$ ./cudaProperty
Device Number: 0
  Device name: GK20A
  Major revision number: 3
  Minor revision number: 2
  Total global memory: 1984397312
  Total shared memory per block: 49152
  Memory Clock Rate(KHz): 924000
  Memory Bus Width(bits): 64
  Peak Memory Bandwidth(GB/s): 14.784000
  Total registers per block:  32768
  Warp size: 32
  Maximum memory pitch: 2147483647
  Maximum threads per block: 1024
  Clock rate: 852000
  Total constant memory: 65536
  Texture alignment: 512
  Concurrent copy and execution: Yes
  Number of multiprocessors: 1
  Kernel execution timeout: No
    Maximum dimension 0 of block: 1024
    Maximum dimension 0 of grid: 2147483647
    Maximum dimension 1 of block: 1024
    Maximum dimension 1 of grid: 65535
    Maximum dimension 2 of block: 64
    Maximum dimension 2 of grid: 65535
```

이제 앞의 코드를 살펴보도록 하자. 현재 장치에서 GPU의 갯수는 cudaGetDeviceCount( ) 함수를 통해서 알 수 있고, CUDA에서 디바이스에 대한 정보는 cudaDeviceProp 구조체[3]와 cudaGetDeviceProperties( ) 함수를 사용한다. cudaDeviceProp 구조체는 장치의 정보와 관련된 다양한 멤버들을 가지고 있다. Constant 메모리와 관련된 내용은 뒤에서 살펴본다.

<표 1> cudaDeviceProp 구조체의 주요 멤버

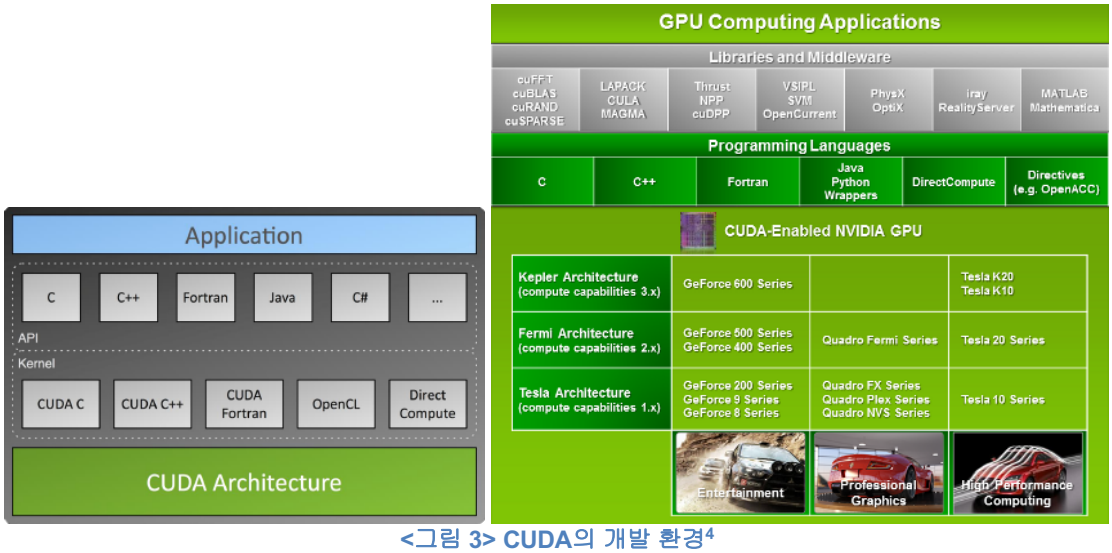
멤버	내용	비고
ECCEnabled	디바이스가ECC를 지원하는지 확인	
asyncEngineCount	비동기(asynchronous) 엔진의 수	
canMapHostMemory	cudaHostAlloc/cudaHostGetDevicePointer와 함께 호스트 메모리를 매핑할 수 있는지 확인	
clockRate	클럭 주파수	kHz
computeMode	Compute 모드(cudaComputeMode 참조)	
concurrentKernels	동시에 여러(Multiple) 커널의 실행이 가능한지 확인	
globalL1CacheSupported	L1 캐쉬 내에서 글로벌 캐싱을 지원하는지 확인	
l2CacheSize	L2 캐쉬의 크기	byte
name[256]	디바이스를 구분할 수 있는 ASCII 코드의 문자열	
major	Major 리비전 번호	
minor	Minor 리비전 번호	
totalGlobalMem	디바이스에서 사용할 수 있는 전역 메모리	byte
sharedMemPerBlock	블록 당 가능한 공유 메모리	byte
memoryClockRate	메모리의 수직(Peak) 클럭 주파수	kHz
memoryBusWidth	전역 메모리 버스 대역	bit
regsPerBlock	블록당 가능한 32비트 레지스터	
warpSize	스레드에서의 워프(Warp) 크기	
memPitch	메모리 복사시 혀용되는 최대 피치(pitch)	byte

[3] http://docs.nvidia.com/cuda/cuda-runtime-api/structcudaDeviceProp.html#structcudaDeviceProp 참조

maxThreadsPerBlock	블록당 최대 스레드의 수	
totalConstMem	디바이스에서 사용가능한 Constant 메모리	byte
textureAlignment	텍스쳐(Texture)를 위해 필요한 정렬	
multiProcessorCount	디바이스에서의 멀티프로세스의 수	
maxThreadsDim[3]	블록의 각 차원(dimension)에서의 최대 크기	
maxGridSize[3]	그리드의 각 차원(dimension)에서의 최대 크기	

2. CUDA™의 기본 문법

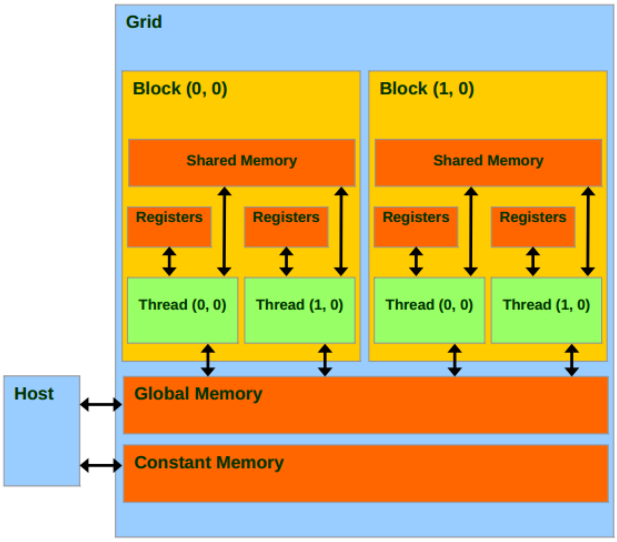
앞에서 본 것과 같이 CUDA는 C 언어를 기반으로 하고 있는데, C++, Perl이나 Java, C#, Python, Visual Basic, .net 등의 언어를 사용해서도 개발이 가능하다.



<그림 3> CUDA의 개발 환경[4]

2-1. CUDA에서의 메모리와 변수 키워드

CUDA에서의 메모리는 전역 메모리(Global Memory), 공유 메모리(Shared Memory), Constant 메모리, 텍스쳐 메모리(Texture Memory) 등으로 구분된다.



<그림 4> CUDA의 메모리 구조[5]

[4] 이미지 출처 : http://docs.nvidia.com/cuda/cuda-c-programming-guide
[5] 이미지 출처 : https://sidkashyap.wordpress.com/tag/cuda

CUDA에서 전역 메모리는 CPU의 메모리와 같은 개념의 GPU에 위치하는 메모리로 속도가 느리기 때문에 랜덤으로 접근하는 경우에 성능이 떨어지게 된다. 공유 메모리는 하나의 블록 안에 있는 스레드들이 동시에 접근해서 사용할 수 있는 메모리로 속도가 빠르지만 스레드가 동시에 접근하는 것에 약간의 제약이 있다. Constant 메모리는 디바이스에서는 쓸 수 없고 읽기만 가능한 메모리다. 커널에서 사용하기 전에 메모리의 값을 설정해주고 사용해야 한다. 전역 메모리에 있지만 보다 빠르게 접근할 수 있다.

CUDA의 기본 데이터형은 C 언어의 데이터형을 사용하는데, 각 자료형의 사용에 CUDA의 메모리를 지정하기 위한 변수 키워드를 제공하고 있다.

<표 2> CUDA의 변수 키워드

키워드	내용	비고
__device__	전역 메모리 영역에 할당되어 애플리케이션의 종료시까지 유효	
__constant__	상수 메모리(constant memory) 영역에 할당되어 애플리케이션의 종료시까지 유효	cudaMemoryToSymbol( ) 함수
__shared__	공유 메모리 영역에 할당되어 현재 실행되고 있는 스레드 내에서 유효	

2-2. CUDA에서의 함수 키워드

앞의 **hellocuda.cu에서 본 것과 같이** CUDA에서는 함수의 선언시 "__global__"을 이용해야 한다. __global__ 키워드 이외에도 다음과 같은 키워드를 제공한다.

<표 3> CUDA의 함수 키워드

키워드	내용	비고
__global__	CPU에 의해서 실행되는 GPU 커널 함수	- 반드시 void로 반환 - 함수 내에 static 변수 사용 금지 - 재귀호출 금지
__device__	GPU 함수로 부터 호출 가능	- 함수 내에 static 변수 사용 금지 - 재귀호출 금지
__host__	CPU 함수로 부터 호출 가능(기본값)	

앞의 표에서 "__host__"와 "__device__"은 함께 사용할 수 있다. 이제 앞의 예제를 조금 수정해서 GPU를 이용해서 덧셈을 수행하는 예제를 만들어보자.

<코드 4> cudaAdd.cu

```
#include <stdio.h>

/* 덧셈을 위한 커널 */
__global__ void add(int *a, int *b, int *sum) {
   *sum = *a + *b;
}

int main(int argc, char** argv)
{
   int a = 2, b = 4, sum;
   int *dev_a, *dev_b, *dev_sum;       /* 변수 a, b, sum을 디바이스로 복사하기 위한 변수 */
   int size = sizeof( int );           /* 정수형을 저장하기 위한 크기 */

   /* 변수 a, b, sum을 위한 디바이스의 공간 확보 */
   cudaMalloc( (void**)&dev_a, size );
   cudaMalloc( (void**)&dev_b, size );
   cudaMalloc( (void**)&dev_sum, size );
```

```
    /* 디바이스로 복사 */
    cudaMemcpy( dev_a, &a, size, cudaMemcpyHostToDevice );
    cudaMemcpy( dev_b, &b, size, cudaMemcpyHostToDevice );

    /* GPU에서 커널 함수 add() 실행 */
    add<<< 1, 1 >>>( dev_a, dev_b, dev_sum );

    /* GPU에서 계산한 결과값을 호스트의 sum 변수로 복사 */
    cudaMemcpy( &sum, dev_sum, size, cudaMemcpyDeviceToHost );

    /* 사용이 끝난 메모리 공간 해제 */
    cudaFree( dev_a );
    cudaFree( dev_b );
    cudaFree( dev_sum );

    /* 결과 출력 */
    printf("2 + 4 = %d from CUDA\n", sum);

    return 0;
}
```

일반적으로 CPU와 GPU는 메모리가 다르기 때문에 GPU에서 무언가를 실행해서 CPU로 값을 넘기기 위해서는 다른 방법이 필요하다. 이를 위해서는 GPU에서 메모리를 할당하고 CPU와 GPU 사이에서 값을 전달해야 한다.

CUDA에서도 메모리 할당을 위해서 cudaMalloc( ), cudaFree( ), cudaMemcpy( )와 같은 함수들을 제공하고 있다.

<표 4> CUDA의 메모리 관련 함수

함수	내용	C 언어와의 비교
cudaMalloc( )	GPU에 메모리를 할당한다.	malloc( )
cudaFree( )	GPU에 할당한 메모리를 해제한다.	free( )
cudaMemcpy( )	CPU와 GPU 사이에서 메모리를 복사한다.	memcpy( )

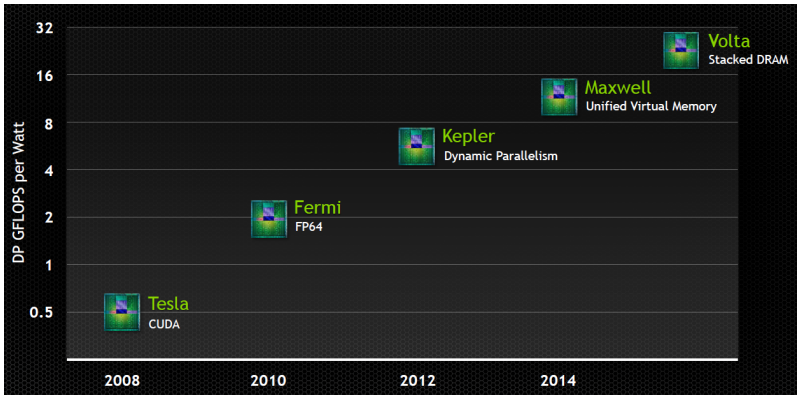
GPU에서 메모리를 할당한 후 cudaMemcpy( ) 함수를 이용해서 복사한다. 메모리의 복사는 2가지 방향이 있는데 호스트(CPU)에서 디바이스(GPU)로 복사할 때에는 인자로 cudaMemcpyHostToDevice를 사용하고, 반대로 디바이스(GPU)에서 호스트(CPU)로 복사할 때에는 cudaMemcpyDeviceToHost를 사용한다. 먼저 앞의 코드를 실행해서 결과를 확인해보자. 결과를 확인해보면 원하는 덧셈이 수행된 것을 볼 수 있다.

```
ubuntu@tegra-ubuntu:~$ nvcc -o cudaAdd cudaAdd.cu
ubuntu@tegra-ubuntu:~$ ./cudaAdd
2 + 4 = 6 from CUDA
```

3. CUDA™를 이용한 병렬 프로그래밍의 기본

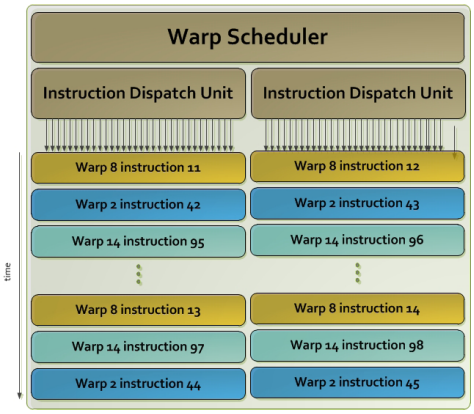
3-1. CUDA와 병렬 처리

NVIDIA에서는 그래픽 처리를 위한 다양한 GPU를 개발해왔다. 하나의 GPU는 다수의 SM(streaming multiprocessors)로 구성되며, 각 SM은 여러 개의 코어들을 가지고 있다는데 GPU의 코어들은 CPU의 코어와는 다른 개념이다.



<그림 5> NVIDIA GPU의 로드맵

NVIDIA Jetson TK1에서 사용하고 있는 GPU는 데슬라 구조로, 테슬라 시리즈에서는 페르미(Fermi) 기반의 구조를 사용한다. SM(Streaming Multiprocessor)는 32개의 코어를 가지고 있으며, 정수와 실수 연산을 처리하기 위한 유닛을 가지고 있다. 동시에 1536개의 스레드를 관리할 수 있는 2개의 워프(Warp) 스케줄러를 가지고 있다.



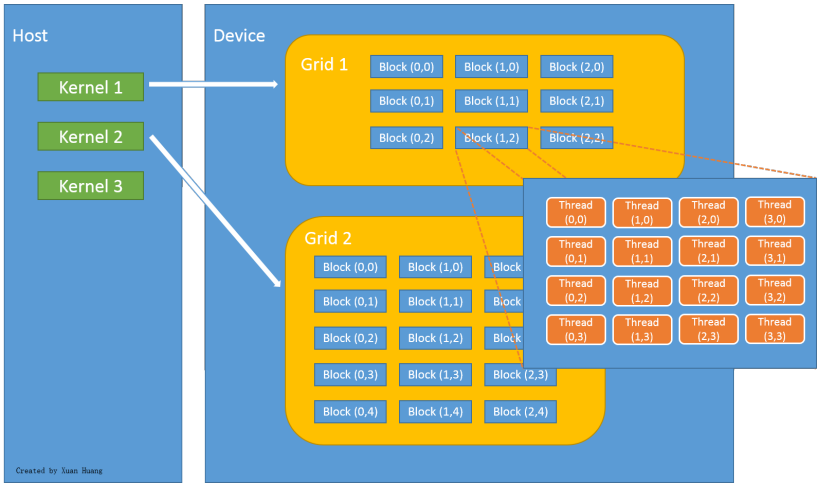
<그림 6> 캐플러의 워프 스케쥴러

SM의 8개의 SP(Streaming Processor)를 가지고 있고 SP는 한 번에 4개의 스레드를 실행할 수 있다. 즉 SM은 한 번에 32개(8 X 4)의 스레드를 실행할 수 있다. 이 32개의 스레드를 묶어 워프(Warp)라고 한다. 페르미 아키텍처에서는 SM안에 워프 스케줄러가 2개로 늘어났기 때문에 SM이 한 번에 2개의 워프를 실행할 수 있다.

3-2. CUDA의 스레드와 그리드 그리고 블록

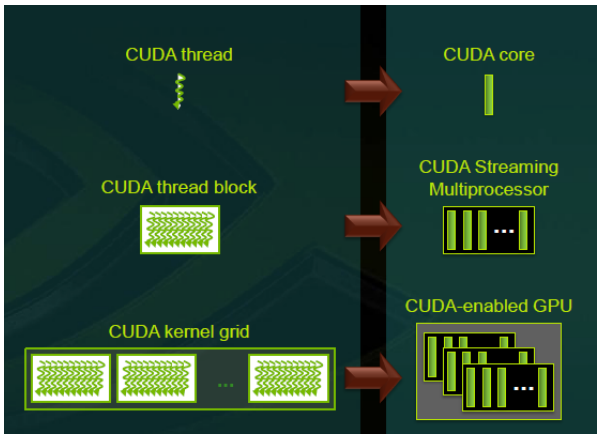
스레드는 커널 함수 처리의 가장 작은 단위이다. GPU에서 커널을 실행하면 스레드(Thread)가 생성되면서 병렬로 처리된다. CUDA의 스레드는 가벼우며(lightweight), 빨리 스위칭되고 계층 구조를 가지고 있다.

CUDA에서는 이러한 스레드를 보다 효율적으로 처리하기 위해서 여러 개념으로 묶어서 사용하는데, CUDA의 스레드가 모여서 블록이 되고, 블록이 모여 그리드를 이루고 하나의 블록은 1~512개의 스레드를 가질 수 있다. 실생활과 비교해보면 스레드가 대학교에서 일을 하는 하나의 학생이라면, 블록은 과로 생각할 수 있으며, 그리드는 하나의 단대와 비교해볼 수 있다.



<그림 7> CUDA의 스레드, 그리드, 블록[6]

SM은 동시에 여러 개의 블럭들을 수행하는데, 쓰레드 블럭을 생각할 수 있다. 하나의 SM은 하나의 블록에 대응하고, 블록 한 개가 최대 512개의 스레드를 가질 수 있다. 쓰레드 블럭은 워프(Warp) 단위로 실행된다. 각 쓰레드 블럭들은 32개를 단위로 하나 또는 그 이상의 워프로 매핑된다.



<그림 8> 스레드와 하드웨어의 매핑

3-3. CUDA에서 커널의 호출

이제 앞의 CUDA 함수에서 사용한 커널의 호출에 대해서 보다 자세히 살펴보자. CUDA에서 커널의 호출은 함수명<<블록의 수,블록 당 스레드의 수>>(함수의 인자)의 형태를 이용하여 블록과 스레드를 지정한다.

앞의 **hellocuda.cu** 코드를 아래와 같이 수정해서 블럭과 스레드의 관계를 살펴보자. 앞의 코드와 달리 커널에서도 터미널로 출력하도록 했다. 디바이스의 printf( ) 함수에 “from Device”를 추가해서 커널에서 출력하는 것을 표시하였고, 호스트의 printf( ) 함수에는 “from Host”를 추가하였다.

<코드 5> cudaThread.cu

```
#include <stdio.h>

#define NUM_BLOCKS 2
#define NUN_THREAD 1
```

[6] http://www.umbc.edu/hpcf/user-resources/how-to-run-cuda.php 참조

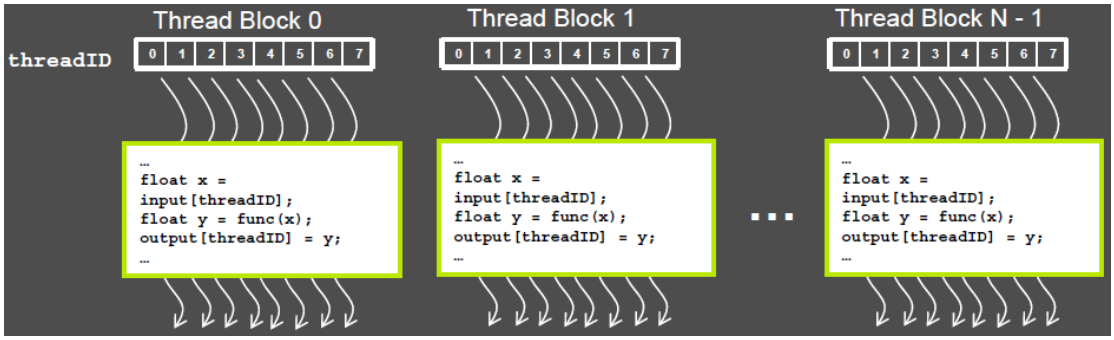
```
__global__ void kernel()
{
    printf("Hello CUDA!(%d Thread in %d Block) from Device\n", threadIdx.x, blockIdx.x);
}

int main(int argc, char **argv)
{

    /* 커널 호출 */
    kernel<<<NUM_BLOCKS, NUN_THREAD>>>();

    /* 커널에서 사용한 printf( ) 함수의 결과를 화면으로 출력(flush) */
    cudaDeviceSynchronize();

    /* Hello CUDA! 출력 */
    printf("Hello CUDA! from Host\n");

    return 0;
}
```

커널은 별도의 스레드로 처리되는데, 스레드는 각각의 인덱스를 가지고 있다. 커널의 쓰레드에 대한 인덱스는 threadIdx를 이용해서 가져올 수 있다. 일반적으로 GPU에서는 3차원을 사용하기 때문에 CUDA도 마찬가지로 threadIdx는 x, y, z의 값을 가지고 있다. 또한 블록에 대한 인덱스는 blockIdx를 이용해서 가져올 수 있다.

<표 5> CUDA의 자동 변수

변수명	내용	비고
dim3[7] gridDim	그리드에서 블록의 크기(dimension)	
dim3 blockDim	블록에서의 스레드의 수	
dim3 blockIdx	블록의 인덱스	
dim3 threadIdx	스레드의 인덱스	

블록에 대한 크기와 인덱스는 gridDim과 blockIdx를 사용하면 가져올 수 있고, 스레드에 대한 크기와 인덱스는 blockDim과 threadIdx을 사용하면 된다.



<그림 9> 블록에서의 스레드의 구분

앞의 코드를 실행해보면 다음과 같은 결과를 볼 수 있다.

```
ubuntu@tegra-ubuntu:~$ nvcc -o cudaThread cudaThread.cu
ubuntu@tegra-ubuntu:~$ ./cudaThread
Hello CUDA!(0 Thread in 0 Block) from Device
Hello CUDA!(0 Thread in 1 Block) from Device
Hello CUDA! from Host
```

[7] dim3 은 정수형의 x, y, z 의 값을 가지고 있는 타입이다.

블록의 수를 2를 지정하고 스레드는 1을 지정해서 실행해보면 블록에서의 스레드의 id가 표시된다. 위의 코드에서 스레드의 수를 3으로 변경해서 실행해보자.

```
ubuntu@tegra-ubuntu:~$ nvcc -o cudaThread cudaThread.cu
ubuntu@tegra-ubuntu:~$ ./cudaThread
Hello CUDA!(0 Thread in 0 Block) from Device
Hello CUDA!(1 Thread in 0 Block) from Device
Hello CUDA!(2 Thread in 0 Block) from Device
Hello CUDA!(0 Thread in 1 Block) from Device
Hello CUDA!(1 Thread in 1 Block) from Device
Hello CUDA!(2 Thread in 1 Block) from Device
Hello CUDA! from Host
```

블록 내에서 스레드가 먼저 실행되고 다음 블록의 스레드가 순차적으로 실행된다. 이렇듯 커널은 GPU에서 병렬로 처리되는 스레드의 배열을 의미하며, CUDA의 스레드는 같은 코드라도 다른 방향으로 실행할 수 있다.

이번달에는 CUDA 의 기본에 대해서 알아보았다. Hello CUDA 를 만들어보면서 기존의 C 언어와 다른 커널 함수에 대해서 알아보았고, 디바이스와 호스트의 차이점에 대해서 살펴보았다. 그리고 CUDA 에서 장치의 정보를 가져오는 예제를 살펴보았다. CUDA 에서 GPGPU 프로그래밍을 위한 메모리 구조와 여러 키워드를 제공한다. 또한 CUDA 를 이용하면 GPU 를 이용한 병렬 프로그래밍을 수행할 수 있는데, 이번달에는 CUDA 에 대한 기본적인 내용을 살펴보았다. 다음달은 CUDA 에 병렬 프로그래밍에 대해서 보다 더 살펴보고 NVIDIA 의 Jetson TK1 을 이용한 OpenCV 프로그래밍에 대해서 살펴보도록 하자.