

# 8. 병렬 라이브러리

## 멀티코어 프로그래밍

정내훈

# 목표 및 소개

---

- 차례

- C++11
- OpenMP
- Intel Thread Building Block
- CUDA
- Transactional Memory
- 새로운 언어.

# 목표 및 소개

- C++11 (C++0x)
  - C++의 새로운 표준
  - ISO가 2011년 8월 12일에 승인
  - Visual Studio 2012부터
  - GCC는 4.3에서 4.8에 걸쳐서 구현
  - 멀티쓰레드 프로그래밍 API의 표준화

# 목표 및 소개

## ● C++11

```
#include <thread>
#include <iostream>

void function()
{
    std::cout << "From thread 1" << std::endl;
}

int main()
{
    std::thread t(function);
    t.join();

    std::cout << "From main thread" << std::endl;

    std::cin.ignore();
    return 0;
}
```

# 목표 및 소개

## ● C++11

```
#include <thread>
#include <iostream>
#include <vector>

void hello(){
    std::cout << "Hello from thread " << std::this_thread::get_id()
               << std::endl;
}

int main(){
    std::vector<std::thread> threads;

    for(int i = 0; i < 5; ++i){
        threads.push_back(std::thread(hello));
    }

    for(auto& thread : threads){
        thread.join();
    }
    return 0;
}
```

# 목표 및 소개

## ● C++11

```
#include <thread>
#include <iostream>
#include <vector>

void hello(){
    std::cout << "Hello from thread " << std::this_thread::get_id()
               << std::endl;
}

int main(){
    std::vector<std::thread> threads;

    for(int i = 0; i < 5; ++i){
        threads.push_back(std::thread(hello));
    }

    for(auto& thread : threads){
        thread.join();
    }
    return 0;
}
```

# 목표 및 소개

## ● C++11

```
#include <thread>
#include <iostream>
#include <vector>
#include <mutex>
```

```
class Counter {
public:
    int value;
    std::mutex mtx_lock;
    Counter() {value=0;}
    void increment(){
        std::lock_guard<std::mutex> guard(mtx_lock);
        ++value;
    }
};
```

```
int main()
{
    Counter counter;

    std::vector<std::thread> threads;
    for(int i = 0; i < 5; ++i){
        threads.push_back(std::thread([&counter]() {
            for(int i = 0; i < 10000; ++i){
                counter.increment();
            }
        }));
    }

    for(auto& thread : threads){
        thread.join();
    }

    std::cout << counter.value << std::endl;
    return 0;
}
```

# 목표 및 소개

## ● C++11

– 그리고, 더...

```
#include <atomic>

std::atomic_thread_fence(std::memory_order_seq_cst);
```

```
#include <atomic>

bool atomic_compare_exchange_strong(
    atomic_int *mem,
    int * old_v,
    new_v);
```

```
atomic_int a;
atomic <bool> b;

a++
```



# 목표 및 소개

## ● C++11

— 성능 때문에 다루지 않은 것들.

- 시스템 호출로 구현되기 때문에 또는 블럭킹으로 구현되므로
- `condition_variable`, `future`, `promise`, `async`
- `coroutine`

# 목표 및 소개

---

- 차례

- C++11

- OpenMP

- Intel Thread Building Block

- CUDA

- Transactional Memory

- 새로운 언어.

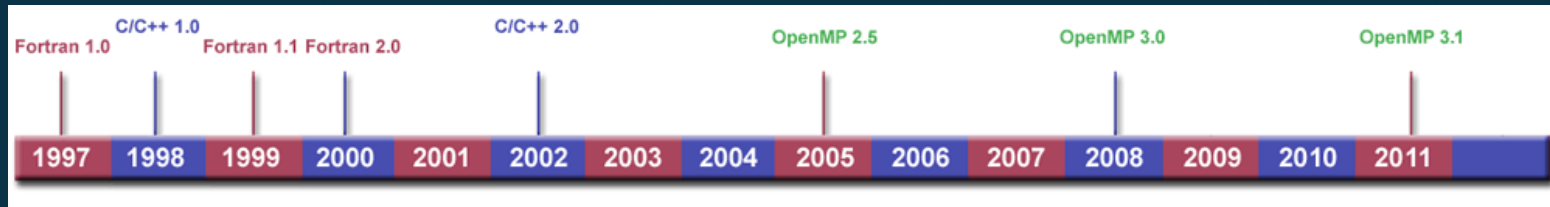
# OpenMP

- C와 C++, FORTRAN에서 병렬프로그램을 가능하게 해주는 API
- 내부적으로 Multi-Thread, 공유메모리를 사용한다.
- 컴파일러 디렉티브(Directive)와 함수, 변수로 구성되어 있다.
- 표준으로 지정되어있어서 대부분의 컴파일러에서 구현되어 있다.
- <http://openmp.org>

# OpenMP

## ● 역사

- 90년대 초 SMP 컴퓨터에서 FORTRAN의 loop를 병렬수행 하기 위해 개발
  - ANSI X3H5 표준제안 1994년
- 1997년부터 OpenMP ARB에서 표준화시작
- 2015년 OpenMP 4.5 출시



“<https://computing.llnl.gov/tutorials/openMP/>”

# OpenMP

- OpenMP의 특징

- 분산 메모리에서는 사용할 수 없다.
- 최상의 공유메모리 사용 패턴을 보장하지 않는다.
- Data Dependency, Data Race, Deadlock검사는 프로그래머가 해야 한다.
- 컴파일러가 알아서 기존 프로그램을 병렬로 변환해 주지는 않는다. 어느 부분을 어떻게 병렬화 할지를 프로그래머가 지정해 주어야 한다.

# OpenMP

- 컴파일러

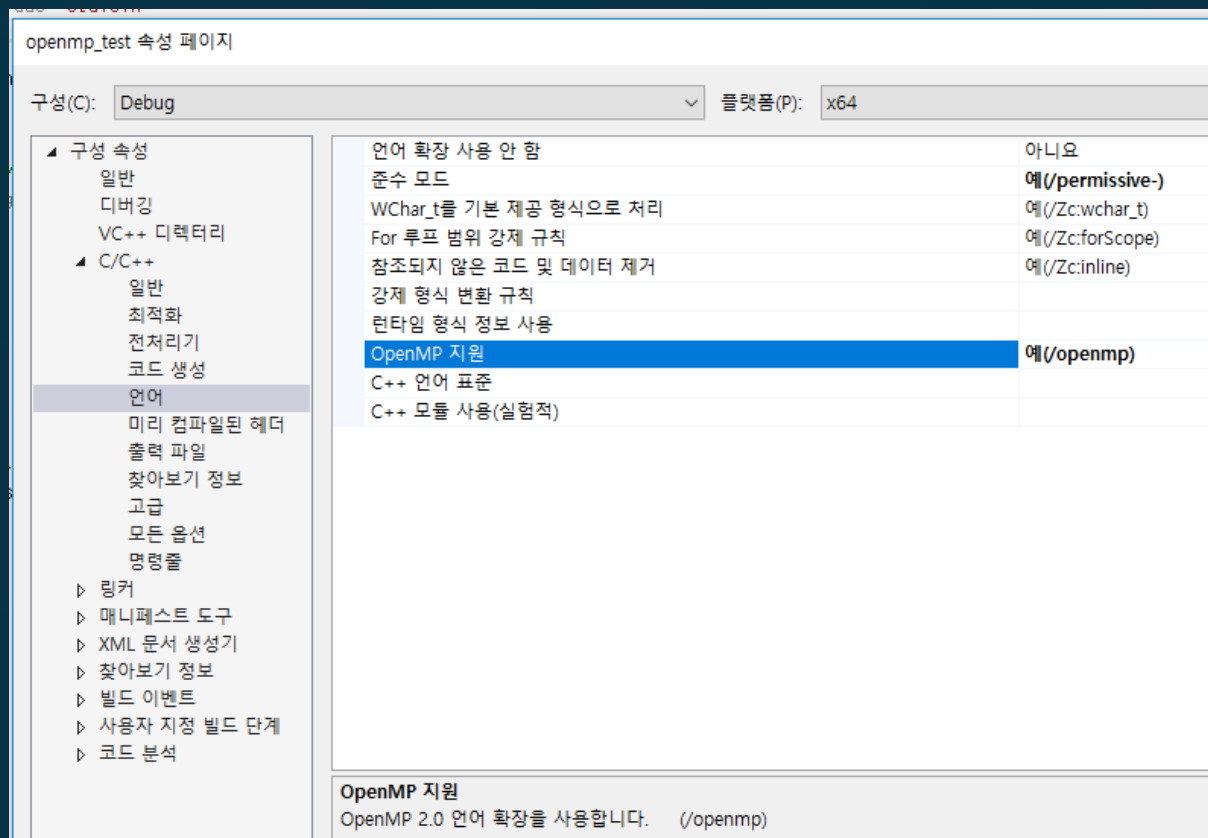
- 리눅스의 gcc는 OpenMP 4.0을 지원한다.
- 컴파일 할 때 “-fopenmp” 옵션을 넣으면 된다.

- 확인

- “top -H” 명령으로 멀티쓰레드 실행을 확인해 볼 수 있다.

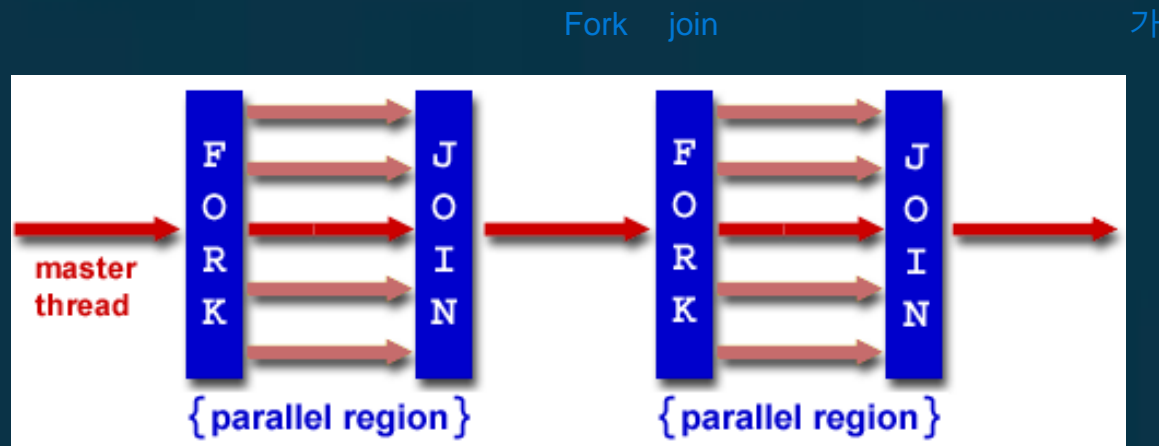
# OpenMP

- 컴파일러
  - 비주얼 스튜디오도 지원



# OpenMP (2023-화금)

- 프로그래밍 모델 (1/2)
  - 공유메모리에서의 멀티쓰레드 구현
  - 자동으로 병렬화를 하지 않고 사용자가 병렬화를 지정
  - Fork-Join 모델



“<https://computing.llnl.gov/tutorials/openMP/>”



# OpenMP

- 프로그래밍 모델 (2/2)
  - 컴파일러 디렉티브에 의존
  - Nesting 가능 (병렬화의 겹침 허용)
  - 동적 스레드 할당
  - 메모리 일관성은 보장하지 않는다. 필요하다면 FLUSH 명령을 사용해야 한다.

# OpenMP

## ● Code Structure

```
#include <omp.h>

main () {
  int var1, var2, var3;

  Serial code
    .
    .
    .

  Beginning of parallel section. Fork a team of threads.
  Specify variable scoping

  #pragma omp parallel private(var1, var2) shared(var3)
  {

    Parallel section executed by all threads
      .
      .
      .

    All threads join master thread and disband

  }

  Resume serial code
    .
    .
    .
}
```

# OpenMP

## ● Directive의 구조

| #pragma omp | directive-name   | [clause, ...]                      |
|-------------|------------------|------------------------------------|
| 필수          | 하나의 directive 필수 | 옵션, 순서에 상관없이 여러 개의 clause가 올 수 있다. |

- directive뒤에는 반드시 C언어의 Block이 와야 한다.
  - Block : {...}

Block "{}"

# OpenMP

- parallel Directive

```
#pragma omp parallel [clause ...] newline
    if (scalar_expression)
    private (list)
    shared (list)
    default (shared | none)
    firstprivate (list)
    reduction (operator: list)
    copyin (list)
    num_threads (integer-expression)
```

structured\_block

가,

가?

private:

shared:

가,

가?

# OpenMP

- parallel Directive

- 멀티 쓰레드가 생성되서 해당되는 블록의 코드를 병렬로 수행한다.  
workwr thread
- 블록의 끝에서 모든 쓰레드의 종료를 확인한 후  
진행을 계속 한다.

# OpenMP

## ● parallel Directive 실습

```
#include <omp.h>
#include <stdio.h>

int main ()
{
    int nthreads, tid;

    /* Fork a team of threads with each thread having a private tid variable */
    #pragma omp parallel private(tid)
    {
        /* Obtain and print thread id */
        tid = omp_get_thread_num();
        printf("Hello World from thread = %d\n", tid);

        /* Only master thread does this */
        if (tid == 0) {
            nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
        }
    } /* All threads join master thread and terminate */
}
```

# OpenMP

- parallel Directive 실습 - 2
  - 1억을 만드는 프로그램을 parallel Directive를 사용해 병렬로 구현하고 그 실행속도를 측정하라.
    - “omp\_get\_num\_threads ()” 함수가 필요할 것이다.
    - “#pragma omp critical”로 data race를 막을 수 있다.

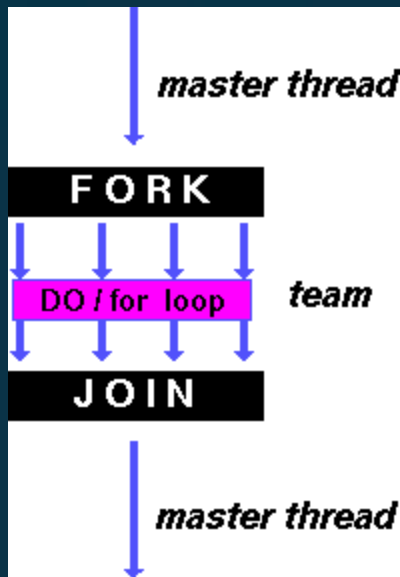
# OpenMP

- 작업 (Work)
  - 병렬성을 지정하는 프로그램의 단위
- 작업 분배 지정 (Work-Sharing Constructs)
  - 작업을 분배하는 방식
    - Do/for : 루프를 여러 쓰레드가 나누어 수행
    - SECTIONS : 프로그램 블록으로 나뉘어진 작업들을 여러 쓰레드가 나누어 수행
    - SINGLE : 한 개의 쓰레드가 전담해서 수행

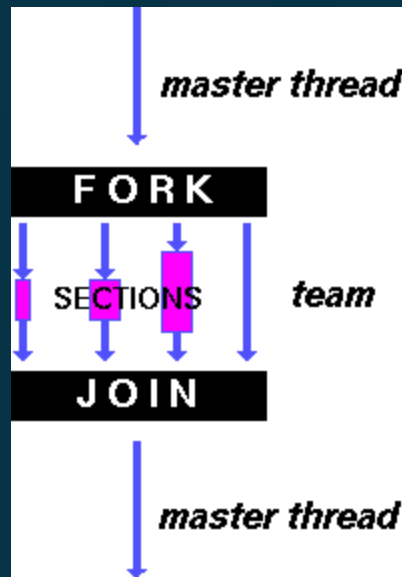


# OpenMP

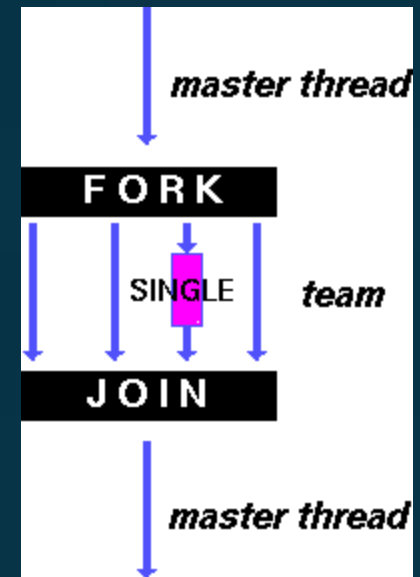
- 작업 분배 지정 (Work-Sharing Constructs)



Do/For



Sections



Single

# OpenMP

- Do/For

```
#include <omp.h>
#define CHUNKSIZE 100
#define N      1000

main ()
{
    int i, chunk;
    float a[N], b[N], c[N];

    /* Some initializations */
    for (i=0; i < N; i++)  a[i] = b[i] = i * 1.0;
    chunk = CHUNKSIZE;

    #pragma omp parallel shared(a,b,c,chunk) private(i)
    {

        #pragma omp for schedule(dynamic,chunk) nowait
        for (i=0; i < N; i++)
            c[i] = a[i] + b[i];
    } /* end of parallel section */
}
```

# OpenMP

- Do/For

- “schedule” 루프가 병렬로 실행하는 방식을 지정
  - STATIC : 모든 쓰레드가 공평한 개수의 묶음을 실행
  - DYNAMIC : 먼저 끝난 쓰레드가 다음 묶음을 실행
- 묶음 (chunk)
  - 쓰레드가 한번에 실행할 루프의 횟수
- “nowait” 쓰레드의 실행을 동기화 하지 않음
  - 먼저 끝난 쓰레드가 다른 쓰레드의 작업종료를 기다리지 않고 다음 작업 실행

# OpenMP

- 주의 사항

- For 병렬화에서 data dependency를 검사하지 않는다.
  - $c[i] = c[i-1] + a[i];$  // 오동작.
  - data dependency는 프로그래머의 책임.

# OpenMP

## ● SECTIONS

```
#include <omp.h>
#define N      1000

main ()
{
    int i;
    float a[N], b[N], c[N], d[N];
    /* Some initializations */
    for (i=0; i < N; i++) {
        a[i] = i * 1.5;  b[i] = i + 22.35;
    }

    #pragma omp parallel shared(a,b,c,d) private(i)
    {
        #pragma omp sections nowait
        {
            #pragma omp section
            for (i=0; i < N; i++) c[i] = a[i] + b[i];

            #pragma omp section
            for (i=0; i < N; i++) d[i] = a[i] * b[i];
        } /* end of sections */
    } /* end of parallel section */
}
```

# OpenMP

- 그밖에 쓸만한 Directive

- CRITICAL

```
#pragma omp critical  
sum = sum + 2;
```

- ATOMIC

```
#pragma omp atomic  
x[index[i]] += y;
```

# 목표 및 소개

- 차례
  - OpenMP
  - Intel Thread Building Block
  - CUDA
  - Transactional Memory
  - 새로운 언어.

# TBB

- Intel Thread Building Block
- 스레드 사용에 편리한 여러 API를 가짐
- Task관리 기능 포함
- Intel CPU에서 동작함
  - 비공식적인 Android/ARM버전도 존재함.
- 최근 OneAPI라는 프로젝트에 통합되었다.
- 최근 C++11과의 연동이 강조되고 있다. 특히 램다.
- 공식 홈페이지
  - <https://software.intel.com/en-us/tbb>
    - 설치 프로그램을 다운받을 수 있음.
  - <http://threadingbuildingblocks.org/>



# TBB

## ● TBB의 기능 들

### — Loop Parallelizer

- #pragma를 사용하지 않고 고유의 함수를 사용

### — Containers Non blocking lazy가 blocking

- STL과 유사한 형태의 멀티쓰레드 non-blocking container를 제공

### — Mutual Exclusion

- 다양한 형태의 lock을 제공

### — 메모리 일관성 지시 (지원에서 제거 됨)

### — 메모리 할당자 (2023 화금)

- 멀티쓰레드 상에서의 효율적인 메모리 할당자
- 기존의 메모리 할당자를 교체

### — Task 스케줄링

# TBB

---

- 매뉴얼

- [oneTBB — oneAPI Specification 1.2-rev-1 documentation](https://spec.oneapi.io/versions/latest/elements/oneTBB/source/nested-index.html)  
(<https://spec.oneapi.io/versions/latest/elements/oneTBB/source/nested-index.html>)

# TBB

- Loop Parallelizer

- #pragma 형태가 아니므로 사용자가 문제를 TBB가 멀티쓰레드를 적용할 수 있는 형태로 변형해야 한다.

- 루프의 범위를 지정할 수 있어야 한다.
    - TBB가 호출할 operator를 등록해야 한다.

# TBB

## ● Loop Parallelizer (1/3)

- Loop 본체를 Foo(int i)와 같은 함수로 변환
- 다음과 같은 형태의 함수로 만든 이후

```
for(int i=0; i < n; ++i ) {  
    ... = a[i];  
    ...;  
}
```



```
for(int i=0; i < n; ++i ) Foo(a[i]);
```



```
void SerialApplyFoo( float a[], int n ) {  
    for(int i=0; i!=n; ++i )  
        Foo(a[i]);  
}
```

# TBB

## ● Loop Parallelizer (2/3)

—TBB용 클래스로 변환한다.

- `blocked_range`만큼만 실행하는 객체

```
#include "tbb/tbb.h"
using namespace tbb;
class ApplyFoo {
    float *const my_a;
public:
    void operator()( const blocked_range<int>& r ) const {
        float *a = my_a;
        for(int i=r.begin(); i!=r.end(); ++i )
            Foo(a[i]);
    }
    ApplyFoo( float a[] ) : my_a(a)
    {}
};
```

# TBB

- Loop Parallelizer (3/3)
  - TBB Parallelizer를 호출한다.

```
parallel_for(blocked_range<int>(0,n), ApplyFoo(a));
```

# TBB

- 더 간단한 Loop Parallelizer

```
for (auto i = first; i < last; i += step) f(i);
```



```
parallel_for(first, last, step, f);
```

- Step은 생략 가능

— f에 람다를 사용

# TBB

- Loop Parallelizer
  - 램다를 사용한 구현

```
#include <tbb\parallel_for.h>
#include <iostream>

using namespace std;
using namespace tbb;

atomic_int sum = 0;

int main()
{
    int n = 50000000;

    parallel_for(0, n, [&](int i) {
        sum += 2;
    });
    cout << "Sum = " << sum << endl;
}
```



# 실습

- 앞의 TBB 프로그램을 4개 쓰레드의 멀티쓰레드 프로그램과 속도 비교를 해보자.

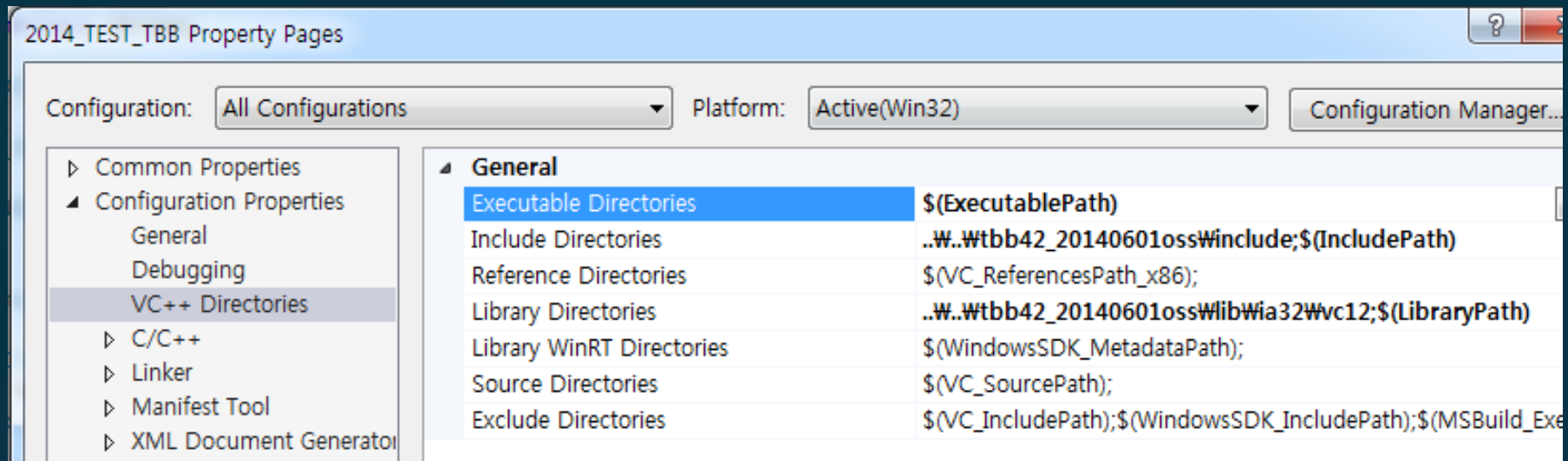
# 실습

- TBB 설치

- <https://www.threadingbuildingblocks.org/>
- 위 사이트에서 다운 받아서 프로젝트 디렉토리에 압축 풀기.
- tbb.dll을 실행디렉토리에 추가.

# 실습

## ● TBB 연결



|                           |  |
|---------------------------|--|
| Include Directories       | ..\..\wtbb42_20140601ossWininclude;\$(IncludePath)                 |
| Reference Directories     | \$(VC_ReferencesPath_x86);   |
| Library Directories       | ..\..\wtbb42_20140601ossWlibWia32Wvc12;\$(LibraryPath)             |
| Library WinRT Directories | \$(WindowsSDK_MetadataPath);                                       |
| Source Directories        | \$(VC_SourcePath);   |
| Exclude Directories       | \$(VC_IncludePath);\$(WindowsSDK_IncludePath);\$(MSBuild_ExecPath) |

# TBB

## ● 설치

— Visual Studio의 경우 Nuget을 사용하면 편함

intel

**inteltbb.devel.win** 작성자: Intel Corporation, 14.3K개 다운로드

v2021.1.1.133

Package includes header files, win-x64 and win-x86 dynamic and import libraries

— 하지만 현재 버그로 인해서 사용 불가 (2021)

- 현재 버그 수정(2022)

— 대안 : vcpkg를 사용한다.

# TBB

- 설치 : vcpkg
  1. vcpkg를 설치한다.
  2. visual studio installer를 통해 영어언어를 설치한다.
  3. `vcpkg install tbb` 명령으로 설치한다.
  4. `vcpkg integrate install`로 비주얼 스튜디오에 연동시킨다.
- 자세한 내용은 별도 파일 참조
  - TBB\_VCPKG.pdf

# TBB (2024)

- 멀티쓰레드환경에서 사용할 수 있는 다음의 Container를 제공한다.
  - Sequences
    - [concurrent\\_vector](#)
  - Queues
    - [concurrent\\_queue/concurrent\\_bounded\\_queue/concurrent\\_priority\\_queue](#)
  - Unordered associative containers
    - [concurrent\\_hash\\_map](#)
    - [concurrent\\_unordered\\_map/concurrent\\_unordered\\_multimap](#)
    - [concurrent\\_unordered\\_set/concurrent\\_unordered\\_multiset](#)
  - Ordered associative containers
    - [concurrent\\_map/concurrent\\_multimap](#)
    - [concurrent\\_set/concurrent\\_multiset](#)
- Lock-Free혹은 Fine-grained locking으로 구현되어  
고성능을 발휘한다.

# TBB

- Container : `concurrent_unordered_map`
  - STL의 `unordered_map`과 비슷
  - `insert()`와 `find()`, `count()`, `size()`, `at()`이 thread safe하다.
  - `erase()`가 thread\_safe하지 않다.
    - 따라서 데이터를 넣었다 뺐다 하는 용도로는 사용 못함.

# TBB

## ● String의 출현 개수를 세기 위한 프로그램

```
const size_t N = 1000000;  
string Data[N];  
void CountOccurrences() {  
    unordered_map<string, int> table;  
    for (auto &s : Data)  
        table[s]++;  
    for( auto &t : table) cout << t.first << " : " << t.second << endl;  
}
```



```
void CountOccurrences() {  
    // Construct empty table.  
    concurrent_unordered_map<string, int> table;  
    // Put occurrences into the table  
    parallel_for(size_t(0), N, [&table](int i) {  
        table[Data[i]]++;  
    });  
    // Display the occurrences  
    for( auto &t : table) cout << t.first << " : " << t.second << endl;  
}
```



# TBB

- Container : `concurrent_hash_map`
  - `find()`, `insert()`, `remove()` 메소드를 제공한다.
  - 멀티쓰레드 환경에서는 컨테이너의 내용이 수시로 변할 수 있으므로 모든 자료 접근은 `accessor`를 통해 이루어진다.
    - `accessor`는 일종의 스마트 포인터이다.
    - 읽기만 하고 수정하지 않을 경우 `const_accessor`를 사용하는 것이 좋다.

# TBB

- String의 출현 개수를 세기 위한 자료 구조

```
typedef concurrent_hash_map<string,int> StringTable;
```

- Parallel\_For를 위한 실행 객체

```
struct Tally {  
    StringTable& table;  
    Tally( StringTable& table_ ) : table(table_) {}  
    void operator()( const blocked_range<string*> range ) const {  
        for( string* p=range.begin(); p!=range.end(); ++p ) {  
            StringTable::accessor a;  
            table.insert( a, *p );  
            a->second += 1;  
        }  
    }  
};
```

# TBB

- Container : `concurrent_vector`
  - Shrink는 불가능하다.
  - `push_back()`, `grow_by()`, `grow_to_at_least()`, `size()` 메소드를 제공한다.
  - `clear()`, `swap()`, `resize()`, `reserve()` 메소드는 병렬수행이 불가능하니 꼭 다른 메소드와 동시에 호출되지 않도록 해야한다.
  - 원소들이 연속된 주소에 있지 않으므로 일반적인 `pointer` 연산은 불가능하다.
  - 원소를 읽을 때 원소가 생성 중일 수 있으므로 읽기 전에 생성완료 여부를 확인하도록 프로그래밍해야 한다.

# TBB

- Container : `concurrent_queue`  
`concurrent_priority_queue`
  - `push()`, `try_pop()` 메소드를 제공한다.
  - `try_pop()`을 제공하는 이유는 `empty()`호출이 `pop()`의 성공을 보장하지 않기 때문이다.
  - Thread unsafe 함수들
    - `unsafe_size()`, `empty()`, `clear()`, `swap()`

# TBB

- Mutual Exclusion

- `scoped_lock` (C++11의 `lock_guard`)

- `lock`을 선언하면 선언된 블록을 빠져 나올때 자동적으로 `unlock`이 된다.

- 예외상황 프로그래밍 편리

- 실수로 `unlock`을 하지 않는 경우를 제거

- C++17의 `std::scoped_lock`과는 다르다.

- 다양한 locking을 지원한다.

- `RWlocking`을 지원한다.

- 이름에 `_rw_`가 붙어 있다.

- C++14의 `std::shared_mutex`과 동일

# TBB

## ● Mutual Exclusion : 사용 예

```
Node* FreeList;
typedef spin_mutex FreeListMutexType;
FreeListMutexType FreeListMutex;
Node* AllocateNode() {
    Node* n;
    {
        FreeListMutexType::scoped_lock lock(FreeListMutex);
        n = FreeList;
        if( n )
            FreeList = n->next;
    }
    if( !n )
        n = new Node();
    return n;
}
void FreeNode( Node* n ) {
    FreeListMutexType::scoped_lock lock(FreeListMutex);
    n->next = FreeList;
    FreeList = n;
}
```

# TBB

- Mutual Exclusion : 종류
  - Scalable : busy waiting을 없애 CPU낭비를 막는다. overhead가 크다.
  - Fair : Critical Section에 도착한 순서대로 lock을 얻는다.
  - Recursive : 같은 스레드는 lock을 다중으로 얻을 수 있다. recursive알고리즘에서 편리
  - Long wait : 오래 기다리고 있을 경우
    - yield : 같은 프로세스의 다른 스레드 실행
    - block : 깨워 줄 때 까지 멈춤

# TBB

## ● Mutual Exclusion : 종류

| Mutex                     | Scalable     | Fair | Recursive | Long Wait | Size          |
|---------------------------|--------------|------|-----------|-----------|---------------|
| spin_mutex                | no           | no   | no        | yields    | 1 byte        |
| mutex                     | ✓            | no   | no        | blocks    | 1 byte        |
| speculative_spin_mutex    | HW dependent | no   | no        | yields    | 2 cache lines |
| queuing_mutex             | ✓            | ✓    | no        | yields    | 1 word        |
| spin_rw_mutex             | no           | no   | no        | yields    | 1 word        |
| spin_rw_mutex             | ✓            | no   | no        | blocks    | 1 word        |
| speculative_spin_rw_mutex | HW dependent | no   | no        | yields    | 3 cache lines |
| queuing_rw_mutex          | ✓            | ✓    | no        | yields    | 1 word        |
| null_mutex                | moot         | ✓    | ✓         | never     | empty         |
| null_rw_mutex             | moot         | ✓    | ✓         | never     | empty         |

<https://www.intel.com/content/www/us/en/develop/documentation/onetbb-documentation/top/onetbb-developer-guide/mutual-exclusion/mutex-flavors.html>

speculative : Transactional Memory



# TBB

- Mutual Exclusion : RW Lock

- 때에 따라 공유자원을 여러 스레드에서 동시에 읽는 것이 문제가 되지 않을 수 있다

```
#include <tbb/spin_rw_mutex.h>           // 헤더

tbb::spin_rw_mutex  glock;                // 선언

// Writer Lock
glock.lock();  glock.unlock();

// Reader Lock
glock.shared_lock();  glock.shared_unlock();
```

# 실습

- RWLOCK 성능 비교
  - Linked List의 Coarse Grained List를 RWLOCK으로 구현한 것과 성능 비교를 해보자. 이 때 Contains의 비중을 증가 시키자.

# 실습

## ● RWLOCK 성능 비교

| Coarse | RW-Lock | 60% Read | 80% Read | 98% Read | 98% C++14 shared_mutex |
|--------|---------|----------|----------|----------|------------------------|
| 2272   | 2251    | 2108     | 3090     | 2704     | 2913                   |
| 2724   | 3316    | 2910     | 2388     | 1354     | 2028                   |
| 3463   | 4521    | 3072     | 2506     | 622      | 2095                   |
| 11744  | 5997    | 3930     | 2883     | 676      | 1616                   |
|        |         |          |          |          |                        |

# TBB

- 메모리 할당자

- STL의 `std::allocator`와 유사한 memory allocator template를 제공.
  - `scalable_allocator<T>`, `cache_aligned_allocator<T>`
- 직렬 프로그램에서 고안된 memory allocator들은 single share pool에 동시에 하나의 스레드만 접근 가능한 문제가 있다.
  - `scalable_allocator<T>`는 이러한 병목 현상을 피할 수 있게 해준다.
- 두 개의 스레드가 같은 cache line을 사용할 때 문제가 있다.
  - `cache_aligned_allocator<T>`는 잘못된 cache line 공유를 막아주는 것을 보장.

# TBB

## ● 메모리 할당자

– Windows와 Linux 시스템에서, 기본 동적 메모리 할당 함수들은 Intel TBB의 할당 함수들로 자동적으로 대체되어 호출된다

- C 라이브러리 : malloc, calloc, realloc, free
- C++ : new와 delete

– Windows 사용법

- #include "tbb/tbbmalloc\_proxy.h" 추가

– Linux 사용법

- LD\_LIBRARY\_PATH or add it to /etc/ld.so.conf. 변경
- 실행 전에 LD\_PRELOAD 변경

```
# Set LD_PRELOAD so that loader loads release version of proxy
LD_PRELOAD=libtbbmalloc_proxy.so.2
# Link with release version of proxy and scalable allocator
g++ foo.o bar.o -ltbbmalloc_proxy -ltbbmalloc -o a.out
```

# 실습

- TBB 메모리 관리자 성능 비교
  - Lock-Free Stamped Queue의 성능을 비교해 보자.
  - 다음 DLL이 필요함

```
tbbmalloc.dll  
tbbmalloc_proxy.dll
```

# TBB

- 태스크 스케줄링

- 작업을 여러 개의 Task로 나누어서 병렬로 처리하는 방식
- Parallel\_for와 달리 작업끼리 서로 연관관계가 있거나 작업이 동적으로 추가되는 경우도 다룰 수 있음.
- 사용상 주의
  - 하나의 task 관리 객체에 많은 task를 넣어 주는 것은 비효율적이다, task관리 객체를 로컬 객체로 생성하고 소수의 task를 넣는 방식으로 구현하는 것을 권장.

# TBB

## ● 사용법

— #include <tbb/task\_group.h>

- tbb::task\_group [객체명]

— 메소드

- run() : 함수/실행객체/람다를 별도 task로 실행
- run\_and\_wait() : run(); wait(); 연속 호출과 동일
  - 좀더 효율적, run()을 현재 스레드에서 실행
- wait() : 모든 task가 종료하기를 기다림
- cancel() : 모든 task를 취소 시킴
- is\_canceling() : cancel이 진행 중이면 true



# TBB

## ● 태스크 스케줄링 : 예제

```
long SerialFib( long n ) {  
    if( n<2 )  
        return n;  
    else  
        return SerialFib(n-1)+SerialFib(n-2);  
}
```

# TBB

## ● 태스크 스케줄링 : 예제

```
#include <tbb/task_group.h>

int parallel_fib(int n)
{
    if (n < 30)
        return serial_fibo(n);
    else {
        int x, y;
        tbb::task_group g;
        g.run([&] {x = parallel_fib(n - 1); }); // spawn a task
        g.run([&] {y = parallel_fib(n - 2); }); // spawn another task
        g.wait(); // wait for both tasks to complete
        return x + y;
    }
}
```

# 목표 및 소개

- 차례
  - OpenMP
  - Intel Thread Building Block
  - CUDA
  - Transactional Memory
  - 새로운 언어.

# CUDA

---

- Computer Unified Device Architecture
- NVIDIA에서 2007년에 발표
- 대규모 병렬 처리를 GPU에서 수행함
  - GPU가 CPU보다 몇백배 빠름
  - GPGPU (General Purpose GPU)의 일종
- 단점 : Nvidia 하드웨어만 지원.

# CUDA

## ● Core 개수 비교 : 3090 [10496개 코어]



### 1 MSI 지포스 RTX 3090 슈프리미 X D6X 24GB 트라이프로저 2S

RTX 3090 / 8nm / 부스트클럭: 1860MHz / 스트림 프로세서: 10496개 / PCIe4.0x16 / GDDR6X(DDR6X) / 출력단자: HDMI2.1, DP1.4 / 부가기능: 제로팬(0-dB기술), 멀티 VGA, 8K 해상도 지원, 4K 해상도 지원, HDR 지원, Dual BIOS, HDCP 2.3, HDCP 지원 / 사용전력: 최대 420W / 정격파워 850W 이상 / 전원 포트: 8핀 x3개 / 3개 팬 / 가로(길이): 336mm / 백플레이트 / LED 라이트 / MYSTIC LIGHT / 그래픽카드 지지대 포함

관련기사 MSI코리아, 지포스 RTX 30 시리즈 그래픽 탑재 게이밍 구매 시 최신 게임 증정

등록일 2020.11. | 상품의견 242건 | 브랜드로그 | 관심상품

|      |            |
|------|------------|
| 11번가 | 3,774,910원 |
| G마켓  | 3,774,960원 |
| 옥션   | 3,775,020원 |
| 롯데ON | 3,799,000원 |
| 샵다나와 | 3,918,320원 |

150물 ☐

[더보기 >](#)

— RTX 4090 16384

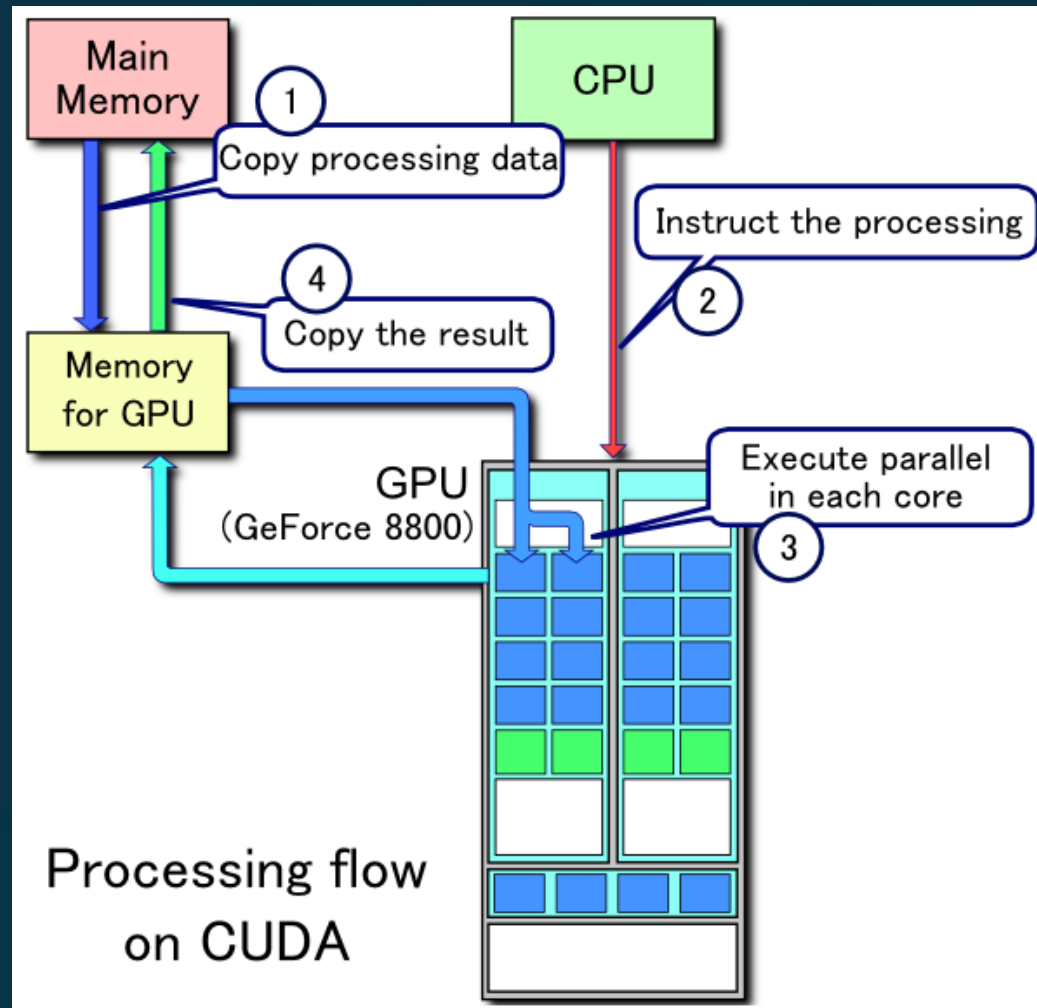
# CUDA

---

- 대안

- DirectCompute : DirectX의 일부분 GPU 벤더에 상관없이 동작
- OpenCL : Apple이 Mac OSX에 구현하고 공개, AMD에서도 잘 동작.

# CUDA



# CUDA

- Example

- [http://on-demand.gputechconf.com/gtc-express/2011/presentations/GTC\\_Express\\_Sarah\\_Tariq\\_June2011.pdf](http://on-demand.gputechconf.com/gtc-express/2011/presentations/GTC_Express_Sarah_Tariq_June2011.pdf)
- GPU에서 실행될 코드

```
__global__ void add(int *a, int *b, int *c)
{
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

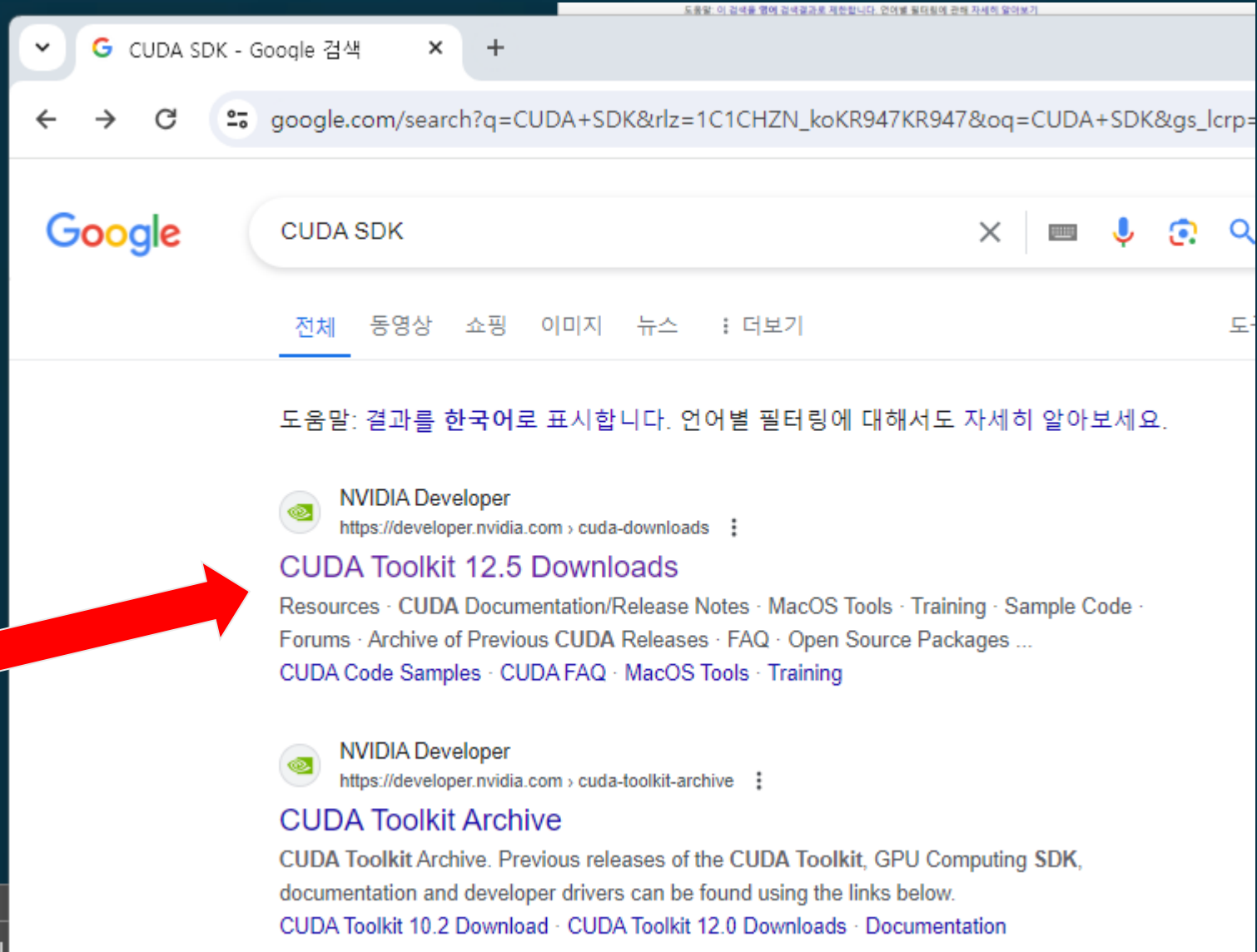


# CUDA

```
#define N 512
int main(void) {
    int *a, *b, *c;           // host copies of a, b, c
    int *d_a, *d_b, *d_c;     // device copies of a, b, c
    int size = N * sizeof(int);
    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);
    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
    // Launch add() kernel on GPU with N blocks
    add<<<N,1>>>>(d_a, d_b, d_c);
    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);
    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

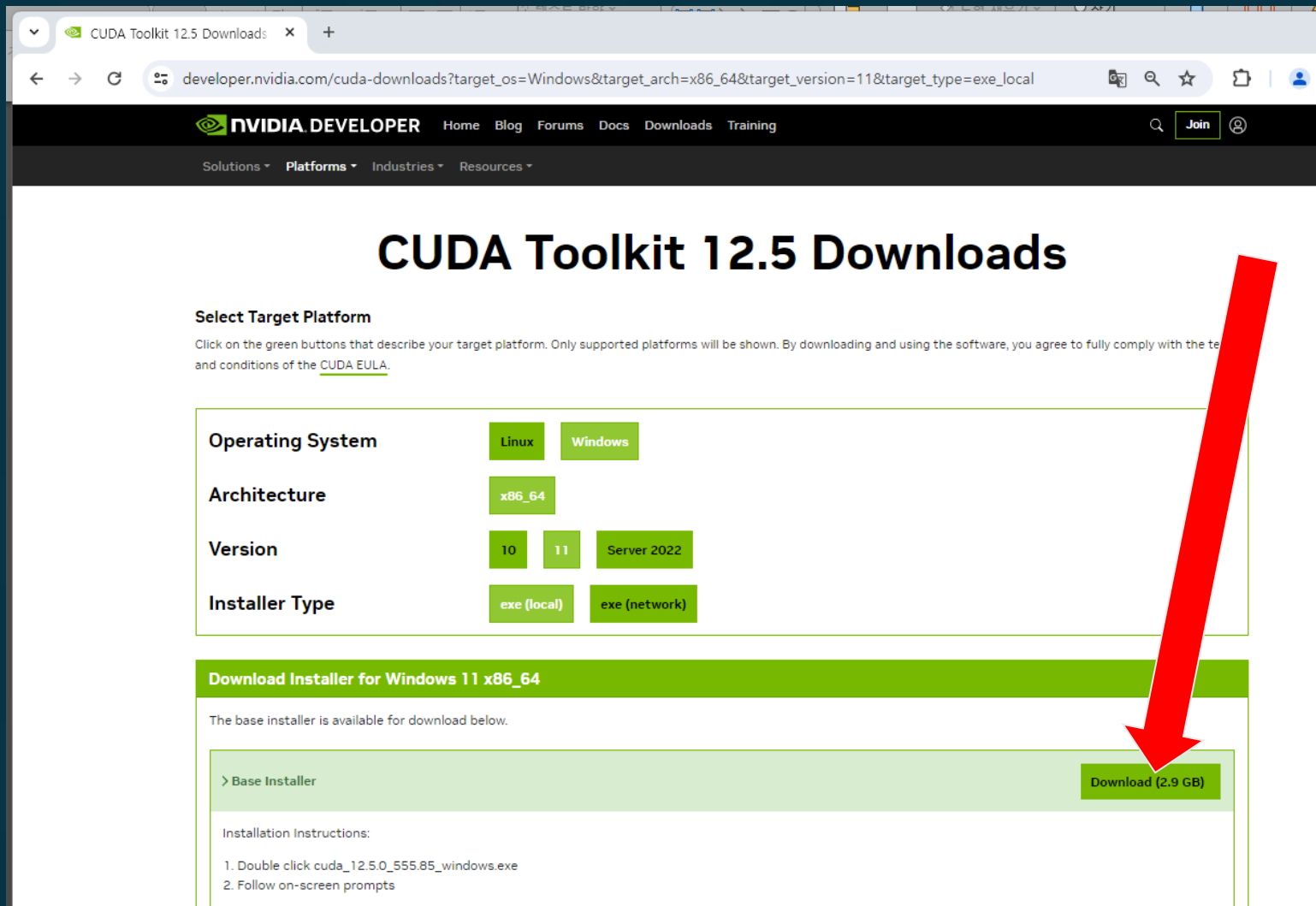
# CUDA

## ● 설치



# CUDA

## ● 설치



CUDA Toolkit 12.5 Downloads

Select Target Platform

Click on the green buttons that describe your target platform. Only supported platforms will be shown. By downloading and using the software, you agree to fully comply with the terms and conditions of the [CUDA EULA](#).

|                  |             |               |             |
|------------------|-------------|---------------|-------------|
| Operating System | Linux       | Windows       |             |
| Architecture     | x86_64      |               |             |
| Version          | 10          | 11            | Server 2022 |
| Installer Type   | exe (local) | exe (network) |             |

**Download Installer for Windows 11 x86\_64**

The base installer is available for download below.

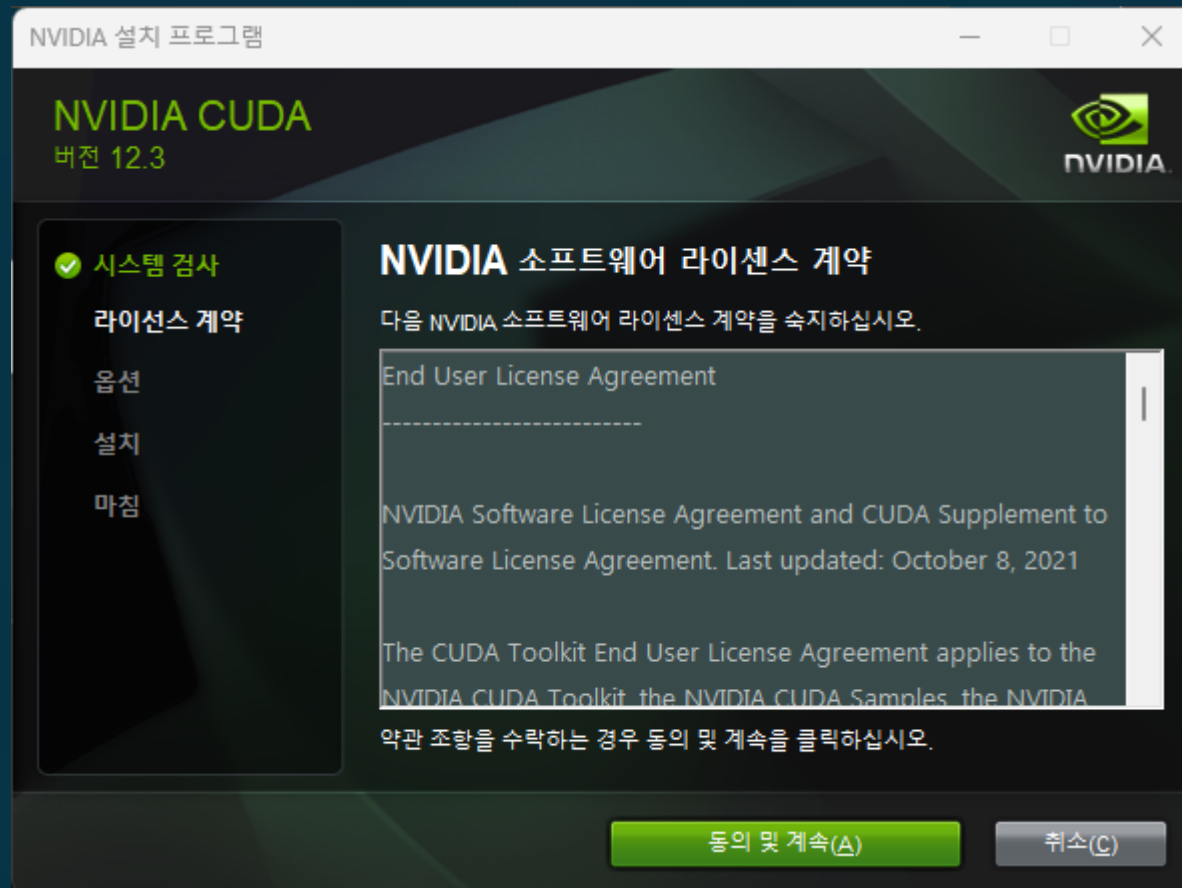
|                  |                   |
|------------------|-------------------|
| > Base Installer | Download (2.9 GB) |
|------------------|-------------------|

Installation Instructions:

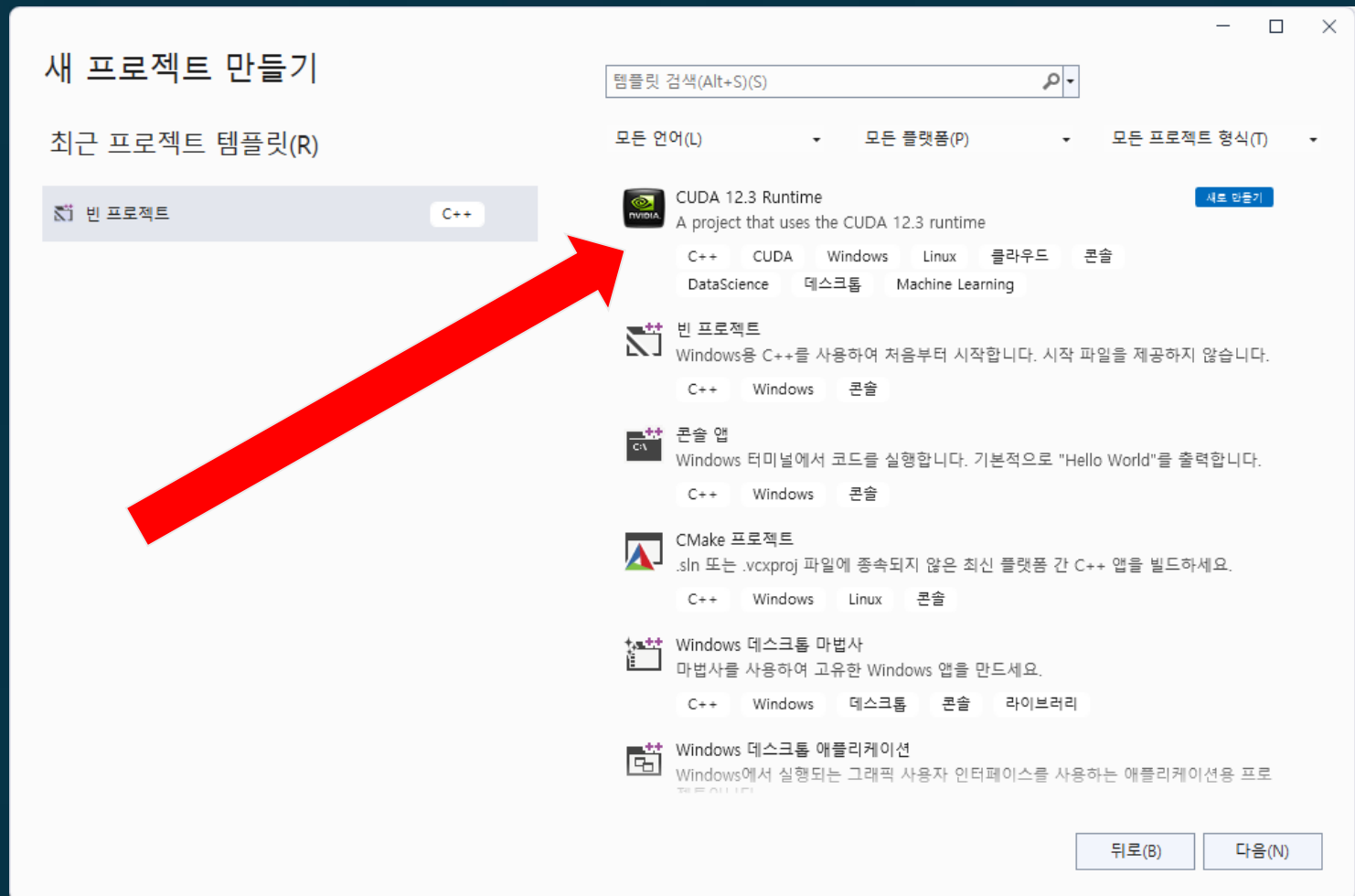
1. Double click cuda\_12.5.0\_555.85\_windows.exe
2. Follow on-screen prompts

# CUDA

## ● 설치



# Visual Studio 연동



# CUDA

## ● 속도비교

```
Microsoft Visual Studio 디버그 콘솔
[Matrix Multiply Using CUDA] - Starting...
GPU Device 0: "Ampere" with compute capability 8.6

MatrixA(960,960), MatrixB(1920,960)
Starting CUDA..
Computing result using CUDA Kernel...
done
Performance= 2589.85 GFlop/s, Time= 1.366 msec, Size= 3538944000 Ops, WorkgroupSize= 1024 threads/block
Checking computed result for correctness: Result = PASS

NOTE: The CUDA Samples are not meant for performance measurements. Results may vary when GPU Boost is enabled.
End CUDA.
Computing result using CPU ...
done
Performance= 0.66 GFlop/s, Time= 5347.000 msec, Size= 3538944000 Ops, Checking computed result for correctness: Result :
PASS

NOTE: The CUDA Samples are not meant for performance measurements. Results may vary when GPU Boost is enabled.

C:\ProgramData\NVIDIA Corporation\CUDA Samples\v11.5\bin\win64\Release\matrixMul.exe(프로세스 29744개)이(가) 종료되었습니다(코드: 0개).
디버깅이 중지될 때 콘솔을 자동으로 닫으려면 [도구] -> [옵션] -> [디버깅] > [디버깅이 중지되면 자동으로 콘솔 닫기]를 사
하도록 설정합니다.
이 창을 닫으려면 아무 키나 누르세요...
```

# OpenCL

- 다운로드
  - <http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/>
- 설치
  - Window 설치파일을 다운 받아서 실행
- 실행
  - Sample\Wcl\W1.x\HelloWorld
- 또는 Nuget

# GPGPU

- 장점

- CPU의 몇 십 배의 속도가 가능 (백배 이상도...)
  - 계산량이 많을 경우

- 단점

- 낮은 I/O 및 직렬 계산 속도
  - CPU와 GPU 사이의 병목 현상
- 적은 메모리 (그래픽 카드의 VRAM)
  - HAS(Hybrid System Architecture)로 극복 (CPU 내장 GPU)



# 목표 및 소개

- 차례
  - OpenMP
  - Intel Thread Building Block
  - CUDA
  - Transactional Memory
  - 새로운 언어.

# 병렬 하드웨어

- Hyper Thread

- Out of Order 를 사용해도 비는 파이프라인을 메꾸기 위해서 도입
- 2개의 코어가 캐시와 ALU를 공유
  - 진실 : 코어에 명령어 Fetch Unit만 하나 추가.
  - 그냥 마케팅 용.
- 벤치마크 프로그램에 따라 성능차이가 큼. 게임은 보통 성능저하...
  - 퇴출

- Big-Little

- Performance-Economic 두종류의 코어
- Bottle Neck Thread가 E Core에서 스케줄링 되는 경우 재앙이 닥침

- Transactional Memory

# Transactional Memory

- 지금까지

- 멀티 스레드용 자료구조를 구현하였다.
  - 리스트, 큐, 스택, 스킵리스트
- 여러 동기화 도구를 사용하여 구현했다.
  - Locking, Spinning, CompareAndSet
  - 각각 단점들을 갖고 있다.
- 크게 **블럭킹**과 **논블럭킹** 방식이 있다.

# 지금 까지...

## ● 블럭킹 (Locking)

- 직관적이다. 의도대로 잘 동작한다.
- 문제 #1
  - 병렬성이 없다. => 성능개선의 여지가 없다.
  - 의도하지 않은 멈춤현상을 야기한다.
    - 우선순위 역전 (priority inversion)
      - 높은 우선순위의 스레드가 잠금이 없어서 실행되지 못함
    - 호위현상 (convoying)
      - 잠금을 잡은 스레드가 실행을 멈춘 동안에는 모든 잠금을 원하는 스레드가 대기해야 한다.

# 지금 까지...

## ● 블럭킹 (Locking)

### – 문제 #2

- 프로그래밍을 주의 깊게 하지 않으면 교착상태(deadlock)에 빠진다.
  - 여러 객체를 동시에 잠가야 할 때 문제가 생긴다.
  - 이전의 해결법 : 능숙한 프로그래머를 고용한다.
    - 멀티 프로세서프로그래밍이 희귀했을 때나 가능
  - 해결법 : 객체간의 순서를 정한다
    - 객체가 동적으로 생성되면?

# 지금 까지...

- 현실 : 잠금에 의존하는 거대한 시스템
  - 성능 때문에 복수의 Lock을 사용
  - 주석에 의존
  - 재앙!

```
/*  
 * When a locked buffer is visible to the I/O layer BH_Laundry  
 * is set. This means before unlocking we must clear BH_Laundry,  
 * mb() on alpha and then clear BH_Lock, so no reader can see  
 * BH_Laundry set on an unlocked buffer and then risk to deadlock.  
 */
```

Figure 18.1 Synchronization by convention: a typical comment from the Linux kernel.

# 지금까지...

## ● Non-Blocking 알고리즘

- HW도움으로 Wait Free하게 수행되는 CAS연산을 사용
- Lock으로 인한 멈춤 현상을 회피할 수 있다.
- 문제
  - 이러한 원자적인 연산으로 알고리즘이나 자료구조를 설계하는 것은 **매우 어려운 일**이다.
  - 프로세서가 많아질수록 CAS연산의 부하가 커진다.

# 지금까지...

- CAS의 근본적인 문제
  - 연산의 단위가 Word이다.
  - 여러 개의 Word의 변경을 원자적으로 할 수 있으면 알고리즘의 구현이 훨씬 쉬워진다.
- 예)
  - `multiCAS({addr1, addr2}, {v1, v2}, {new_v1, new_v2});`



# 지금까지...

```
void enq(int x) {
    Node *e = new Node(x);
    while (true) {
        Node *last = tail;
        Node *next = last->next;
        if (last != tail) continue;
        if (NULL == next) {
            if (NULL == CAS(&(last->next), NULL, e)) {
                CAS(&tail, last, e);
                return;
            }
        } else CAS(&tail, last, next);
    }
}
```



```
void enq(int x) {
    Node *e = new Node(x);
    while (true) {
        Node *last = tail;
        Node *next = last->next;
        if (last != tail) continue;
        if (OK == MultiCAS( {last->next, tail}, {next, last}, {e, e}));
            return;
    }
}
```

# 지금까지...

- multiCAS?
  - 그런 것은 존재하지 않는다.
- singleCAS를 사용한 구현
  - 여러 단계의 singleCAS를 사용해서 구현해야 할 필요가 생기고
  - 연속된 singleCAS실행 사이에 다른 스레드에서 어떠한 행동을 하더라도 제대로 동작하게 만드는 것은 힘들다.
    - 다른 스레드의 행동을 제한하면 Lock-free가 깨진다.

# 지금까지...

- Lock-free algorithm의 한계들
  - Lock-Free 구현은 확장성이 떨어진다.
    - Queue에 Copy()메소드 추가
    - Set에 Clear()메소드 추가
    - 기존의 method들을 전부 수정하여야 한다.
  - 자료구조의 합성
    - 복수개의 method호출의 atomic화는 어렵다.
      - 예) `If_queue.deq(); If_queue.enq(b); => If_queue.atomic_replace(b);`
    - 다른 자료구조의 method들의 연속동작을 atomic하게 구현하는 것은 더 어렵다.
      - 예) `A.enq(B.deq()); A.enq(B.deq())`
      - `=> atomic_transfer2(A, B);`

# 지금까지...

- 자료구조의 정확성
  - Lock-Free 알고리즘의 정확성을 증명하는 것은 매우 어려운 일이다.
    - memory ordering 최적화 까지 겹치면 정말 어렵다.

# 지금까지...

## ● 요약

- 시스템이 커지면 잠금의 효과적인 관리가 어렵다.
- Non-Blocking 알고리즘의 경우 CAS가 워드 단위밖에 되지 않으므로 알고리즘이 복잡해진다.
  - 확장성, 합성, 정확성(생산성)

# 트랜잭션

- 트랜잭션(transaction)
  - 지금까지의 단점들을 보완하기 위하여 고안된 새로운 프로그래밍 모델
  - 하나의 스레드가 실행하는 프로그램 구간 => 트랜잭션으로 정의
  - 정의 : 각각의 트랜잭션은 Atomic하다.
    - 한번에 하나씩 실행된 것처럼 보여야 한다.
    - 교착상태를 발생시키지 않는다.
  - DB의 트랜잭션 개념과 같음
  - 하나의 메모리 연산 아닌, 프로그램 구간을 Atomic하게 변경

# 트랜잭션

- 트랜잭션의 동작
  - <투기적(speculative) 실행>
    - “안되면 말고 ”
  - 하나의 트랜잭션에 속하는 모든 메모리 연산은 임시적(tentative)으로 실행
  - 트랜잭션의 실행이 끝난 후 동기화 충돌 검사
  - 충돌이 없으면 임시적 실행을 영구화 한다.
    - Commit
  - 충돌이 있었으면 모든 실행을 무효화 한다.
    - Abort

# 트랜잭션

- (옵션) 트랜잭션(transaction)의 중첩
  - 트랜잭션안에 트랜잭션
  - Child 트랜잭션의 Abort가 Parent를 Abort시키지 않음



# 트랜잭션

- 예) 무잠금 무제한 큐의 Enqueue
  - **Atomic**{ } : 하나의 트랜잭션 구간을 정의

```
void enq(int x) {  
    Node *e = new Node(x);  
    atomic {  
        tail->next = e;  
        tail = e;  
    }  
}
```

# 트랜잭션

- 예) 원소 두개를 원자적으로 push

```
atomic {  
    q0->push(a);  
    q0->push(b);  
}
```

- 예) 큐에서 두개 원소의 원자적인 이동

```
atomic {  
    q1->enq(q0->deq());  
    q1->enq(q0->deq());  
}
```

# 트랜잭션 메모리의 구현

---

- 하드웨어 트랜잭션 메모리와 소프트웨어 트랜잭션 메모리가 있다.

# 트랜잭션 메모리의 구현

- 소프트웨어 트랜잭션 메모리
  - [http://en.wikipedia.org/wiki/Software\\_transactional\\_memory](http://en.wikipedia.org/wiki/Software_transactional_memory)
  - 많은 종류의 STM 구현이 있다.
    - Tboost.STM
    - Intel STM Compiler
    - SXM, MicroSoft C# 에서의 STM

# 트랜잭션 메모리의 구현

- 구현 아이디어

- 모든 공유 메모리접근을 Transactional Memory 모듈을 통해서 수행.
- 성능을 위해 트랜잭션 종료 시 충돌 검사 수행
- 충돌 시 Undo
  - 이를 위해 Undo 및 Redo 리스트 관리
- Transaction의 state : ACTIVE, COMMIT, ABORTED로 관리

# 트랜잭션 메모리의 구현

- STM 구현 이슈

- Zombie 트랜잭션

- 동기화 충돌 이후에도 트랜잭션이 계속 실행될 수 있다.
    - 충돌 결과 원래는 불가능한 값을 갖게 될 수 있다.
      - 무한루프
      - 예외상황 발생
    - 틀린 값을 읽지 않도록 보장해야 한다.

- 성능

- 모든 Read/Write를 API를 통해서 해야 하므로 성능이 좋지 않다.

# 트랜잭션 메모리의 구현

- 하드웨어 트랜잭션 메모리
  - Cache 일관성 프로토콜을 수정해서 구현
  - SUN의 Rock Processor (망했어요...)
  - Intel의 x86 (Haswell 부터, 2013년 6월 발매)
- 구현
  - 캐시의 태그에 transaction bit추가
  - Transaction 메모리 연산은 transaction cache line에 한다.
  - Transaction cache line이 invalidate되면 cache line을 메모리에 쓰지 않고 폐기한다. => CPU의 실행도 abort한다.

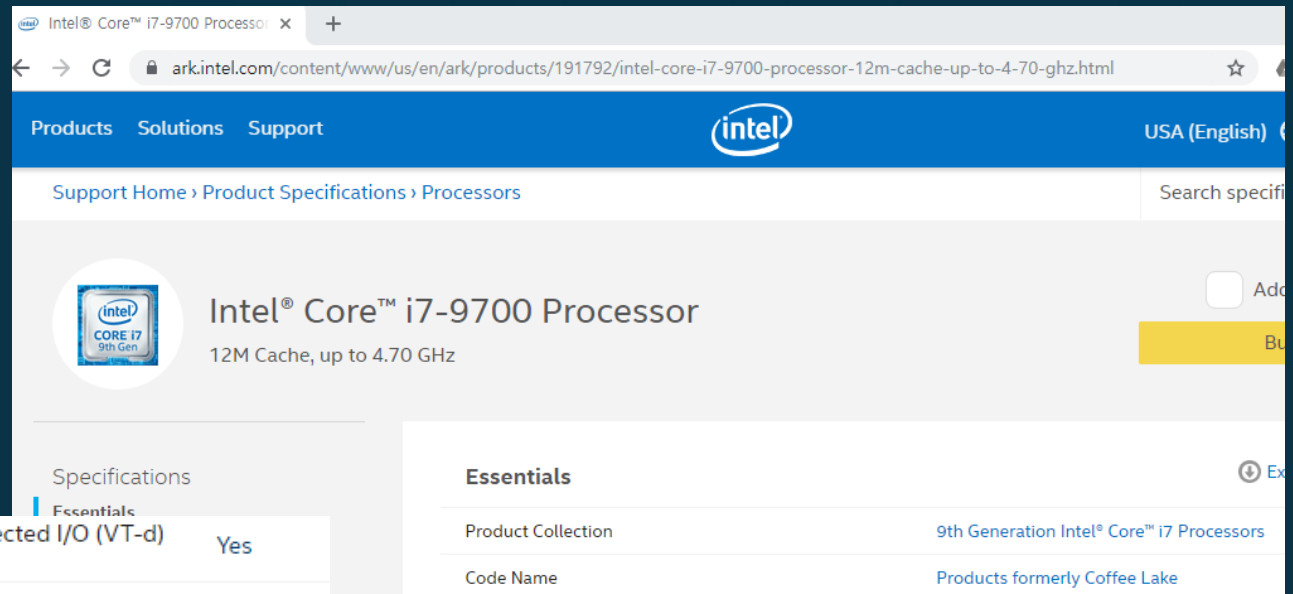
# 현실

- Intel Haswell





# HTM



Intel® Core™ i7-9700 Processor

12M Cache, up to 4.70 GHz

Specifications

Essentials

|                    |   |
|--------------------|---|
| Product Collection | 9th Generation Intel® Core™ i7 Processors |
| Code Name          | Products formerly Coffee Lake             |

|  |             |
|--|-------------|
| Intel® Virtualization Technology for Directed I/O (VT-d) ‡ ? | Yes         |
| Intel® VT-x with Extended Page Tables (EPT) ‡ ?              | Yes         |
| Intel® TSX-NI ?  | Yes         |
| Intel® 64 ‡ ?  | Yes         |
| Instruction Set ?  | 64-bit      |
| Instruction Set Extensions ?                                 | Intel® SSE4 |
| Idle States ?  | Yes         |
| Enhanced Intel SpeedStep® Technology ?                       | Yes         |

# 트랜잭션 메모리의 구현

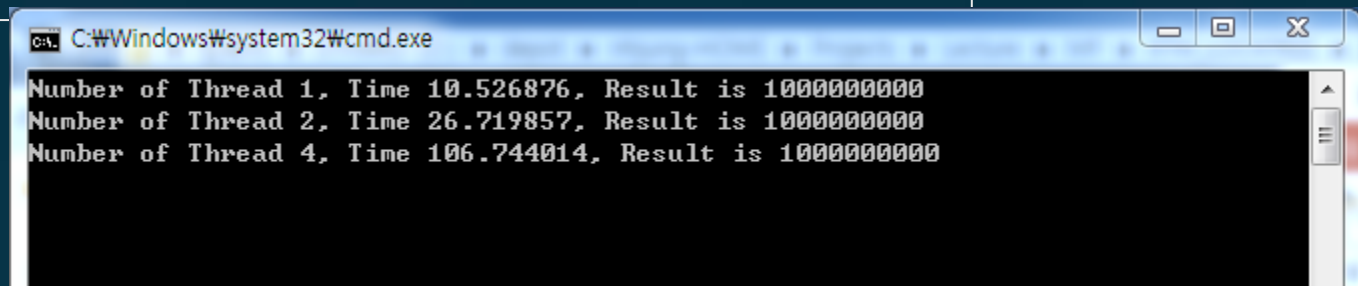
- Haswell의 XTM

- 복수개의 메모리에 대한 Transaction을 허용한다.
  - L1 캐시의 용량 한도 까지
- CPU에서 transaction 실패시의 복구를 제공한다.
  - 메모리와 레지스터의 변경을 모두 Roll-back한다.
- Visual Studio 2012, update 2 부터 지원

# 트랜잭션 메모리의 구현

- 하드웨어 트랜잭션 메모리 예제

```
DWORD WINAPI ThreadFunc(LPVOID lpVoid)
{
    for (int i=0;i<500000000 / num_thread;++i) {
        while (true)
            if (_xbegin() == _XBEGIN_STARTED) break;
        sum = sum + 2;
        _xend();
    }
    return 0;
}
```



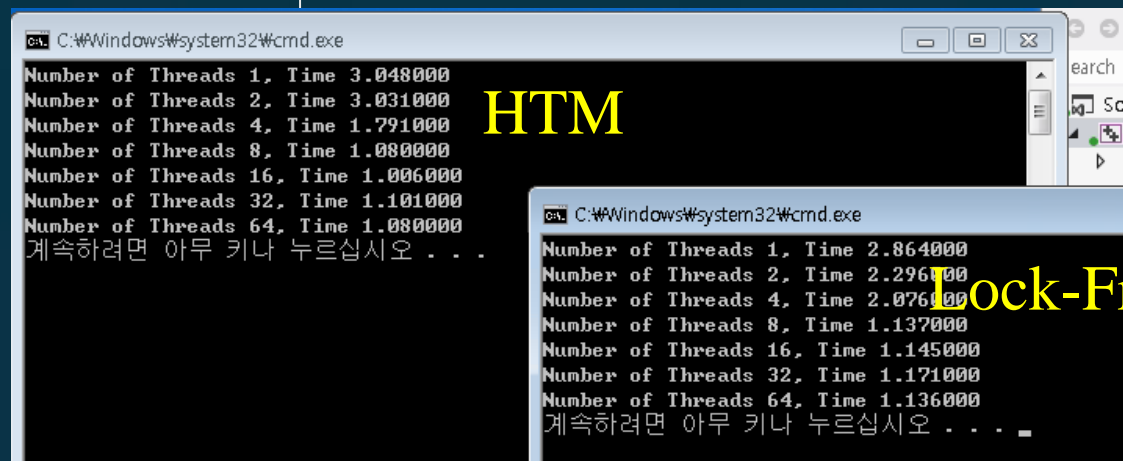
```
C:\Windows\system32\cmd.exe
Number of Thread 1, Time 10.526876, Result is 1000000000
Number of Thread 2, Time 26.719857, Result is 1000000000
Number of Thread 4, Time 106.744014, Result is 1000000000
```

# 트랜잭션 메모리의 구현

## ● 하드웨어 트랜잭션 메모리 예제 (Set의 Add)

```
bool Add(int key)
{
    NODE *pred, *curr;

    NODE *node = new NODE(key);
    while (true) {
        pred = &head;
        curr = pred->next;
        while (curr->key < key) {
            pred = curr;
            curr = curr->next;
        }
        if (_XBEGIN_STARTED != my_xbegin()) continue;
        if (!validate(pred, curr)) {
            _xabort(0);
            continue;
        }
        if (key == curr->key) {
            _xend();
            delete node;
            return false;
        } else {
            node->next = curr;
            pred->next = node;
            _xend();
            return true;
        }
    }
}
```



HTM

Lock-Free

```
C:\Windows\system32\cmd.exe
Number of Threads 1, Time 3.048000
Number of Threads 2, Time 3.031000
Number of Threads 4, Time 1.791000
Number of Threads 8, Time 1.080000
Number of Threads 16, Time 1.006000
Number of Threads 32, Time 1.101000
Number of Threads 64, Time 1.080000
계속하려면 아무 키나 누르십시오 . . .

C:\Windows\system32\cmd.exe
Number of Threads 1, Time 2.864000
Number of Threads 2, Time 2.296000
Number of Threads 4, Time 2.076000
Number of Threads 8, Time 1.137000
Number of Threads 16, Time 1.145000
Number of Threads 32, Time 1.171000
Number of Threads 64, Time 1.136000
계속하려면 아무 키나 누르십시오 . . .
```

# 트랜잭션 메모리의 구현

## ● (SKIP) HTM 실행 레포트

```
#include <ppl.h>
#include <concurrent_unordered_map.h>

using namespace concurrency;

concurrent_unordered_map <unsigned int, unsigned int> xrm_record;

unsigned int my_xbegin()
{
    unsigned int result = _xbegin();
    if (result == _XBEGIN_STARTED) return result;
    atomic_uint *a = reinterpret_cast<atomic_uint *>(&xrm_record[result]);
    (*a)++;
    return result;
}

void report_xrm()
{
    for_each(begin(xrm_record), end(xrm_record), [](const pair<unsigned int, int>& x) {
        cout << "Abort Count: " << x.second << " CODE : " << hex << x.first << dec << "    Abort Type :";
        if (x.first & _XABORT_EXPLICIT) printf("Explicit ");
        if (x.first & _XABORT_RETRY) printf("Retry ");
        if (x.first & _XABORT_CONFLICT) printf("Conflict ");
        if (x.first & _XABORT_CAPACITY) printf("Capacity ");
        if (x.first & _XABORT_DEBUG) printf("Debug ");
        if (x.first & _XABORT_NESTED) printf("Nested ");
        cout << endl;
    });
}
```

# 트랜잭션 메모리의 구현

- (skip) HTM 실행 레포트



```

C:\Windows\system32\cmd.exe
Number of Threads 1, Time 3452
Abort Count: 729 CODE : 0   Abort Type :
Abort Count: 2243 CODE : 8   Abort Type :Capacity
Number of Threads 2, Time 3180
Abort Count: 47388 CODE : 6   Abort Type :Retry Conflict
Abort Count: 546 CODE : 0   Abort Type :
Abort Count: 8399 CODE : 8   Abort Type :Capacity
Abort Count: 83 CODE : 1   Abort Type :Explicit
Number of Threads 4, Time 1961
Abort Count: 106995 CODE : 6   Abort Type :Retry Conflict
Abort Count: 224 CODE : 0   Abort Type :
Abort Count: 12797 CODE : 8   Abort Type :Capacity
Abort Count: 580 CODE : 1   Abort Type :Explicit
Abort Count: 2 CODE : 5   Abort Type :Explicit Conflict
Number of Threads 8, Time 1129
Abort Count: 137801 CODE : 6   Abort Type :Retry Conflict
Abort Count: 100 CODE : 0   Abort Type :
Abort Count: 19097 CODE : 8   Abort Type :Capacity
Abort Count: 1157 CODE : 1   Abort Type :Explicit
Abort Count: 4 CODE : 5   Abort Type :Explicit Conflict
Number of Threads 16, Time 1280
Abort Count: 21340 CODE : 8   Abort Type :Capacity
Abort Count: 142 CODE : 0   Abort Type :
Abort Count: 1236 CODE : 1   Abort Type :Explicit
Abort Count: 153668 CODE : 6   Abort Type :Retry Conflict
Abort Count: 8 CODE : 5   Abort Type :Explicit Conflict
Number of Threads 32, Time 1178
Abort Count: 151564 CODE : 6   Abort Type :Retry Conflict
Abort Count: 20991 CODE : 8   Abort Type :Capacity
Abort Count: 104 CODE : 0   Abort Type :
Abort Count: 1243 CODE : 1   Abort Type :Explicit
Abort Count: 10 CODE : 5   Abort Type :Explicit Conflict
Number of Threads 64, Time 1146
Abort Count: 144490 CODE : 6   Abort Type :Retry Conflict
Abort Count: 120 CODE : 0   Abort Type :
Abort Count: 19821 CODE : 8   Abort Type :Capacity
Abort Count: 1228 CODE : 1   Abort Type :Explicit
Abort Count: 2 CODE : 5   Abort Type :Explicit Conflict
계속하려면 아무 키나 누르십시오 . . .
  
```

# 트랜잭션 메모리

- 장점

- 생산성

- single thread 알고리즘을 그대로 사용 가능 (X)
      - 성능 향상을 위해 접근하는 공유메모리의 개수를 줄여야 함.
      - 성능 향상을 위해 트랜잭션의 실행 시간을 단축시켜야 함
      - 시스템 호출 제거
    - Lock-free 알고리즘 보다는 매우 간단함

- 확장성

- 여러 공유 객체 연결에 제한이 없음

- 정확성

- 제대로 동작하는 알고리즘이라는 것을 검증하기가 쉬움

- 성능

- Lock-free에 근접한 성능
    - 하지만 충돌이 심한 경우 성능 저하

# 트랜잭션 메모리

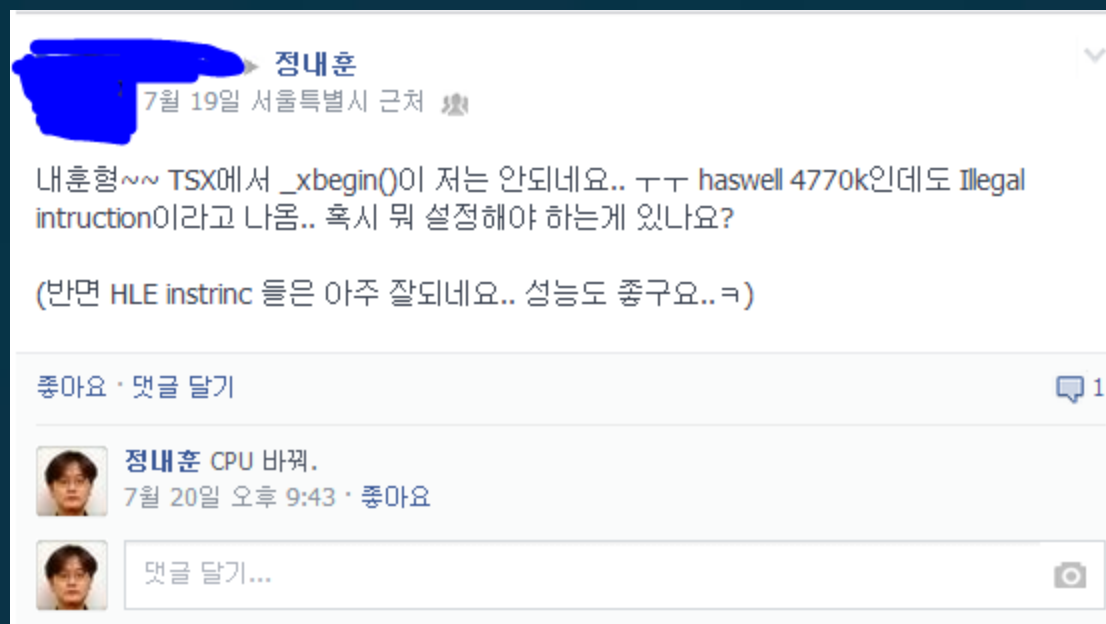
- STM 단점
  - 성능
    - 오버헤드가 커서 Core가 매우 많지 않으면 오히려 성능 저하
- HTM 단점
  - 범용성
    - 일부 CPU에서만 지원 (AMD x, ARM x, 모바일 x...)
  - 제한성
    - HW 용량의 한계로 Algorithm이 제한됨
      - 성공이 불가능할 것을 대비한 대체 실행 프로그램 필요. (아마도 블럭킹)
    - 게으른 동기화 정도의 작성 난이도
    - High Contention 상황에서 lock-free보다 성능 저하.
- 한계
  - Core의 개수가 많아질 경우 성능향상의 한계가 찾아옴 (64개 정도로 예측 by tim sweeny)



# 트랜잭션 메모리의 구현

- Haswell HTM의 한계
  - 모든 알고리즘에 적용 불가능
    - HW 용량 한계 => **알고리즘의 맞춤형 수정 필요.**
    - Nested Transaction 불가능 (가능하지만 무조건 몽땅 Roll-Back)
  - 오버헤드
    - 모든 레지스터 내용 저장 및 Roll-back

그리고...



# 미래

- HTM이 업그레이드 되어서 보급되면 끝인가?
  - 스레드가 많아 질 수록 충돌확률이 올라가 TM의 성능이 떨어진다.
  - 64Core 정도가 한계일 것이라고 예측하고 있다. (2010 GameTech, Tim Sweeny)

# 현실

- ARM
  - Transactional Memory Extension
  - 채용한 CPU가 보이지 않음.
- AMD : Advanced Synchronization Facility (ASF)
  - 결과 없음
- Intel : Haswell 부터 지원, Comet Lake에서 삭제.
  - 지원 중단 : 보안관련 문제들이 발목을 잡음

# 목표 및 소개

- 차례
  - OpenMP
  - Intel Thread Building Block
  - CUDA
  - Transactional Memory
  - 새로운 언어.

# 새로운 언어

- 지금까지
  - 각종 테크닉과 라이브러리들...
  - 성능 향상에 한계가 있다.
- 이유는?
  - C 스타일 언어를 사용하기 때문이다.
    - 쓰레드 사이의 메모리 공유 => Data Race
    - side effect
- 해결책 => 다른 언어 사용
  - 함수형 언어 사용
    - 메모리 공유 없고 side effect 없음
    - 프로그램 자체에 자연스러운 병렬성 내장
  - 최신 언어 사용
    - GO, RUST

# 함수형 언어

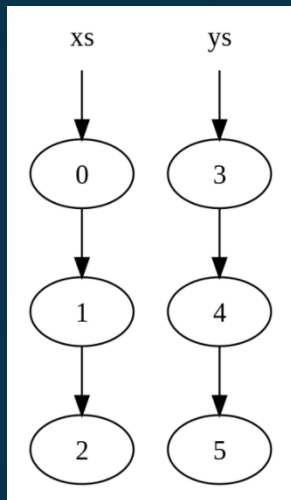
- 함수형 언어의 프로그래밍 스타일
  - 모든 변수가 불변이다. (C++의 `const`)
    - 불변 : 한번 값이 정해지면 바뀌지 않음.
  - 불변 변수(`immutable variable`)는 `data race`를 일으키지 않는다.

함수(입력, 출력) = 함수'(입력, 중간 값) && 함수''(중간 값, 출력)

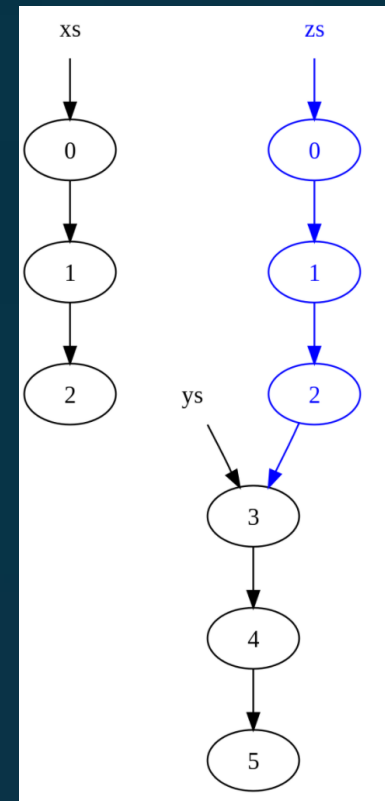
# 함수형 언어

- 불변 변수(immutable variable)
  - Persistent 자료구조라고도 한다.

```
xs = [0, 1, 2]
ys = [3, 4, 5]
```



```
zs = xs ++ ys
```



# 새로운 언어

## ● 하스켈

- 순수 함수형 언어로 1990년에 개발
- 개념은 뛰어나나 난이도로 인해 많이 사용되지 못하고 있음.
- 병렬성
  - 순수 병렬성 (Parallelism) : 언어에 내재된 병렬성 이용, 항상 같은 결과값, data race나 deadlock이 전혀 없음, I/O처리 안됨
  - 동시 실행성 (Concurrency) : I/O 처리를 위해 사용. I/O 실행순서 제어는 프로그래머가 해줘야 함. data race나 deadlock이 가능.



# 새로운 언어

- 함수형 언어의 문제  
— 익히기 어렵다.

```
quickSort :: Ord a => [a] -> [a]
quickSort [] = []
quickSort (x:xs) = quickSort[a | a <- xs, a < x] ++ -- Sort the left part of the list
                  [x] ++ -- Insert pivot between two sorted parts
                  quickSort[b | b <- xs, b >= x] -- Sort the right part of the list
```

```
-- Type annotation (optional)
fib :: Int -> Integer
-- With self-referencing data
fib n = fibs !! n
      where fibs = 0 : scanl (+) 1 fibs
              -- 0,1,1,2,3,5,...
-- Same, coded directly
fib n = fibs !! n
      where fibs = 0 : 1 : next fibs
              next (a : t@(b:_)) = (a+b) : next t
-- Similar idea, using zipWith
fib n = fibs !! n
      where fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
-- Using a generator function
fib n = fibs (0,1) !! n
      where fibs (a,b) = a : fibs (b,a+b)
```

# 새로운 언어

- 하스켈 (from: [Real World Haskell](#) by Bryan O'Sullivan, Don Stewart, and John Goerzen)

```
-- file: ch24/Sorting.hs
sort :: (Ord a) => [a] -> [a]
sort (x:xs) = lesser ++ x:greater
    where lesser  = sort [y | y <- xs, y <  x]
          greater = sort [y | y <- xs, y >= x]
sort _ = []
```

# 새로운 언어

## ● 하스켈 (from: [Real World Haskell](#) by Bryan O'Sullivan, Don Stewart, and John Goerzen) — 병렬 버전

```
-- file: ch24/Sorting.hs
module Sorting where

import Control.Parallel (par, pseq)

parSort :: (Ord a) => [a] -> [a]
parSort (x:xs) = force greater `par` (force lesser `pseq`
                                     (lesser ++ x:greater))
    where lesser = parSort [y | y <- xs, y < x]
          greater = parSort [y | y <- xs, y >= x]
parSort _ = []
```

- ‘par’함수는  $a \rightarrow b \rightarrow b$  형태이고, a함수와 b함수를 병렬로 실행하고 b를 리턴하는 함수
- ‘pseq’함수는 ‘seq’ 함수의 병렬 버전  $a \rightarrow b \rightarrow b$ 하고 b를 리턴

# 새로운 언어

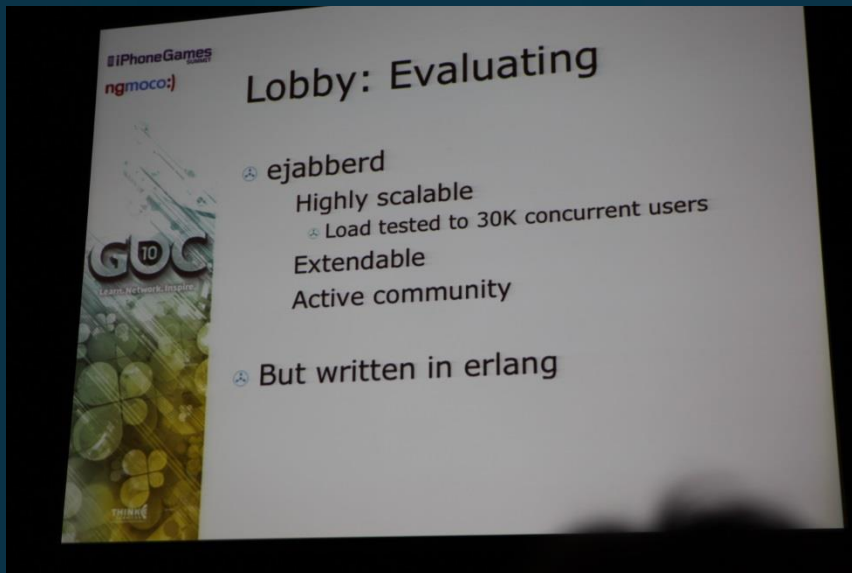
- 하스켈 (from: [Real World Haskell](#) by Bryan O'Sullivan, Don Stewart, and John Goerzen)
  - 최적화 된 병렬버전

```
-- file: ch24/Sorting.hs
parSort2 :: (Ord a) => Int -> [a] -> [a]
parSort2 d list@(x:xs)
  | d <= 0      = sort list
  | otherwise = force greater `par` (force lesser `pseq`
                                     (lesser ++ x:greater))
    where lesser      = parSort2 d' [y | y <- xs, y < x]
          greater     = parSort2 d' [y | y <- xs, y >= x]
          d' = d - 1
parSort2 _ _
```

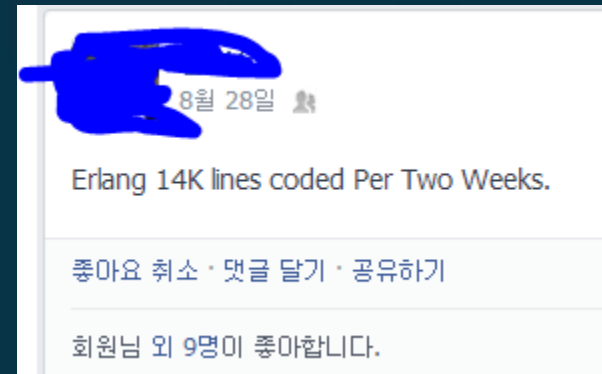
# 새로운 언어

## ● Erlang

- 에릭슨에서 전자 교환기용으로 1982년에 개발
- Scalable한 서버 시스템에 자주 사용되고 있음



2010GDC



# 새로운 언어

- Erlang(사용예) 게임 (from wikipedia)
  - Vendetta Online Naos game server
  - Battlestar Galactica Online game server by Bigpoint
  - Call of Duty server core
  - League of Legends chat system by Riot Games, based on ejabberd

# 새로운 언어

- Erlang(사용예) (from wikipedia)
  - [Amazon.com](#) : to implement [SimpleDB](#), providing database services as a part of the Amazon Web Services offering
  - [Facebook](#) : to power the backend of its chat service, handling more than 100 million active users
  - [WhatsApp](#) : to run messaging servers, achieving up to 2 million connected users per server.
  - [GitHub](#) : used for RPC proxies to ruby processes
  - [CouchDB](#), [Couchbase Server](#), [Mnesia](#), [Riak](#), [SimpleDB](#)

# 새로운 언어

## ● Erlang

Process

Process

- process 단위의 병렬성
  - OS process(x), thread(x), SW context(o)
  - 수천만개의 process 동시 실행 가능
- 2006년부터 SMP(multicore) 지원
- process사이의 동기화
  - shared-nothing asynchronous message passing으로 구현.
  - queue통한 message passing으로 동기화
    - 분산 처리 가능



# 새로운 언어

## ● Erlang : 분산 I/O 구현 예

```
% Create a process and invoke the function
%   web:start_server(Port, MaxConnections)
ServerProcess = spawn(web, start_server, [Port, MaxConnections]),

% Create a remote process and invoke the function
% web:start_server(Port, MaxConnections) on machine RemoteNode
RemoteProcess = spawn(RemoteNode, web, start_server, [Port, MaxConnections]),

% Send a message to ServerProcess (asynchronously).
% The message consists of a tuple with the atom "pause" and the number "10".
ServerProcess ! {pause, 10},

% Receive messages sent to this process
receive
    a_message -> do_something;
    {data, DataContent} -> handle(DataContent);
    {hello, Text} -> io:format("Got hello message: ~s", [Text]);
    {goodbye, Text} -> io:format("Got goodbye message: ~s", [Text])
end.
```

# 새로운 언어

- Elixir

- Erlang에 기반을 둔 함수형 병렬 언어
- Process 지원 (Goroutine과 비슷)
  - 운영체제의 Process와는 이름만 같음
- Send와 Receive를 사용한 동기화.
- 2011년 Jose Valim이 개발
- 달빛 조각사 온라인 게임서버

# 새로운 언어

## ● Elixir 예제

```
defmodule Quicksort do
  def sort([]), do: []
  def sort ([pivot|t]) do
    sort (for x <- t, x < pivot, do: x)
    ++ [pivot] ++
    sort (for x <- t, x >= pivot, do: x)
  end
end
```

```
parent = self()
spawn fn -> send(parent, {:hello, self()}) end
receive do
  {:hello, pid} -> "Got hello from #{inspect pid}"
end
```

# 새로운 언어

## ● 함수형 언어의 문제

### — 생산성

- 전혀 다른 스타일의 프로그래밍 (진입장벽)
- 익숙해지면 오히려 생산성이 올라간다.

### — 성능

- 구조체를 함수에 전달할 때 포인터가 아니라 내용을 전달하는 것과 비슷한 오버헤드.
- 너무 자잘한 병렬화

### — IO 문제

- IO는 순서대로 행해져야 하는데 함수형 언어에서 IO Operation의 순서를 정해주는 것이 비효율적이다.

# 새로운 언어

- Go

- 2009년 구글에서 만든 C와 비슷한 언어.
- 언어에서 멀티쓰레드 지원
  - Goroutine : SW Thread지만 Kernel Thread Pool에서 병렬로 동작.
  - 공유 메모리 모델 : data race존재, mutex 필요
- 멀티쓰레드 문제에 대한 대책 : Channel
  - Goroutine사이의 고속통신, 공유 메모리 대신 사용

# 새로운 언어

## ● Go 예제

```
for {
    conn, err := listener.Accept()
    if err != nil {
        log.Print(err) // e.g., connection aborted
        continue
    }
    go handleConn(conn) // handle connections concurrently
}
```

Goroutine                      가                      .

```
ch := make(chan int, 1)
for i := 0; i < 10; i++ {
    select {
        case x := <-ch:           Datarace                      chanel                      .
            fmt.Println(x) // "0" "2" "4" "6" "8"
        case ch <- i:
    }
}
```

# 목표 및 소개

- 차례

- C++11
- OpenMP
- Intel Thread Building Block
- CUDA
- Transactional Memory
- 새로운 언어.
- LINUX & ARM 실습

# LINUX & ARM

- Linux에서는?

- g++ **-std=c++11 -pthread** hello.cpp

- rand() 함수가 블럭킹!!

- 자체 rand() 함수 필요

```
thread_local unsigned long g_next = 1;

/* RAND_MAX assumed to be 32767 */
int rand_mt(void) {
    g_next = g_next * 1103515245 + 12345;
    return (unsigned) (g_next/65536) % 32768;
}
```



# (SKIP) LINUX & ARM

- ARM에서는?

- 1억 만들기?

- 컴파일러 오류?

- 피터슨 Lock?

- 추가적으로 요구되는 Atomic\_thread\_fence

- Atomic\_compare\_exchange\_weak을 사용한 ABA 방지.

- T\_T TBB, HTM...

# 총정리

- OpenMP, TBB, CUDA, Transactional Memory는 무엇인가?
- 앞으로 우리는 어떠한 프로그래밍을 해야 하는가?
  - 언제까지 non-blocking 알고리즘을 써야 하나?
  - Core의 개수는 점점 늘어날 것인가?
  - C 스타일언어의 미래는?
  - 5년 후는?
  - 10년 후는?

# 정리

- 멀티쓰레드 프로그래밍은 피할 수 없다.
- 일반 프로그래밍에서는 볼 수 없는 많은 골치 아픈 문제가 있다.
- 여러 가지 방법으로 문제를 해결할 수 있다.
  - 성능에 주의 해야 한다.
- 일정 규모 이상의 멀티쓰레드 프로그램에서는 멀티쓰레드용 자료구조가 필요하다.
- Non-Blocking 멀티쓰레드 알고리즘의 사용이 필요하다.
- 자신의 요구에 딱 맞는 Custom한 병렬 알고리즘을 직접 작성해서 사용하는 것이 제일 성능이 좋다. <= 어렵다
- Transactional Memory, Functional Programming도 고려해보자.

# 강의 목적

- 멀티코어에서 성능을 올리려면?
  - non-blocking
- Non-Blocking은 얼마나 성능이 좋은가?
- 왜 어려운가?
- 얼마나 어려운가?
  - 프로그래머 요구사항, 비용
- 할 만한 가치가 있는가?