

C++ 개발자를 위한 병렬 프로그래밍 실전 개발 세미나

구조적 **병렬 패턴**을 통한 **ArBB** 활용법

미래로시스템 고희호

hyungho.ko@gmail.com, <http://hhko.tistory.com>

2011.12.17 (토) 포스코센터 서관 5층

목차

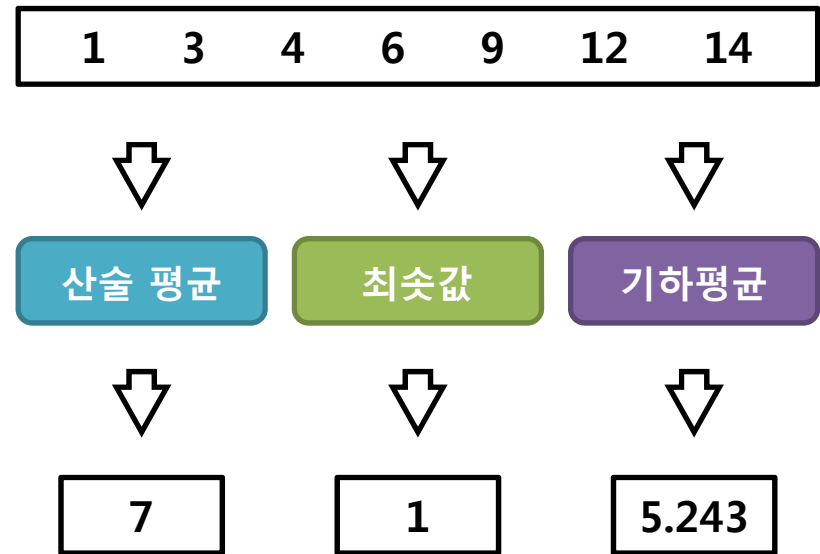
1. 데이터 병렬성
2. ArBB 기초 문법
3. 데이터 의존성
4. 구조적 병렬 패턴
5. Reduction 패턴
6. Map 패턴

1. 데이터 병렬성

병렬성

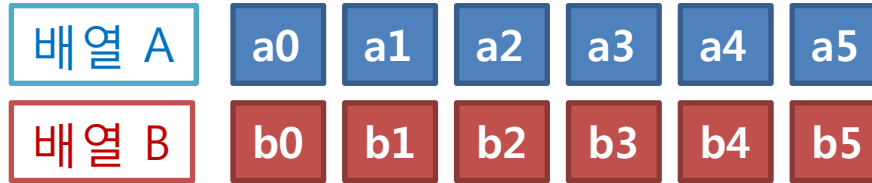


데이터 병렬성
(Data Parallelism)



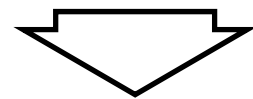
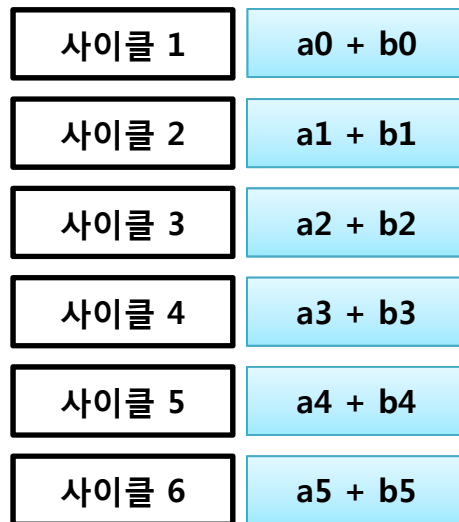
태스크 병렬성
(Task Parallelism)

시스템 자원



배열 A와 B를 더하여라!

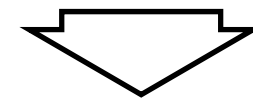
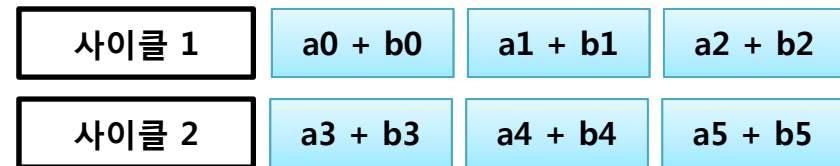
단일 명령으로 단일 데이터 처리



SISD

(Single Instruction stream, Single Data stream)

단일 명령으로 다중 데이터 처리



SIMD

(Single Instruction stream, Multiple Data stream)

Flynn의 병렬 컴퓨터 분류

SISD

Single **I**nstruction stream, Single **D**ata stream

단일 명령으로 단일 데이터 처리

SIMD

Single **I**nstruction stream, Multiple **D**ata stream

단일 명령으로 다중 데이터 처리

MISD

Multiple **I**nstruction stream, Single **D**ata stream

다중 명령으로 단일 데이터 처리

MIMD

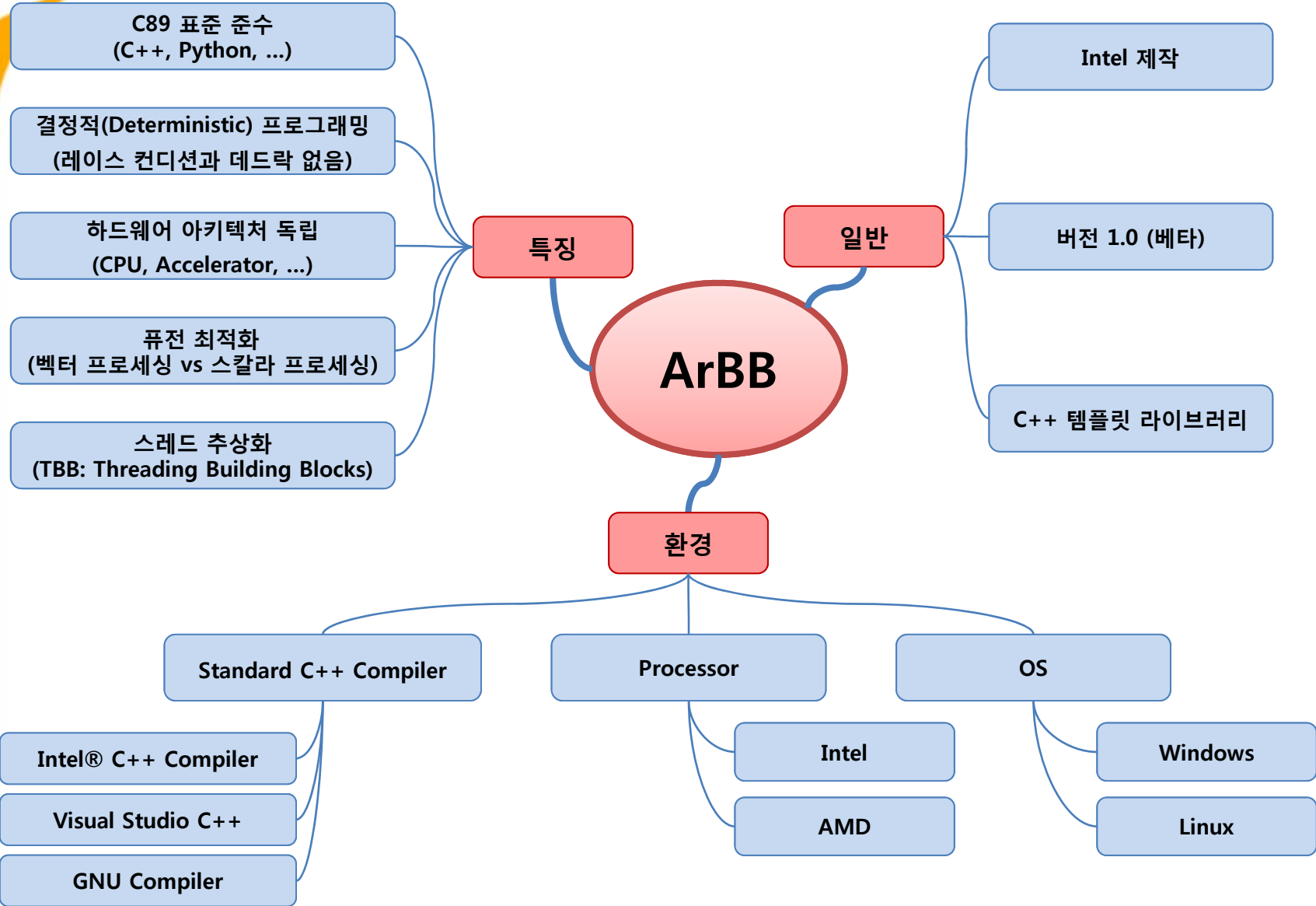
Multiple **I**nstruction stream, Multiple **D**ata stream

다중 명령으로 다중 데이터 처리

SIMD 발전사

| 연도 | CPU | CPU 확장 명령어 셋 | 벡터 처리 능력 (벡터 레지스터 수) |
|------|------------|--------------------------------|-------------------------|
| 1996 | 펜티엄 P55C | MMX Multi Media eXtensions | 64 비트 |
| 1997 | | | |
| 1998 | | | |
| 1999 | 펜티엄 3 카트마이 | SSE Streaming SIMD Extensions | 128 비트 |
| 2000 | | | |
| 2001 | 펜티엄 4 윌라멧 | SSE 2 | |
| 2002 | | | |
| 2003 | | | |
| 2004 | 펜티엄 4 프레스캣 | SSE 3 | |
| 2005 | | | |
| 2006 | 코어2 메룸 | | |
| 2007 | | | |
| 2008 | | | |
| 2009 | 코어 i7 네할렘 | SSE 4 | |
| 2010 | | | |
| 2011 | 샌디 브릿지 | AVX Advanced Vector Extensions | 256 비트 |
| 2012 | | | |
| 2013 | | | |

Array Building Blocks



2. ArBB 기초 문법

스칼라 타입(scalar type)

1. C++과 달리 최적화된 벡터 프로세싱을 위해 **명시적인 데이터 타입 크기 표시**

```
i8, i16, i32, i64, f32, f64 ...
```

2. 내부적으로 템플릿 클래스로 구현

```
template<arbb_scalar_type_t scalar_type>  
class scalar;
```

3. C++ 기본 타입과 비교

| 타입 | 설명 | 유사 C++ 타입 |
|-------------------|-------------------------|-------------------------|
| f32, f64 | 32/64비트 부동소수점 수 | float, double |
| i8, i16, i32, i64 | 8/16/32비트 부호가 있는 정수 | char, short, int |
| u8, u16, u32, u64 | 8/16/32비트 부호가 없는 정수 | unsigned char/short/int |
| boolean | 참/거짓 값 (true 또는 false) | bool |
| usize, isize | 플랫폼에 의존적인 부호가 있거나 없는 정수 | size_t |

스칼라 타입(scalar type)

```
// 8비트 정수  
arbb::i8 value = 10;
```

```
// 32비트 부동소수점 수  
arbb::f32 average = 1.345f;
```

```
// 타입 캐스팅 (arbb::i8 → arbb::f32)  
arbb::i8 value = ...;  
arbb::f32 average = (arbb::f32)value;
```

```
// 타입 캐스팅 (C++ → ArBB)  
float f = 1.5f;  
arbb::f32 average = f;
```

```
// 타입 캐스팅 (ArBB → C++)  
arbb::f32 average = ...;  
float f = (float)average;  
float f = arbb::value(average);
```

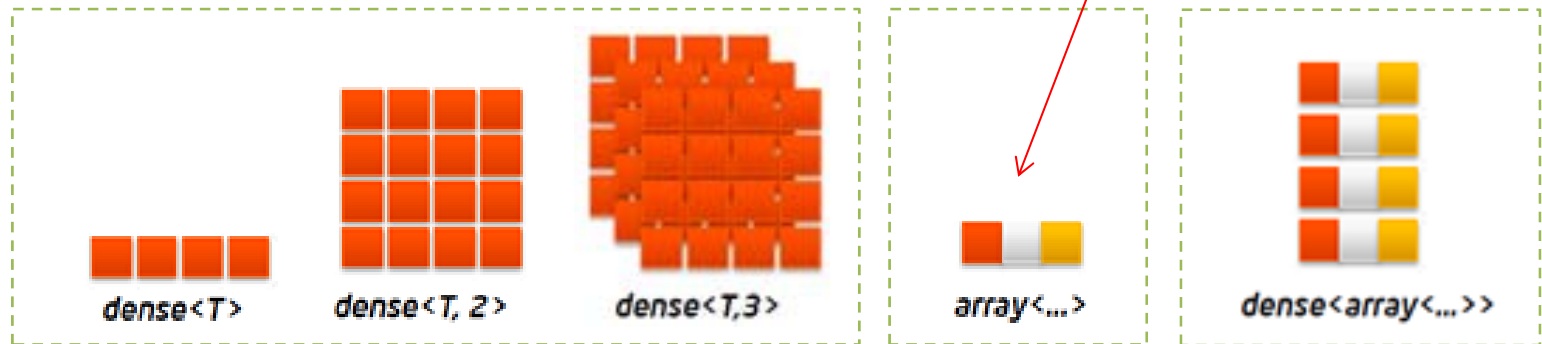
```
template <arbb_scalar_type_t scalar_type>  
typename uncaptured<scalar<scalar_type> >::type value(const scalar<scalar_type> & var)
```

컨테이너 타입(container type)

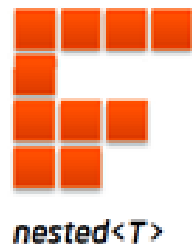
1. 스칼라 타입의 데이터로 구성된 벡터 연산용 컬렉션

2. 종류

① 정형 컨테이너 : `dense`, `array`, 혼용



② 비정형 컨테이너 : `nested`



arbb::dense 컨테이너 타입

1. STL의 vector나 C++의 배열에 해당
2. dense 템플릿 클래스의 원형 : 원소 타입과 차원으로 구성

```
template<typename T, std::size_t D = 1>  
class dense;
```

차수 D는 1에서 3사이의 값 명시하지 않으면 1로 인식됨

제어 흐름

1. 모든 제어 흐름은 **_end** 구문으로 짝을 이루어야 한다(예: **_while**, **_end_while**).
2. ArBB의 제어 흐름 구문은 예약어가 아니라 **매크로**이다.
3. **_for**에 대한 인수들은 세미콜론이 아닌 **콤마**로 구분된다.
4. **do/while** 루프 대신에 **_do/_until** 루프를 사용한다.
5. **switch**와 **goto**는 제공되지 않는다.

```
_for (i = 0, i < max_count, i++)  
{  
    ...  
} _end_for;
```

ArBB의 제어 흐름은 C++과 달리 다음과 같은 규칙을 갖고 있으므로 코드를 작성할 때 유의해야 한다.

_if 매크로

```
#define _if(cond) \
    {{{ \
    ARBB_CPP_NS::detail::if_stack().push(true); \
    ARBB_CPP_NS::detail::if_stack().top().begin(cond); \
    if (ARBB_CPP_NS::detail::capturing() || \
        ARBB_CPP_NS::detail::if_stack().top().if_value()) {{
```

사용자 정의 함수

1. 사용자 정의 함수 호출

함수 포인터

```
call(my_function)(arg1, arg2);
```

새로 캡처 또는 이전에 캡처된 클로저를 반환 반환된 클로저에 대입될 인수들

- ① 호출할 함수를 **클로저(closure)**에 **캡처(capture)**한다.
- ② 두 번째 이후 호출부터는 **이미 캡처된 클로저가 반환**된다.

2. 사용자 정의 함수

- ① **반환 타입**은 반드시 **void**이어야 한다.
- ② 함수의 매개변수 개수는 반드시 **35개 이하**여야 한다.
- ③ 매개변수 타입은 반드시 **ArBB 타입**(스칼라 및 컨테이너), **사용자 정의 ArBB 타입** 또는 이러한 타입들에 대한 참조자이어야 한다.

3. 데이터 의존성

데이터 의존성

데이터 읽기/쓰기 순서에 따른 두 명령어 사이에서 발생한다.

```
x = y + 1; // write  
z = x * 2; // read
```

RAW(Read-After-Write) 의존성

```
z = x * 2; // read  
x = y + 1; // write
```

WAR(Write-After-Read) 의존성

```
x = z * 2; // write  
x = y + 1; // write
```

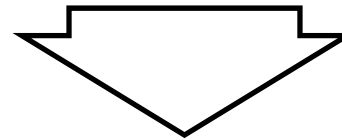
WAW(Write-After-Write) 의존성

WAR 의존성 제거하기

RAW(Read-After-Write) 의존성

$z = x * 2;$
 $x = y + 1;$
 $a = x / 2;$

WAR(Write-After-Read) 의존성

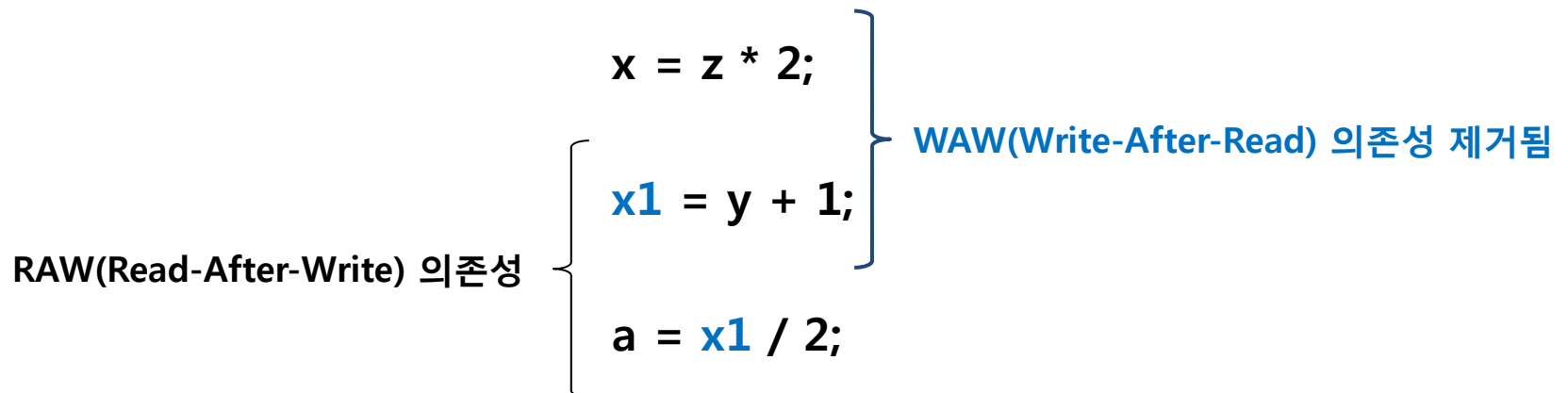
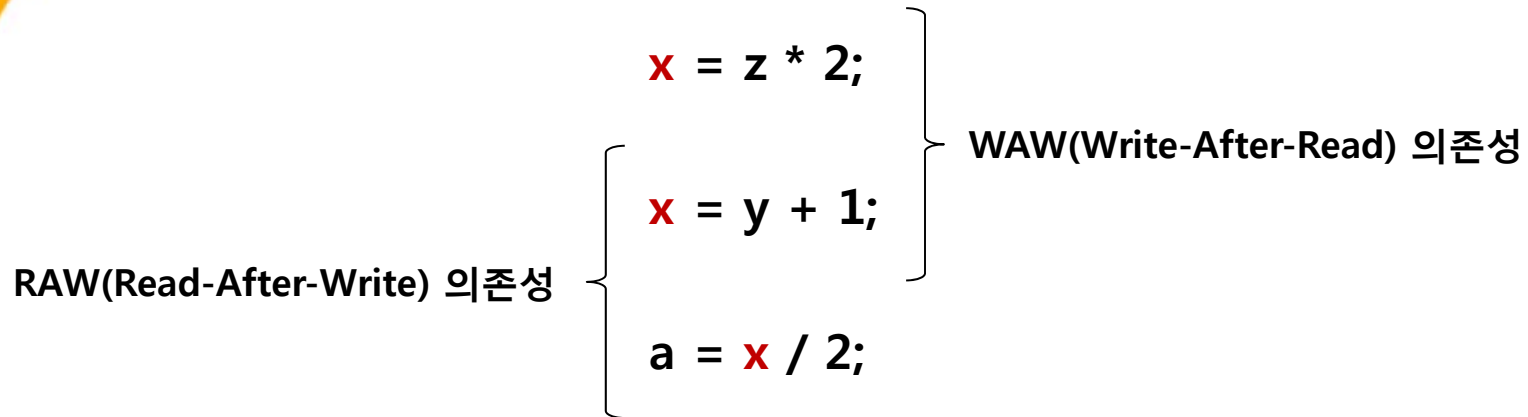


RAW(Read-After-Write) 의존성

$z = x * 2;$
 $x1 = y + 1;$
 $a = x1 / 2;$

WAR(Write-After-Read) 의존성 제거됨

WAW 의존성 제거하기



진짜/가짜 의존성

RAW(Read-After-Write)
의존성



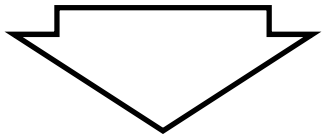
변수 이름 바꾸기



데이터 의존성 제거 안됨

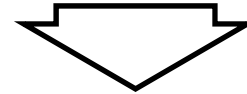


명령어 처리 순서를 변경할 수 없다.

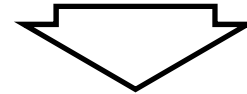


진짜 의존성
(true dependency)

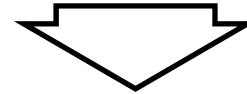
WAR(Write-After-Read) WAW(Write-After-Write)
의존성 의존성



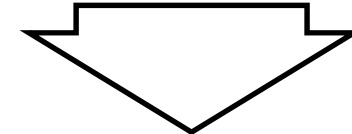
변수 이름 바꾸기



데이터 의존성 제거됨



명령어 처리 순서를 변경할 수 있다.

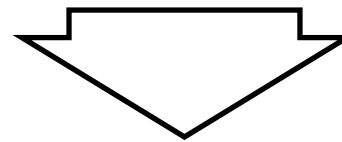


가짜 의존성
(false dependency)

루프 전이 의존성

데이터 의존성이 루프 순환 사이에서 일어날 때

```
for (int index = 1; index < N; index++)  
{  
    A[index] = A[index - 1] + 1;  
}
```



RAW(Read-After-Write) 의존성

index = 1일 때: $A[1] = A[0] + 1;$

index = 2일 때: $A[2] = A[1] + 1;$

index = 3일 때: $A[3] = A[2] + 1;$

RAW(Read-After-Write) 의존성

4. 구조적 병렬 패턴

구조적 직렬 패턴

다음 패턴들은 직렬 연산을 위한 “구조적 프로그래밍”의 기반이 된다.

- 시퀀스
- 선택
- 반복
- 중첩
- 기능
- 재귀
- 랜덤 읽기
- 랜덤 쓰기
- 스택 할당
- 힙 할당
- 객체
- 클로저

이 패턴을 사용하면 “goto”를 거의 다 제거할 수 있어서 소프트웨어의 유지보수성이 향상된다.

구조적 병렬 패턴

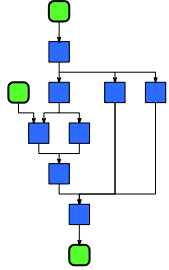
다음 병렬 패턴들은 "구조적 병렬 프로그래밍"에 사용될 수 있다.

- 슈퍼스칼라 시퀀스
- 슈퍼스칼라 선택
- 맵
- 재발생
- 스캔
- 리덕션
- 패킹/확장
- 포크/조인
- 파이프라인
- 파티션
- 분할
- 스텐실
- 검색/매칭
- 수집
- 병합 스캐터링
- 우선순위 스캐터링
- *순열 스캐터링
- !원자적 스캐터링

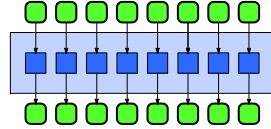
이 패턴을 사용하면 스레드와 벡터 고유 함수를 거의 제거할 수 있어서 소프트웨어의 유지보수성이 향상된다.

구조적 병렬 패턴

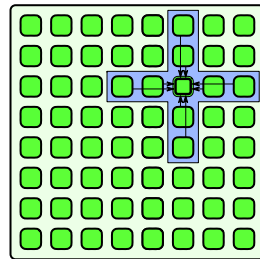
Superscalar sequence



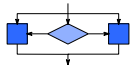
Map



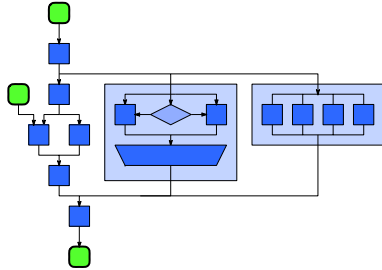
Stencil



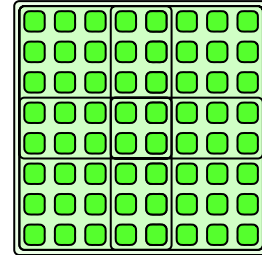
Speculative selection



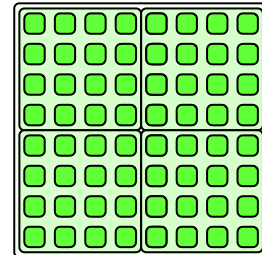
Nesting



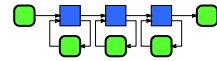
Geometric decomposition



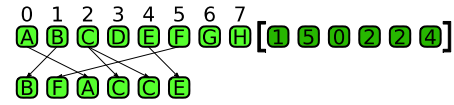
Partition



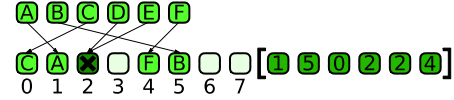
Pipeline



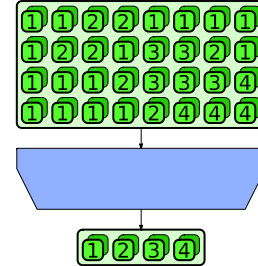
Gather



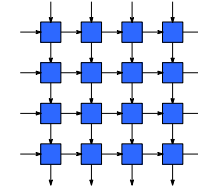
Scatter



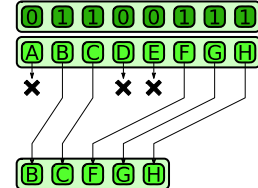
Category Reduction



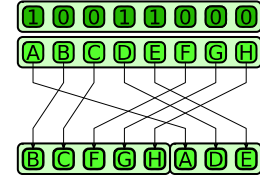
Recurrence



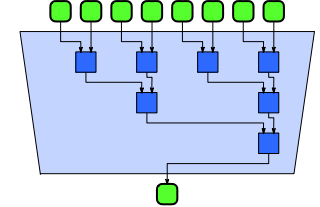
Pack



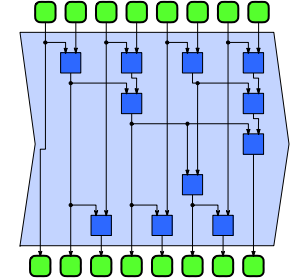
Split



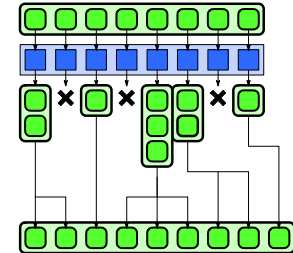
Reduction



Scan

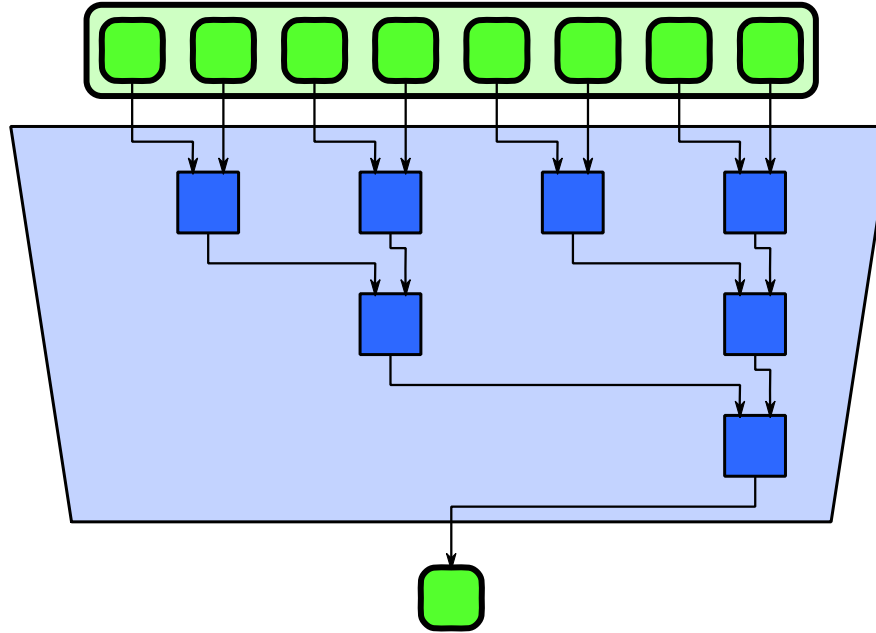


Expand



5. Reduction 패턴

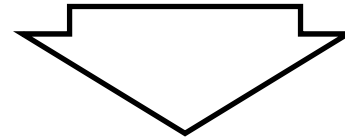
Reduction 패턴



- 리덕션은 결합 연산자를 사용하여 컬렉션에 있는 모든 원소를 하나의 원소로 통합한다.
- 예를 들어, 배열의 합계 또는 최대값을 구하는데 리덕션 연산을 사용할 수 있다.

Reduction 패턴 vs. 데이터 의존성

```
int sum = 0;
for (int index = 1; index < N; index++)
{
    sum += index;           // RAW(Read-After-Write) 의존성
}
```



```
int partial_sums[2] = {0, 0};
```

동시 수행

```
for (int index = 1; index < N/2; index++)    for (int index = N/2; index < N; index++)
{
    partial_sums[0] += index;                {
                                             partial_sums[1] += index;
    }
}
```

```
int sum = partial_sums[0] + partial_sums[1];
```

∴ RAW 의존성이 존재하지만, 그 **중간 연산 순서**는 바뀌어도 **상관없다**.
최종 결과 값만이 중요하다.

Reduction 함수

| 연산 | Reduction | Full Reduction |
|--------------|--------------|----------------|
| 덧셈 | add_reduce() | sum() |
| 곱셈 | mul_reduce() | |
| 최솟값 | min_reduce() | |
| 최댓값 | max_reduce() | |
| 논리곱(and) | and_reduce() | all() |
| 논리합(or) | or_reduce() | any() |
| 배타적 논리합(xor) | xor_reduce() | |

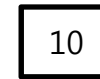
Reduction 예제

1. 1D -> scalar

`dstScalar = add_reduce(srcVec);`

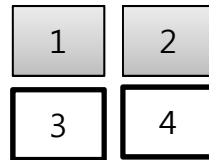


$$1 + 2 + 3 + 4$$



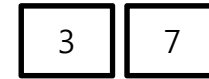
2. 2D -> 1D

`dstVec = add_reduce(srcVec2D);`



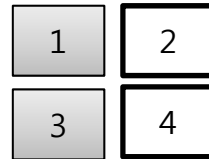
$$1 + 2$$

$$3 + 4$$



3. 2D -> 1D, strided

`dstVec = add_reduce(srcVec2D, level);`



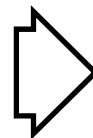
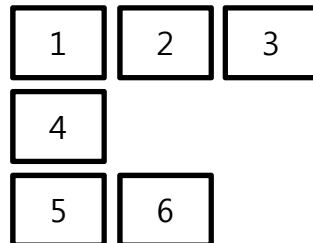
$$1 + 3$$

$$2 + 4$$



4. 2D nested -> 1D

`dstVec = add_reduce(srcNestedVec);`



$$1 + 2 + 3$$

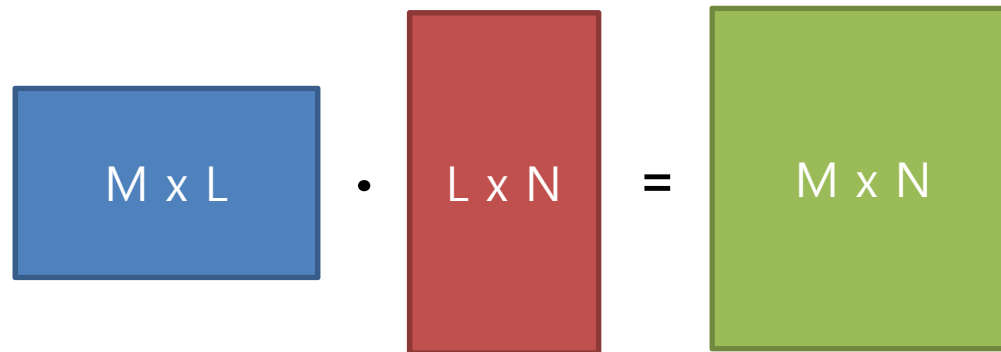
$$4$$

$$5 + 6$$



행렬 곱셈

$$A \cdot B = C$$



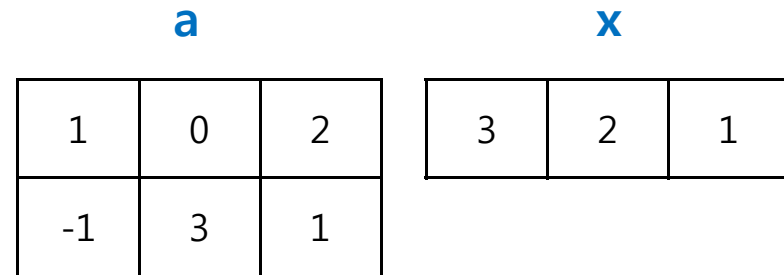
$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} p & q \\ r & s \end{bmatrix} = \begin{bmatrix} (ap + br) & (aq + bs) \\ (cp + dr) & (cq + ds) \end{bmatrix}$$

Reduction 수행

행렬-벡터 곱셈

1. 데이터 할당

$$\begin{pmatrix} 1 & 0 & 2 \\ -1 & 3 & 1 \end{pmatrix} \cdot \begin{pmatrix} 3 \\ 2 \\ 1 \end{pmatrix}$$



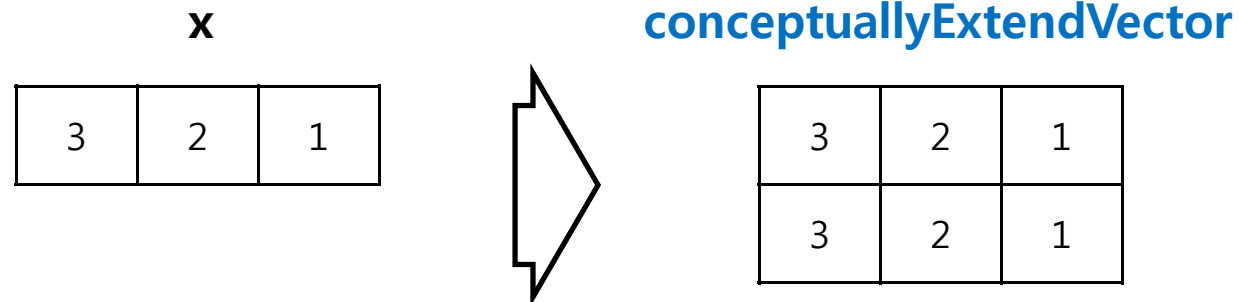
`dense<f32, 2> a;`

`dense<f32> x;`

행렬-벡터 곱셈

2. 차원 확장

`arbb::repeat_row` 함수를 이용하여 두 행렬을 같은 **Dimension**으로 일치시킴!
물리적인 확장이 아닌 **개념적인 확인**임을 명심하자.



```
dense<f32, 2> conceptuallyExtendVector = repeat_row(x, a.num_rows());
```

행렬-벡터 곱셈

3. 컨테이너 곱

행렬 `a`와 `conceptuallyExtendVector`의 곱 계산
(`arbb::dense`의 `operator*` 함수 호출!)

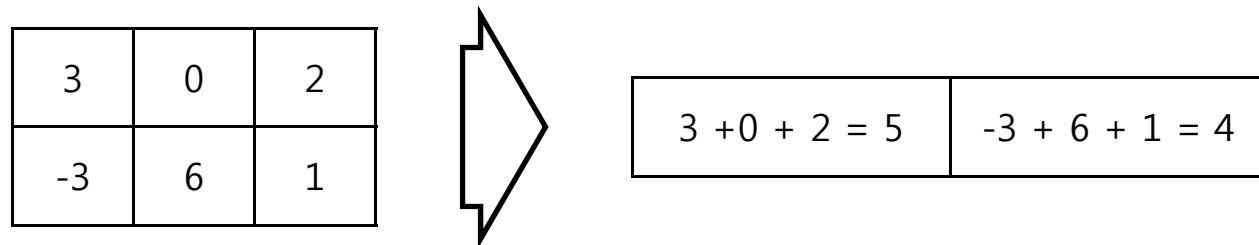
| a | | conceptuallyExtendVector | | mult | | | | | | | | | | | | | | | | | | |
|---|------------------|--------------------------|---|------|---|---|---|--|---|---|---|---|---|---|---|--|------------------|------------------|------------------|--------------------|------------------|------------------|
| <table border="1"><tr><td>1</td><td>0</td><td>2</td></tr><tr><td>-1</td><td>3</td><td>1</td></tr></table> | 1 | 0 | 2 | -1 | 3 | 1 | X | <table border="1"><tr><td>3</td><td>2</td><td>1</td></tr><tr><td>3</td><td>2</td><td>1</td></tr></table> | 3 | 2 | 1 | 3 | 2 | 1 | = | <table border="1"><tr><td>$1 \times 3 = 3$</td><td>$0 \times 2 = 0$</td><td>$2 \times 1 = 2$</td></tr><tr><td>$-1 \times 3 = -3$</td><td>$3 \times 2 = 6$</td><td>$1 \times 1 = 1$</td></tr></table> | $1 \times 3 = 3$ | $0 \times 2 = 0$ | $2 \times 1 = 2$ | $-1 \times 3 = -3$ | $3 \times 2 = 6$ | $1 \times 1 = 1$ |
| 1 | 0 | 2 | | | | | | | | | | | | | | | | | | | | |
| -1 | 3 | 1 | | | | | | | | | | | | | | | | | | | | |
| 3 | 2 | 1 | | | | | | | | | | | | | | | | | | | | |
| 3 | 2 | 1 | | | | | | | | | | | | | | | | | | | | |
| $1 \times 3 = 3$ | $0 \times 2 = 0$ | $2 \times 1 = 2$ | | | | | | | | | | | | | | | | | | | | |
| $-1 \times 3 = -3$ | $3 \times 2 = 6$ | $1 \times 1 = 1$ | | | | | | | | | | | | | | | | | | | | |

```
dense<f32, 2>mult = a * conceptuallyExtendVector;
```

행렬-벡터 곱셈

4. Reduction 수행

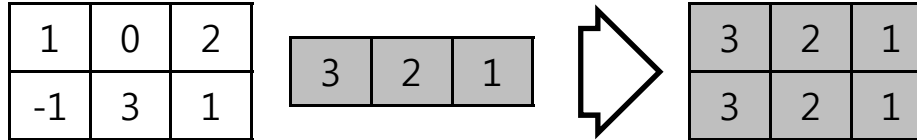
mult에 대해 행 단위로 열 총합을 계산하기 위해 `arbb::add_reduce` 함수 호출!!



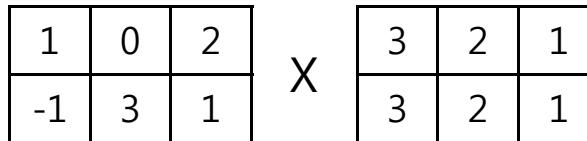
```
b = add_reduce(mult);
```

행렬-벡터 곱셈 정리

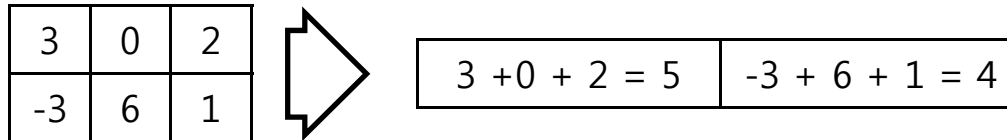
1. repeat_row



2. operator *



3. add_reduce



```
void arbb_matrix_vector(
    const dense<f32, 2>& a,           // 입력: 행렬
    const dense<f32>& x,             // 입력: 벡터
    dense<f32>& b)                   // 출력: 행렬-벡터 곱셈
{
    dense<f32, 2> conceptuallyExtendVector = repeat_row(x, a.num_rows());
    dense<f32, 2> mult = a * conceptuallyExtendVector;
    b = add_reduce(mult);
}
```

행렬 곱셈

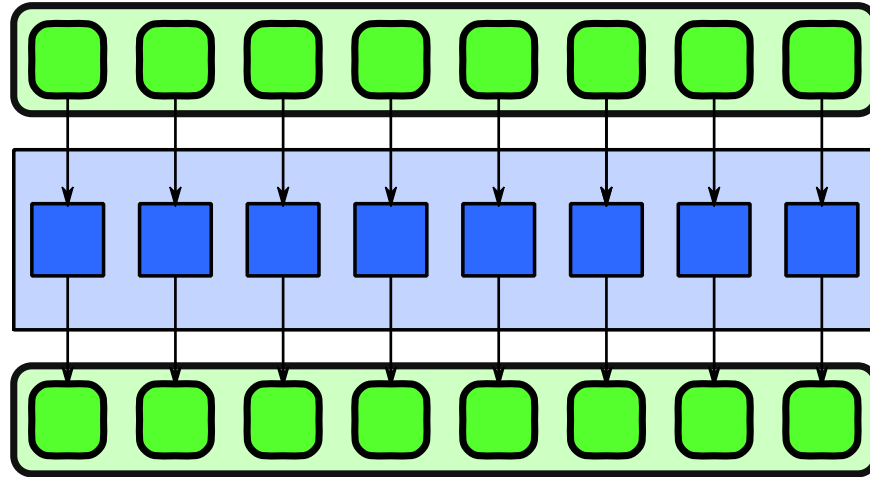
```
void arbb_matrix_multiplication(
    const arbb::dense<arbb::f32, 2>& a, // 입력: M x Q행렬
    const arbb::dense<arbb::f32, 2>& b, // 입력: Q x N행렬
    arbb::dense<arbb::f32, 2>& c)      // 출력: 행렬 곱셈 결과
{
    using namespace arbb;

    size m = a.num_rows();
    size n = b.num_cols();

    _for (size i = 0, i < n, ++i)
    {
        dense<f32, 2> mult = a * repeat_row(b.col(i), m);
        dense<f32> col = add_reduce(mult);
        c = replace_col(c, i, col);
    } _end_for;
}
```

6. Map 패턴

Map 패턴



- 맵은 인덱스 집합의 모든 원소에 대해 함수를 복제한다.
- 인덱스 집합은 추상화될 수 있고 배열의 원소들과 연결될 수 있다.
- 맵은 직렬 프로그램에 있는 특정한 반복자의 사용을 대체한다(**독립적 동작**).

Map 패턴 vs. 데이터 의존성

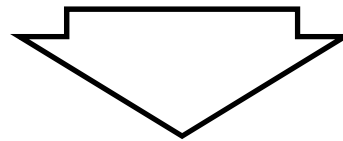
```
for (int i = 0; i < size; i++)  
{  
    a[i] = a[i] + b[i] / c[i] * d[i];  
}
```

$a[0] = a[0] + b[0] / c[0] * d[0];$

$a[1] = a[1] + b[1] / c[1] * d[1];$

$a[2] = a[2] + b[2] / c[2] * d[2];$

RAW/WAR/WAW 의존성이 존재하지 않는다.



모든 연산은 독립적으로 수행 가능하다.

Map 함수

함수 포인터

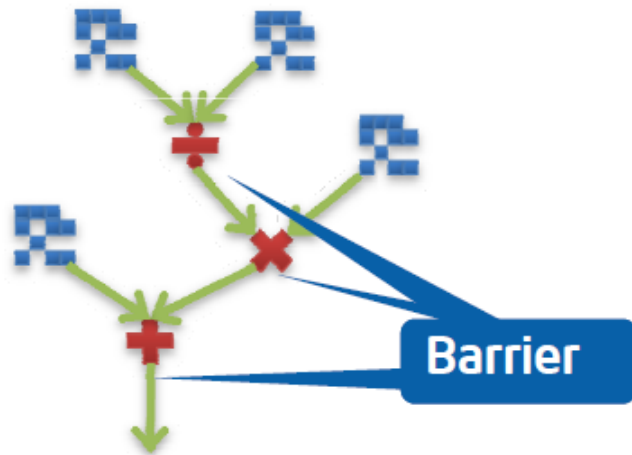
```
map(my_function)(arg1, arg2);
```

새로 캡처 또는 이전에 캡처된 클로저를 반환 반환된 클로저에 대입될 인수들

1. map 함수를 통해 호출되는 사용자 정의 함수는 **동시에 실행**된다.
2. map에 의해 실행되는 각 인스턴스는 **완전히 독립**이다.
3. 하나의 인스턴스에 의한 효과는 모든 인스턴스가 실행되기 전까지 나타나지 않는다.
4. **가장 중요한 사실은 반드시 call 함수 안에서 map 함수가 호출되어야 한다는 점이다.**

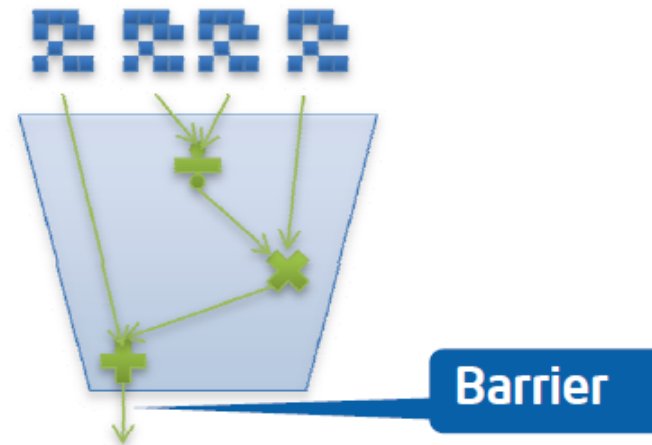
Map 패턴 vs. Barrier

벡터 프로세싱



```
dense<f32> A, B, C, D;  
A = A + B/C * D;
```

스칼라 프로세싱



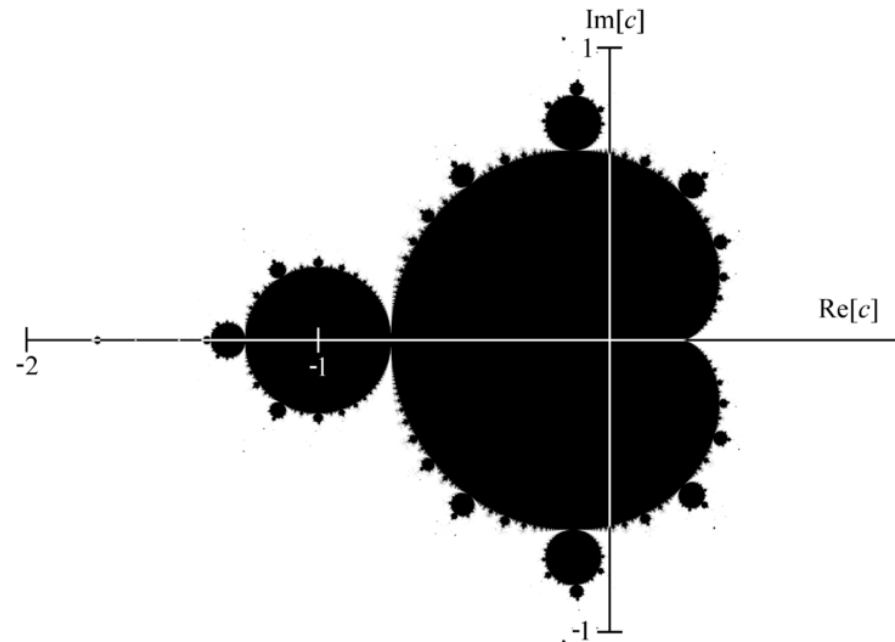
```
void kernel(f32& a, f32 b, f32 c, f32 d) {  
    a = a + (b/c)*d;  
}  
...  
dense<f32> A, B, C, D;  
map(kernel)(A, B, C, D);
```

만델브로 집합

- ✓ **만델브로 집합**은 브누아 만델브로라는 사람이 창시한 프랙탈 기하학에서 나온 개념
- ✓ 다음과 같은 수열이 발산하지 않게 하는 복소수 c 의 집합으로 정의됨.

$$z_0 = 0$$

$$z_{n+1} = z_n^2 + c$$



만델브로 집합

```
void serial_mandel(const std::complex<double>& c,
    unsigned int max_reurrences, unsigned char& output)
{
    unsigned int i = 0;
    std::complex<double> z = c;

    while (i < max_reurrences)    {
        if (z.real() * z.real() + z.imag() * z.imag() > 4.0)
            break;
        z = z * z + c;
        i++;
    }

    output = static_cast<unsigned char>(static_cast<double>(i) / max_reurrences * 255);
}

void serial_mandelbrot(double x0, double y0, double x1, double y1,
    unsigned int width, unsigned int height, unsigned int max_reurrences, unsigned char* output)
{
    double dx = (x1 - x0) / width;
    double dy = (y1 - y0) / height;

    for (unsigned int j = 0; j < height; j++) {
        for (unsigned int i = 0; i < width; i++) {
            unsigned int index = j * width + i;
            double x = x0 + i * dx;
            double y = y0 + j * dy;
            serial_mandel(std::complex<double>(x, y), max_reurrences, output[index]);
        }
    }
}
```

만델브로 집합

```
for (unsigned int j = 0; j < height; j++)           // 루프 1.
{
    for (unsigned int i = 0; i < width; i++)         // 루프 2.
    {
        unsigned int index = j * width + i;
        double x = x0 + i * dx;
        double y = y0 + j * dy;
        serial_mandel(std::complex<double>(x, y), max_reurrences, output[index]);
    }
}
```

serial_mandel 함수에서는 지역 변수만 사용한다.

루프 1과 2에서 루프 전이 의존성(RAW/WAR/WAW)이 존재하지 않는다.

만델브로 집합

```
void serial_mandel(const std::complex<double>& c, unsigned int max_reurrences, unsigned char& output)
{
    unsigned int i = 0;
    std::complex<double> z = c;

    while (i < max_reurrences)    {
        if (z.real() * z.real() + z.imag() * z.imag() > 4.0)
            break;
        z = z * z + c;
        i++;
    }

    output = static_
}
```

Serial

```
void arbb_mandelbrot_map(u32 max_reurrences, f64 x0, f64 y0, f64 dx, f64 dy, u8& output)
{
    u32 i = 0;
    const array<f64, 2> pos = position<2>().as<f64>();

    const std::complex<f64> c(x0 + pos[0] * dx, y0 + pos[1] * dy);
    std::complex<f64> z = c;

    _while (i < max_reurrences)
    {
        _if (norm(z) > 4.0)
        {
            _break;
        } _end_if;

        z = z * z + c;
        ++i;
    } _end_while;

    output = u8(f64(255u * i) / f64(max_reurrences));
}
```

ArBB

만델브로 집합

```
void arbb_mandelbrot(double x0, double y0, double x1, double y1,
    unsigned int max_reurrences, dense<u8, 2>& output)
{
    call(arbb_mandelbrot_call)(x0, y0, x1, y1, max_reurrences, output);
    output.read_only_range();
}

void arbb_mandelbrot_call(f64 x0, f64 y0, f64 x1, f64 y1,
    u32 max_reurrences, dense<u8, 2>& output)
{
    usize width = output.num_cols();
    usize height = output.num_rows();

    f64 dx = (x1 - x0) / f64(width);
    f64 dy = (y1 - y0) / f64(height);
    map(arbb_mandelbrot_map)(max_reurrences, x0, y0, dx, dy, output);
}
```