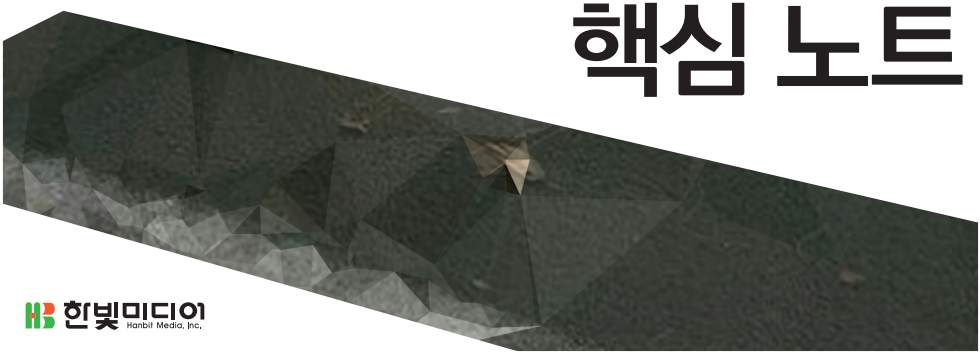
Hanbit RealTime 124

실무자를 위한

C++ AMP 핵심 노트

기창서 지음



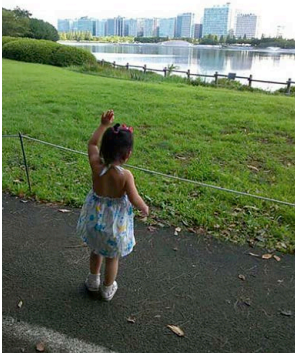
한빛미디어 Hanbit Media, Inc.

실무자를 위한

C++ AMP 핵심 노트

기창서 지음

한빛미디어 Hanbit Media, Inc.



표지 사진 **기창서**

이 책의 표지는 기창서 님이 보내 주신 풍경사진을 담았습니다.
리얼타임은 독자의 시선을 담은 풍경사진을 책 표지로 보여주고자 합니다.

사진 보내기 **ebookwriter@hanbit.co.kr**

실무자를 위한 **C++ AMP 핵심 노트**

초판발행 2016년 1월 22일

지은이 기창서 / **펴낸이** 김태헌
펴낸곳 한빛미디어(주) / **주소** 서울시 마포구 양화로 7길 83 한빛미디어(주) IT출판부
전화 02-325-5544 / **팩스** 02-336-7124
등록 1999년 9월 30일 제10-1779호
ISBN 978-89-6848-797-2 15000 / **정가** 7,000원

총괄 전태호 / **책임편집** 김창수 / **기획·편집** 정지연
디자인 표지/내지 여동일, 조판 최송실
마케팅 박상용, 송경석 / **영업** 김형진, 김진불, 조유미

이 책에 대한 의견이나 오탈자 및 잘못된 내용에 대한 수정 정보는 한빛미디어(주)의 홈페이지나 아래 이메일로 알려주십시오.
한빛미디어 홈페이지 www.hanbit.co.kr / **이메일** ask@hanbit.co.kr

Published by HANBIT Media, Inc. Printed in Korea

지금 하지 않으면 할 수 없는 일이 있습니다.
책으로 펴내고 싶은 아이디어나 원고를 메일(ebookwriter@hanbit.co.kr)로 보내주세요.
한빛미디어(주)는 여러분의 소중한 경험과 지식을 기다리고 있습니다.

저자 소개

지은이_ **기창서**(mpegdvd@gmail.com)

어셈블리 코딩을 좋아하고 비디오 코덱 최적화를 생업으로 삼고 있는 평범한 개발자다. 기계어부터 C# 닷넷 프레임워크까지 모든 코딩을 좋아한다. 코드 최적화를 매우 재미있는 분야라고 생각하고, 부족하지만 GPU를 연산 장치로 이용하는 주제로 지식을 나누고자 노력하고 있다. 현재 영상 보안업체 이노뎁㈜(http://innodep.co.kr/wp/korean/)에서 코덱 개발을 총괄하고 있다.

GPGPU(General-Purpose computing on Graphics Processing Unit) 프로그래밍은 그래픽카드의 연산 능력을 범용적으로 사용하려는 목적에서 출발하고 발전해왔다. 2009년 처음 GPGPU 프로그래밍 환경인 OpenCL을 접했을 때 GeForce 9400으로 실험했는데 그 당시 GPU 연산 코어 수는 16개였다. 지금은 10만 원대 초반 GTX 750만 해도 코어 수가 512개고 CPU에 내장된(6세대 인텔 기준) GPU만 해도 168개(24EU×7스레드)의 코어를 가지고 있다.

이렇게 급속도로 GPU 하드웨어가 발전함에 따라 GPU는 슈퍼컴퓨터가 했던 역할을 대신해가고 있고 빅데이터 처리, 딥러닝, 고감도 인지 등의 산업을 흡수하고 창출해가고 있다. CPU에 GPU를 내장하는 기술도 나날이 발전하여 인텔은 USB보다 조금 큰 베이트레일 스틱 PC를 발표했고 앞으로도 꾸준히 소형화될 것이다.

이런 GPU의 눈부신 발전 속도에 맞게 GPU 프로그래밍 방법도 발전해왔다. 기존 GPU 프로그래밍 방법은 새로운 환경과 언어를 배워야 했지만, C++ AMP는 C++ 언어의 확장으로 비주얼 스튜디오만 있으면 개발할 수 있다.

OpenCL, CUDA, DirectCompute, OpenAcc 등의 기존 개발 환경도 장단점이 있지만, C++ AMP로 작성한 코드를 다시 OpenCL로 작성해보면 C++ 언어에 흡수된 형태가 얼마나 효율적으로 코딩할 수 있게 도와주는지 알 수 있다.

다소 익숙하지 않은 구문들을 쉽게 찾아보고 참고할 수 있기를 바라며 이 책을 집필했다. C++ AMP를 사용할 때 기본이 되는 용어들과 자주 사용하는 구문들을 참고하기 쉽도록 정리하고, GPGPU 프로그래밍을 할 때 키워드나 관용 구문을 암기하지 않고 필요한 코드를 가져다 쓸 수 있게 하여 생산성을 높이고 좀 더 쉽게 개발할 수 있도록 도와줄 것이다.

또한, OpenCL 등 다른 GPU 환경에서 C++ AMP 플랫폼으로 이동하려고 하는 경우에도 도움이 될 것이다. OpenCL에서 간단한 'Hello World'를 처리하는 데도 몇 페이지의 코딩을 해야 하는데, 몇 줄만으로 동일하게 처리할 수 있는 C++ AMP는 GPGPU 프로그래밍 입문에 훌륭한 도구가 될 것이다.

이 책은 실무에서 바로 활용할 수 있게 하는 것이 주목적이므로 특정 알고리즘이나 일반적으로 널리 알려진 성능 최적화는 다루지 않고 C++ AMP 언어 본연의 특성만을 소개하였으며, 모든 예제는 비주얼스튜디오 2015에서 검증하였다.

끝으로 이 책을 쓸 수 있게 직접 도와준 사내 GPU 동아리 회원들께 감사하고, 늦게까지 일할 수 있게 도와준 아내와 몇 달간 놀아주지 못한 딸 지원이에게 미안함과 무한한 고마움을 전한다.

차례

chapter 1

C++ AMP를 위한 C++ 문법

C++ AMP[Accelerated Massive Parallelism]는 비주얼 스튜디오에서 GPGPU 프로그래밍을 쉽게 하기 위한 가장 편리한 방법이지만, C 언어를 주로 사용하거나 최신 C++ 문법과 STL[Standard Template Library]에 익숙하지 않은 경우에는 관련 선행지식이 필요하다. 이를 위해 1장에서는 GPU 연산 함수 제작에 필요한 C++ 문법을 간단히 살펴보겠다.

1.1 함수자

함수자[functor]는 함수 객체[function object]로도 불리며, C의 함수 포인터와 같이 클래스 자체를 함수처럼 사용하는 것을 말한다. 간단히 두 값을 더하는 함수자는 다음과 같다.

[예제 1-1] C++ 함수자 선언과 사용

```
#include <iostream>

class Add {
public:
    int operator()(int a, int b) { return a + b; };
};

int main()
{
    std::cout << "1 + 2 = " << Add()(1, 2) << std::endl;
    return 0;
}
```

실행 결과

```
1 + 2 = 3
```

C++에서 함수자는 C로 보면 변수와 함수를 하나로 묶어서 하나의 기능을 표현하는 훌륭한 도구다. 하지만 예제처럼 아주 단순한 코드를 작성할 때에는 비교적 많은 양의 코딩을 해야 하고 별도의 클래스를 만들어야 해서 가독성을 떨어뜨린다. 이는 다음에 설명할 람다를 이용하면 main 함수 내부에서 조금 더 간결한 코딩을 할 수 있다.

1.2 람다

람다는 익명Anonymous의 함수를 담고 있는 코드 조각이다. 함수자로 작성한 코드를 람다로 다시 표현하면 마치 main 함수 내부에서 함수를 선언한 것처럼 보인다.

[예제 1-2] auto 변수에 람다 코드를 넣어 호출

```
#include <iostream>

int main()
{
    auto Add = [=](int a, int b)->int { return a + b; };
    std::cout << "1 + 2 = " << Add(1, 2) << std::endl;
    return 0;
}
```

실행 결과

```
1 + 2 = 3
```

[예제 1-2]에서 굵게 표시된 부분처럼 단 한 줄로 표현되며 함수자로 작성한 코드보다 훨씬 간결하다. 이때 auto는 컴파일러가 형을 자동으로 결정하는 지역변수 선언 키워드로, 람다로 작성된 익명의 코드 조각을 담을 수 있다.

그럼 다음처럼 C로 작성된 add 함수가 있을 때 이를 람다로 쉽게 변경하는 방법을 알아보자.

```
int add(int a, int b) { return a+b; }
```

변화된 부분을 단계별로 살펴보면 다음과 같다.

① 리턴값 int를 앞에서 가운데로 옮긴다.

```
add(int a, int b) int { return a+b; }
```

② 리턴값을 옮길 때 ->를 앞에 붙인다.

```
add(int a, int b) ->int  { return a+b; }
```

③ 앞에 대괄호([ ])를 붙인다.

```
[=]add(int a, int b) ->int  { return a+b; }
```

[ ]는 람다 내부에서 외부변수를 사용할 때 값과 참조 중 어떤 것을 사용하는지를 표현하는 '캡처'다.

캡처 사용 예

- **[=]** 모두 값으로 사용
- **[&]** 모두 참조로 사용
- **[=, &i]** 기본은 값으로, i 변수는 참조로 사용
- **[&, =i]** 기본은 참조로, i 변수는 값으로 사용

1.3 std::function

std::function은 C 함수를 래핑한 클래스로 볼 수 있는데, add 함수가 있다고 할 때 다음 예제의 세 변수 add0, add1, add2는 같은 역할을 한다.

[예제 1-3] 두 값을 더하는 세 가지 변수 선언 방법

```
#include <iostream>
#include <functional>

using namespace std;

int add(int a, int b) { return a + b; }

int main()
{
    auto add0 = function<int(int, int)>(add);
    auto add1 = function<int(int, int)>([](int a, int b) -> int { return a + b; });
    auto add2 = [](int a, int b) -> int { return a + b; };

    cout << add0(1, 2) << endl;
    cout << add1(1, 2) << endl;
    cout << add2(1, 2) << endl;
    return 0;
}
```

실행 결과

```
3
3
3
```

같은 역할의 세 변수에서 add0, add1은 각각 생성자로 함수 포인터와 람다식을 넣어 만든 변수고, add2는 람다 자체를 담은 변수다. auto 변수는 지역변수로만 사용할 수 있고, 전역변수로 함수 역할을 하는 클래스가 필요하면 function을 사용해야 한다.

1.4 std::vector, std::array

std::vector는 C의 배열처럼 데이터를 모아서 처리하기 위한 C++ 템플릿 라이브러리다. C++11에서 이와 유사한 std::array가 추가되었는데, vector와의 차이는 선언 시 반드시 초기화해야 한다는 점이다.

[vector 사용]

```
#include <vector> vector<int> v(5);
```

[array 사용]

```
#include <array> array<int, 5> a = { 0, 1, 2, 3, 4 };
```

참고로, C++ AMP에는 vector는 없고 array만 있다. C++ AMP에서 array는 선언 시 반드시 초기화할 필요가 없어서 std::array보다는 std::vector에 가깝다고 보면 된다.

1.5 std::for_each

std::for_each는 vector나 array 내부 데이터를 순차적으로 액세스할 수 있는 함수로, C++11에서 추가되었다. PPL이나 C++ AMP에도 for_each가 있고 의미상 유사하므로 익숙해지면 유용하다.

std::for_each의 마지막 파라미터는 unary 함수로, vector나 array 등의 인덱스가 아닌 인덱스가 가리키는 개별 값이 입력으로 들어가는 고정된 형태다.

[예제 1-4] for_each를 이용하여 vector와 array의 내부 데이터를 더하는 예제

```
#include <iostream>
#include <vector>
#include <array>
#include <numeric>
```

```
using namespace std;
int main()
{
    vector<int> v(5);
    array <int, 5> a = { 0, 1, 2, 3, 4 };
    iota(begin(v), end(v), 0); //  set v : 0~5

    int v_sum = 0;
    int a_sum = 0;

    for_each(v.begin(), v.end(), [&](int n) { v_sum += n; });
    for_each(a.begin(), a.end(), [&](int n) { a_sum += n; });

    cout << "vector : 0 + 1 + 2 + 3 + 4 = " << v_sum << endl;
    cout << "array : 0 + 1 + 2 + 3 + 4 = " << a_sum << endl;
}
```

실행 결과

```
vector : 0 + 1 + 2 + 3 + 4 = 10
array : 0 + 1 + 2 + 3 + 4 = 10
```

1.6 nullptr

C++11에서는 포인터의 번지를 NULL로 대입하거나 이용할 때 NULL 대신에 nullptr을 사용한다. 이를 사용하면 컴파일러가 함수 오버로딩 시 상수 0과 NULL 포인터를 구분해 준다.

```
int *p = nullptr;
```

1.7 정리

1장에 나온 문법과 그 쓰임새를 간단히 정리하여 보자.

내용	문법
함수자	`int operator()(int a, int b) { return a + b; };`
람다, auto	`auto add = [=](int a, int b)->int { return a + b; };` (캡처: [=], [&], [=, &i], [&, =i])
std::vector	`vector<int> values(100);`
std::array	`array<int, 5> values = { 0, 1, 2, 3, 4 };`
std::for_each	`for_each(v.begin(), v.end(), [&](int n) { v_sum += n; });`
nullptr	`char *p = nullptr;`