[KISTI 기술문서] ISBN 978-89-294-1695-9

X86_64 SIMD 프로그래밍 시작하기

2024. 7. 1.

한국과학기술정보연구원
슈퍼컴퓨팅기술개발센터

저자소개

김상완

한국과학기술정보연구원
국가슈퍼컴퓨팅본부 슈퍼컴퓨팅기술개발센터
책임연구원
sangwan@kisti.re.kr

정기문

한국과학기술정보연구원
국가슈퍼컴퓨팅본부 슈퍼컴퓨팅기술개발센터
책임연구원
kmjeong@kisti.re.kr

이 기술보고서는 2024년도 한국과학기술정보연구원(KISTI)의
기본사업으로 수행된 연구입니다.
과제번호: (KISTI) K24-L02-C06-S01
과제명: 초고성능컴퓨팅 공동활용을 위한 통합 환경 개발 및 구축

목 차

1. 개요

- SIMD(Single Instruction Multiple Data, 단일명령 복수자료)는 컴퓨터 아키텍처의 병렬 처리 기법 중 하나로 하나의 명령어를 사용하여 여러 데이터 요소를 동시에 처리할 수 있다. 데이터 병렬성(data parallelism)을 활용하는 것으로 벡터 프로세서와 GPU 등에서 사용됨. 과학계산, 이미지 처리, 신호처리 등 응용분야에서 활용됨.
- SIMD 개념이 최초로 적용된 벡터 컴퓨터는 일리아크(ILLIAC)[1][2]로 알려져 있다. 256개의 64비트 부동형실수 처리장치(FPU)와 4개의 CPU로 구성되도록 설계되었음. 1972년 전체 설계중 일부분만 제작됨. 이후 Cray-1(1976년)[3], CDC STAR-100(1974년)[4] 등의 컴퓨터에 벡터 연산 유닛이 적용됨.
- 다음과 같은 문제 해결을 위해 하드웨어 구조에 대한 이해가 중요함:
  - 메모리 계층 구조, 캐시 일관성, 데이터 병렬성을 고려한 성능 최적화
  - OpenMP, MPI, CUDA, OpenCL 등 병렬프로그래, 쓰레드 관리, 로드 밸런싱
  - 메모리 대역폭, 분산시스템의 통신 대역폭, I/O 병목 문제 해결
  - GPU, 전용가속기, 벡터 명령(SSE, AVX, SIMD) 활용을 통한 성능 향상
- SIMD 프로그래밍을 효과적으로 하기위해서는 다음과 같은 하드웨어의 특징을 이해하는 것이 중요함:
  - 레지스터의 크기 및 레지스터 파일의 구조
  - 명령어 종류 및 기능
  - 메모리 정렬 방식, 로드/스토어 명령어의 특성
  - 파이프라이닝 구조
- 본 문서는 X86시스템에서 SIMD 병렬 프로그래밍을 시작하기위한 기본적으로 요구되는 내용을 실제 예제 코드를 중심으로 설명한다.

1) https://en.wikipedia.org/wiki/ILLIAC_IV
2) https://www.computerhistory.org/revolution/supercomputers/10/160/281
3) https://en.wikipedia.org/wiki/Cray-1
4) https://en.wikipedia.org/wiki/CDC_STAR-100

2. 시스템 정보 및 성능 측정 방법

2.1 CPUID

- cpuid 명령은 프로세서의 특정 정보를 반환하는데 사용됨. 제조사, 모델, 기능 세트 등을 식별하기 위함. EAX 레지스터에 특정 값을 설정한 후 명령어를 실행하면, EAX, EBX, ECX, EDX 레지스터에 결과 값을 반환.

- 아래와 같이 코드를 작성하면 CPUID를 조회할 수 있다. (인텔 제온 8488C 프로세서 기준)

```
// cpuid.c
#include <stdio.h>
void cpuid(int code1, int code2, unsigned int *a, unsigned int *b, unsigned int *c, unsigned in
t *d) {
  // "=a"(*a), "=b"(*b), "=c"(*c), "=d"(*d)
  // : cpuid 명령어 실행 후, 레지스터 EAX, EBX, ECX, EDX의 값을 각각 *a, *b, *c, *d에 저장
  // "a"(code1), "c"(code2)
  // : cpuid 명령어 실행 전에 EAX와 ECX 레지스터에 각각 code1과 code2 값을 설정
    __asm__ volatile( "cpuid"
                    : "=a"(*a), "=b"(*b), "=c"(*c), "=d"(*d)
                    : "a"(code1), "c"(code2)  );
}
int main() {
    unsigned int eax, ebx, ecx, edx;
    cpuid(0, 0, &eax, &ebx, &ecx, &edx); // EAX=0: 프로세서 제조사 문자열

    char vendor[13];
    ((unsigned int *)vendor)[0] = ebx;
    ((unsigned int *)vendor)[1] = edx;
    ((unsigned int *)vendor)[2] = ecx;
    vendor[12] = '\0';
    printf("Processor Vendor: %s\n", vendor);

    cpuid (1, 0, &eax, &ebx, &ecx, &edx); // EAX=1: 프로세서 버전 및 특징 정보
    printf("Processor Version: %08x\n", eax);
    printf("CPUID(EAX=01H):ECX: %08x  EDX: %08x\n", ecx, edx);

    cpuid (7, 0, &eax, &ebx, &ecx, &edx); // EAX=7, ECX=0: AVX-512 features
    printf("CPUID(EAX=07H):EBX: %08x\n", ebx);
    return 0;
}
```

```
$ gcc -g -o cpuid cpuid.c
$ ./cpuid
Processor Vendor: GenuineIntel
Processor Version: 000806f8
CPUID(EAX=01H):ECX: fffab20b  EDX: 1f8bfbff
CPUID(EAX=07H):EBX: f1bf07ab
```

```
ECX : fffab20b ==>  11111111111110101011001000001011
EDX : 1f8bfbff ==>  00011111100010111111101111111111
EBX : f1bf07ab ==>  11110001101111110000011110101011
$ objdump  -j .text --disassemble=cpuid  cpuid
```

```
cpuid:     file format elf64-x86-64
Disassembly of section .text:
0000000000001169 <cpuid>:
    1169:  f3 0f 1e fa           endbr64
[...]
    1188:       8b 45 f4                mov    -0xc(%rbp),%eax
    118b:       8b 55 f0                mov    -0x10(%rbp),%edx
    118e:       89 d1                   mov    %edx,%ecx
    1190:       0f a2                   cpuid
    1192:       89 de                   mov    %ebx,%esi
    1194:       89 c7                   mov    %eax,%edi
[...]
```

- CPUID 명령의 opcode는 0FA2 임을 확인할 수 있다. SDM문서를 참조.

CPUID—CPU Identification

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F A2	CPUID	ZO	Valid	Valid	Returns processor identification and feature information to the EAX, EBX, ECX, and EDX registers, as determined by input entered in EAX (in some cases, ECX as well).

Table 3-8. Information Returned by CPUID Instruction

Initial EAX Value	Information Provided about the Processor	
	Basic CPUID Information	
0H	EAX	Maximum Input Value for Basic CPUID Information.
	EBX	"Genu"
	ECX	"ntel"
	EDX	"inel"
01H	EAX	Version Information: Type, Family, Model, and Stepping ID (see Figure 3-6).
	EBX	Bits 07 - 00: Brand Index. Bits 15 - 08: CLFLUSH line size (Value * 8 = cache line size in bytes; used also by CLFLUSHOPT). Bits 23 - 16: Maximum number of addressable IDs for logical processors in this physical package*. Bits 31 - 24: Initial APIC ID.
	ECX	Feature Information (see Figure 3-7 and Table 3-10).
	EDX	Feature Information (see Figure 3-8 and Table 3-11).
		NOTES: * The nearest power-of-2 integer that is not smaller than EBX[23:16] is the number of unique initial APIC IDs reserved for addressing different logical processors in a physical package. This field is only valid if CPUID.1.EDX.HTT[bit 28]= 1.

- AVX512 명령어를 지원하는지 여부는 CPUID(EAX=07H,ECX=0) 명령으로 알수 있다.

Table 15-1. 512-bit Instruction Groups in the Intel AVX-512 Family

CPUID Leaf 7 Feature Flag Bit	Feature Flag abbreviation of 512-bit Instruction Group	SW Detection Flow
CPUID.(EAX=07H, ECX=0):EBX[bit 16]	AVX512F (AVX-512 Foundation)	Figure 15-2
CPUID.(EAX=07H, ECX=0):EBX[bit 28]	AVX512CD	Figure 15-4
CPUID.(EAX=07H, ECX=0):EBX[bit 17]	AVX512DQ	Figure 15-4
CPUID.(EAX=07H, ECX=0):EBX[bit 30]	AVX512BW	Figure 15-4

- CPUID 결과값의 의미(인텔 제온 8488C 프로세서 기준). AVX-512 F,DQ,CD,BW 등의 기능을 지원함을 확인.

```
                    31   27   23   19   15   11   7       0  Bit
ECX : fffab20b ==>  1111 1111 1111 1010 1011 0010 0000 1011
                     ||| |||| |||| | |  | ||   |       | ||
          rdrand 30 -+|| |||| |||| | |  | ||   |       | |+-  0 sse3
            f16c 29 --+| |||| |||| | |  | ||   |       | +--  1 pclmulqdq
             avx 28 ---+ |||| |||| | |  | ||   |       +----  3 monitor
          osxave 27 -----+||| |||| | |  | ||   +------------  9 ssse3
           xsave 26 ------+|| |||| | |  | |+----------------- 12 fma
             aes 25 -------+| |||| | |  | +------------------ 13 cmpxchg16b(cx16)
    tsc_deadline 24 --------+ |||| | |  +-------------------- 15 pdcm
          popcnt 23 ----------+||| | +----------------------- 17 pcid
           movbe 22 ----------+|| +------------------------- 19 sse4_1
         x2apic  21 ----------++---------------------------- 20 sse4_2
```

```
                    31   27   23   19   15   11   7       0  Bit
EDX : 1f8bfbff ==>  0001 1111 1000 1011 1111 1011 1111 1111
              ht 28 ---+ |||| |    | || |||| | || |||| |||+-  0 fpu
              ss 27 -----+||| |    | || |||| | || |||| ||+--  1 vme
            sse2 26 ------+|| |    | || |||| | || |||| |+---  2 de
             sse 25 -------+| |    | || |||| | || |||| +----  3 pse
            fxsr 24 --------+ |    | || |||| | || |||+------  4 tsc
             mmx 23 ----------+    | || |||| | || ||+-------  5 msr
         clflush 19 ---------------+ || |||| | || |+--------  6 pae
           pse36 17 ----------------+| |||| | || +----------  7 mce
             pat 16 ------------------+ |||| | |+------------  8 cx8
                                        |||| | +-------------  9 apic
                                        |||| +---------------- 11 sep
                                        |||+----------------- 12 mtrr
                                        ||+------------------ 13 pge
                                        |+------------------- 14 mca
                                        +-------------------- 15 cmov
```

```
                    31   27   23   19   15   11   7    3  0  Bit
EBX : f1bf07ab ==>  1111 0001 1011 1111 0000 0111 1010 1011
                     | |             |+---------------------- 16 avx512f
                     | |             +----------------------- 17 avx512dq
                     | +--------------------------------------- 28 avx512cd
                     +----------------------------------------- 30 avx512bw
```

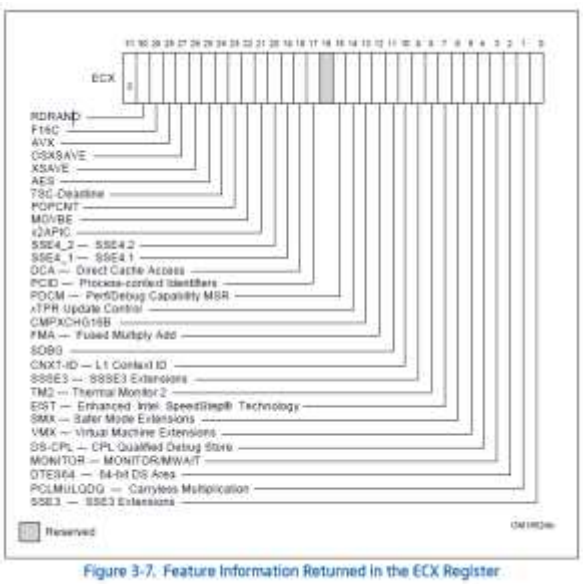


Figure 3-7. Feature Information Returned in the ECX Register

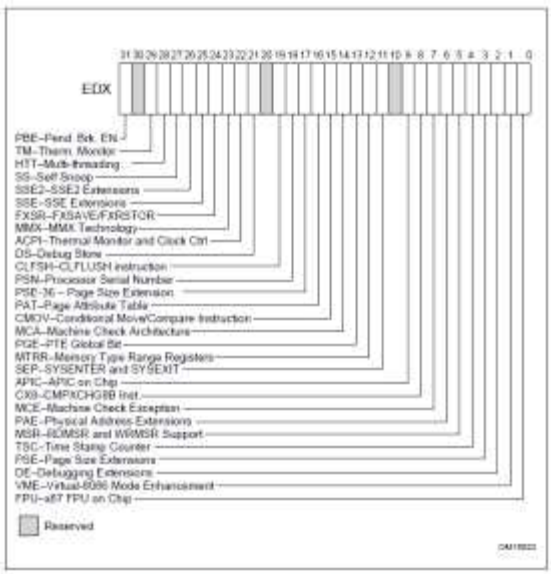


Figure 3-8. Feature Information Returned in the EDX Register

※ cpuid 명령을 이용하면 CPU 정보를 볼수 있음. AMX 관련 명령을 지원하는지 체크하기

```
$ sudo apt install cpuid
$ cpuid
CPU 0:
   vendor_id = "GenuineIntel"
   version information (1/eax):
      processor type  = primary processor (0)
      family          = 0x6 (6)
      model           = 0xf (15)
```

```
      stepping id     = 0x8 (8)
      extended family = 0x0 (0)
      extended model  = 0x8 (8)
      (family synth)  = 0x6 (6)
      (model synth)   = 0x8f (143)
      (simple synth)  = Intel Xeon (unknown type) (Sapphire Rapids) [Sapphire Rapids] {Sunny Co
ve}, 10nm+
[...]
$ cpuid -l 0
CPU 0:
   vendor_id = "GenuineIntel"
```

```
$ cpuid -1 -l 7 | grep AMX
      AMX-BF16: tile bfloat16 support          = true
      AMX-TILE: tile architecture support      = true
      AMX-INT8: tile 8-bit integer support     = true
```

2.2 RDTSC

- RDTSC(Read Time-Stamp Counter)는 프로그램의 정확한 실행시간을 측정하기 위한 명령임. 프로세서의 타임스템프 카운트(64비트 MSR) 값을 읽어서 EDX:EAX 에 저장함.

RDTSC—Read Time-Stamp Counter

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 31	RDTSC	ZO	Valid	Valid	Read time-stamp counter into EDX:EAX.

- 시간 측정 프로그램 예제. 1초동안 sleep을 전후로 TSC값을 측정한 결과 2.4GHz 시스템 동작 클럭값을 확인 할 수 있음

```
// sleeptime.c
#include <stdio.h>
#include <stdint.h>
#include <unistd.h>
// RDTSC를 사용하여 타임스탬프를 읽는 함수
static inline uint64_t rdtsc() {
    unsigned int lo, hi;
    __asm__ volatile ("rdtsc" : "=a"(lo), "=d"(hi));  // EDX(hi):EAX(lo)
    return ((uint64_t)hi << 32) | lo;
}
int main(int argc, char **argv) {
    uint64_t start, end;
    long long unsigned int elapsed;
    start = rdtsc();
    sleep(1); // sleep 1
    end = rdtsc();
    elapsed = end - start;
    printf("sleep Elapsed Time (cycles): %llu\n", elapsed);
}
```

```
$ gcc -o sleeptime  sleeptime.c
./sleeptime
sleep Elapsed Time (cycles): 2400339036
```

```
$ grep "cpu MHz" /proc/cpuinfo
cpu MHz    : 2400.000
cpu MHz    : 2400.000
```

```
$ objdump -j .text --disassemble=rdtsc ./sleeptime
0000000000001169 <rdtsc>:
    1169:  55                   push   %rbp
    116a:  48 89 e5             mov    %rsp,%rbp
    116d:  0f 31                rdtsc
    116f:  89 45 f8             mov    %eax,-0x8(%rbp)
    1172:  89 55 fc             mov    %edx,-0x4(%rbp)
    1175:  8b 45 fc             mov    -0x4(%rbp),%eax
[...]
```

※ RDTSCP (Read TSC and Processor ID) 명령은 타임스탬프 카운터 값을 읽는 것 외에도, CPU의 코어 ID를 반환함. 이는 다중 코어 시스템에서 각 코어의 수행 시간을 측정할 때 유용하다.

※ MSR 테이블에서 TSC 카운타의 레지스터 주소는 10H(16)임.

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address: Hex, Decimal		Architectural MSR Name (Former MSR Name)
Bit Fields	MSR/Bit Description	Comment
Register Address: 6H, 6	IA32_MONITOR_FILTER_SIZE	
See Section 9.10.5, "Monitor/Mwait Address Range Determination."		0F_03H
Register Address: 10H, 16	IA32_TIME_STAMP_COUNTER (TSC)	
See Section 18.17, "Time-Stamp Counter."		05_01H

※ msr-tools 을 이용하면 MSR 정보를 읽을 수 있다. msr 커널 모듈이 설정되어 있어야 함.

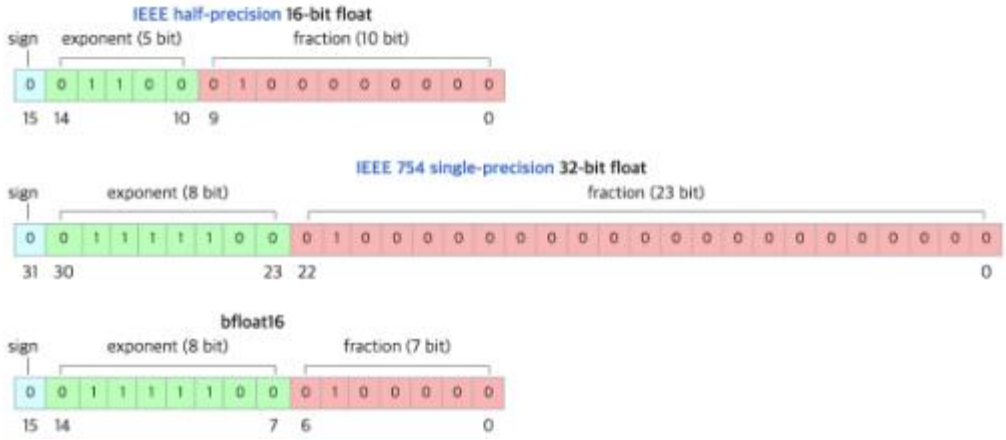
```
# 참고 https://github.com/intel/msr-tools.git
# modprobe  msr
# lsmod ¦ grep msr
msr                    12288  0
$ rdmsr -p 0 0x10   ; 프로세스 0번의 MSR 주소 0x10 를 읽음
410cedfb908
```

2.3 부동소수점 표현

- IEEE 754 [5][6]는 IEEE에서 개발한 컴퓨터에서 부동소수점을 표현하는 표준. 최상위 비트는 부호를 표시하는데 사용, 지수 부분(exponent)과 가수(fraction/mantissa)로 나뉜다.

```
- half precision   : sign 1비트, exponent 5비트,  significand 10비트 (총16비트)
- single precision : sign 1비트, exponent 8비트,  significand 23비트 (총32비트)
- double precision : sign 1비트, exponent 11비트, significand 52비트 (총64비트)
- bfloat16         : sign 1비트, exponent 8비트,  significand 7비트  (총16비트)
```

5) https://en.wikipedia.org/wiki/IEEE_754
6) https://en.wikipedia.org/wiki/Bfloat16_floating-point_format



[그림 1] 부동소수점 타입

※ bfloat16은 32-bit float와 exponent 크기가 8비트로 같기 때문에 하위 32-bit float 에서 16비트를 버리면 쉽게 bfloat16 타입으로 바꿀수 있음

- 십진수에서 변환은 부동소수점 계산기[7][8] 등을 이용하거나, 수기로도 계산할 수 있음. [9]

```
예) 십진수 118.625 을 IEEE754 부동소수점으로 표현하기
  118.625 * 1024 = 121472 ==> 1DA80h ==> 00011101101010000000b  이므로
  118.625 = 00011101101010000000b * 2^-10 이 됨.
  즉 2진수 표현으로 0001110110.1010000000 이고,  1.110110101 * 2^6 가된다.
  sign bit : 양수 이므로 0
  exponent 8 bits : 6 + 127(bias) = 133 = 10000101b
  fraction 23 bits : 11011010100000000000000
  정리하면 0 + 10000101 + 11011010100000000000000
   0100 0010 1110 1101 0100 0000 0000 0000
      4    2    e    d    4    0    0    0 ==> 42ed400
```

- C언어에서 bfloat16 타입을 핸들링하려면 다음과 같이 구조체를 이용하면 가능함. C언어에서는 16비트 half precision float 타입을 지원하지 않기 때문에 32비트 float 타입에서 16비트 right shift 연산을 하면 bfloat16 이 됨.

```c
// bfloatsum.c
#include <stdio.h>
#include <stdint.h>
#include <math.h>
typedef struct {
    uint16_t value;
} bfloat16;

bfloat16 float_to_bfloat16(float f) {
  uint32_t int_value = *(uint32_t*)&f;
  uint16_t bfloat_value = int_value >> 16;
  bfloat16 result;
```

7) https://t.hi098123.com/IEEE-754
8) https://www.h-schmidt.net/FloatConverter/IEEE754.html
9) https://www.youtube.com/watch?v=8afbTaA-gOQ

```
  result.value = bfloat_value;
  return result;
}
float bfloat16_to_float(bfloat16 bf) {
  uint32_t int_value = bf.value << 16;
  float result = *(float*)&int_value;
  return result;
}
bfloat16 bfloat16_add(bfloat16 a, bfloat16 b) {
  float fa = bfloat16_to_float(a);
  float fb = bfloat16_to_float(b);
  float sum = fa + fb;
  return float_to_bfloat16(sum);
}
void print_bfloat16(bfloat16 bf) {
  float f;
  uint32_t int_value;
  printf("bfloat16 value: 0x%04x\n", bf.value);
  f = bfloat16_to_float(bf);
  printf("as float: %f\n", f);
  int_value = *(uint32_t*)&f; // type casting float to uint32_t
  printf("float value: 0x%04x\n", int_value);
}
int main() {
  float a = 3.14;     // bfloat16으로 변환하면서  3.125 가됨
  float b = 2.718;    // bfloat16으로 변환하면서  2.703125 가 됨
  bfloat16 bf_a = float_to_bfloat16(a);
  bfloat16 bf_b = float_to_bfloat16(b);
  bfloat16 bf_sum = bfloat16_add(bf_a, bf_b);
  printf("a ==> \n"); print_bfloat16(bf_a);
  printf("b ==> \n"); print_bfloat16(bf_b);
  printf("a + b ==> \n"); print_bfloat16(bf_sum);
  return 0;
}
```

```
$ gcc -o bfloatsum  bfloatsum.c
$ ./bfloatsum
a ==>
bfloat16 value: 0x4048
as float: 3.125000
float value: 0x40480000
b ==>
bfloat16 value: 0x402d
as float: 2.703125
float value: 0x402d0000
a + b ==>
bfloat16 value: 0x40ba
as float: 5.812500
float value: 0x40ba0000
```

※ 부동소수점 표현에서 정규화된 수에 포함할 수 없는 특별한 값(special values)가 존재함.

- 부호 있는 0 (signed zero) : 지수부와 가수부의 모든 비트가 0인 경우, 부호비트에 따라서 +0과 -0이 존재함. 비교연산 '`+0 == -0`'은 `true`가 되어야 함.

- 무한대: 표현할 수 있는 수의 범위를 넘는 오버플로(overflow)가 발생하였을 때 처리하는 값. 양의무한대는 부호비트:0, 지수비트:모두1, 가수비트:모두0 으로 표현되고, 음의 무한대는 부호비트:1, 지수비트:모두1, 가수비트:모두0으로 표현됨.
- NaN(Not a Number): 연산과정에서 잘못된 입력을 받아 계산하지 못한 경우. (예, `+∞+(-∞)`, `0*∞`, `0÷0`, `∞÷∞`, `sqrt(x<0)`), 지수부: 모든비트1, 가수부:모두 0이 아니어야 함, 부호부 상관없음.
- 비정규화된 수(denormals): IEEE 754에서 가수부는 일의 자리에 1이 생략된 것(1.fraction × $2^{exponent}$)을 가정하지만, 매우작은 숫자, 즉 지수 부분이 가장 낮은 값인 경우는 정규화된 형식으로 표현할 수 없음. 이 경우 비정규화된 표현을 사용함. (0.fraction × $2^{min\text{-}xponent}$)

의미	IEEE754 FP32 비트 표현
+0	0 00000000 00000000000000000000000
-0	1 00000000 00000000000000000000000
+∞	0 11111111 00000000000000000000000
-∞	1 11111111 00000000000000000000000
NaN	1 11111111 00000000000000000000001 (적어도 하나의 비트가 1)
denormals	0 00000000 00000000000000000000001 (가장 작은 비정규화된 수)

- 부호있는 0을 테스트 하는 예제

```
// signzero.c
#include <stdio.h>
#include <math.h>
int main() {
  float positive_zero = 0.0;
  float negative_zero = -0.0;
  printf("1 / +0.0 = %f\n", 1.0 / positive_zero);  // +∞
  printf("1 / -0.0 = %f\n", 1.0 / negative_zero);  // -∞
  if (positive_zero == negative_zero) printf("+0.0 equals to -0.0\n");
  else printf("+0.0 not equals to -0.0\n");
}
```

```
$ gcc -o signzero signzero.c
$ ./signzero
1 / +0.0 = inf
1 / -0.0 = -inf
+0.0 equals to -0.0
```

2.4 연산성능 측정

- CPU의 연산 성능을 간단하게 측정해 보기 위해서 다음과 같은 코드를 작성. double 타입과 int 타입의 연산 성능을 비교. double 곱셈, double 나눗셈, int 곱셈을 반복 실행하고 시간을 측정.

```
// ops.c
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>

static inline uint64_t rdtsc() {
    unsigned int lo, hi;
    __asm__ volatile ("rdtsc" : "=a"(lo), "=d"(hi));
    return ((uint64_t)hi << 32) | lo;
}

int main(int argc, char *argv[]) {
  uint64_t start, end;
  long long unsigned int elapsed;
  int i;
  if (argc < 2) { fprintf(stderr, "Usage: %s <iteration> \n", argv[0]); return 1; }
  int iteration = atoi(argv[1]);

  double da = 1.0, db = 1.0, dc = 0.0;
  start = rdtsc();
  for (i = 0; i < iteration; i++) {
    dc = (da * db) + dc;
  }
  end = rdtsc();
  elapsed = end - start;
  printf("iter=%d double mul+sum Elapsed Time (cycles): %llu\n", iteration, elapsed);

  start = rdtsc();
  for (i = 0; i < iteration; i++) {
    dc = (da / db) + dc;
  }
  end = rdtsc();
  elapsed = end - start;
  printf("iter=%d double div+sum Elapsed Time (cycles): %llu\n", iteration, elapsed);

  int ia = 1.0, ib = 1.0, ic = 0.0;
  start = rdtsc();
  for (i = 0; i < iteration; i++) {
    ic = (ia * ib) + ic;
  }
  end = rdtsc();
  elapsed = end - start;
  printf("iter=%d int mult+sum   Elapsed Time (cycles): %llu\n", iteration, elapsed);

  return 0;
}
```

```
$ gcc -o ops  ops.c
$ ./ops  1000000000
iter=1000000000 double mul+sum Elapsed Time (cycles): 5065466558
iter=1000000000 double div+sum Elapsed Time (cycles): 5065094410
iter=1000000000 int mult+sum   Elapsed Time (cycles): 5063423446
```

- 반복 루프내부에서 곱셈(또는 나눗셈) + 덧셈 2번의 연산이 수행됨.
 연산 횟수 : 반복횟수 * 2번 = 1000000000*2=2000000000
 소요시간 = clocks / 동작HZ(앞의 예제) = 5097465320/2400339036 = 2.12364초
 OPS = 연산횟수/소요시간 = 2000000000/2.12364=941779209 : 941 MOPS
- for loop 한번을 수행하는데 소요된 클럭은 약 5클럭임을 알 수 있다.
  cycles/iteration(cpi) = 5097465320/1000000000=5.09746532

- iteration을 변화시키면서 실행해 보면, iteration이 작을 수록 cpi가 증가함을 알 수 있음.

```
$  (./ops 1; ./ops 2; ./ops 5 ; ./ops 10; ./ops 100; ./ops 1000  ) ¦ grep double ¦ grep mul
iter=1 double mul+sum Elapsed Time (cycles): 168 ; cpi=168
iter=2 double mul+sum Elapsed Time (cycles): 164  ; cpi = 82
iter=5 double mul+sum Elapsed Time (cycles): 184  ; cpi = 36.8
iter=10 double mul+sum Elapsed Time (cycles): 206  ; cpi = 20.6
iter=100 double mul+sum Elapsed Time (cycles): 654  ; cpi = 6.54
iter=1000 double mul+sum Elapsed Time (cycles): 5210  ; cpi = 5.21
```

※ 실행파일을 역어셈블해 보면 다음과 같다. 각 루프는 10개 미만의 어셈블리 명령으로 구성됨을 알 수 있음.

```
$ objdump -d ops
[...]
    1228:       b8 00 00 00 00          mov    $0x0,%eax
    122d:       e8 57 ff ff ff          call   1189 <rdtsc>
    1232:       48 89 45 e8             mov    %rax,-0x18(%rbp)
    1236:       c7 45 bc 00 00 00 00    movl   $0x0,-0x44(%rbp)
    123d:       eb 1c                   jmp    125b <main+0xb4>
    123f:       f2 0f 10 45 d8          movsd  -0x28(%rbp),%xmm0 // for loop start
    1244:       f2 0f 59 45 e0          mulsd  -0x20(%rbp),%xmm0
    1249:       f2 0f 10 4d d0          movsd  -0x30(%rbp),%xmm1
    124e:       f2 0f 58 c1             addsd  %xmm1,%xmm0
    1252:       f2 0f 11 45 d0          movsd  %xmm0,-0x30(%rbp)
    1257:       83 45 bc 01             addl   $0x1,-0x44(%rbp)
    125b:       8b 45 bc                mov    -0x44(%rbp),%eax
    125e:       3b 45 c4                cmp    -0x3c(%rbp),%eax
    1261:       7c dc                   jl     123f <main+0x98> // for loop end
    1263:       b8 00 00 00 00          mov    $0x0,%eax
    1268:       e8 1c ff ff ff          call   1189 <rdtsc>
[...]
```

- CPU 코어가 여러 개일 경우 pthread를 이용하여 각 코어를 동시에 활용할 수 있음

```
형식: #include <pthread.h>
       int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                          void *(*start_routine) (void *), void *arg);
```

```
// rdtsc.h
#include <stdint.h>
extern uint64_t rdtsc(void);
```

```
// rdtsc.c
#include <stdint.h>
uint64_t rdtsc() {
    unsigned int lo, hi;
    __asm__ volatile ("rdtsc" : "=a"(lo), "=d"(hi));
    return ((uint64_t)hi << 32) | lo;
}
```

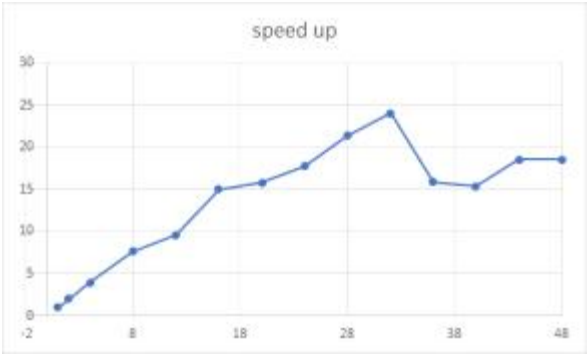
```
// opsth.c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include "rdtsc.h"
typedef struct {
  int thread_no;
  int iteration;
} ThreadData;
void *worker(void *arg) {
  ThreadData *data = (ThreadData *)arg;
  printf("thread no=%d iteration=%d\n", data->thread_no, data->iteration);
  double da = 1.0, db = 1.0, dc = 0.0;
  for (int i = 0; i < data->iteration; i++) {
    dc = (da * db) + dc;
  }
  pthread_exit(NULL);
}
int main(int argc, char *argv[]) {
  int i;
  if (argc < 3) { fprintf(stderr, "Usage: %s <iteration> <#threads>\n", argv[0]); return 1; }
  int iteration = atoi(argv[1]);
  int nthreads = atoi(argv[2]);

  pthread_t  threads[100];
  ThreadData thread_data[100];
  uint64_t start, end;
  long long unsigned int elapsed;

  start = rdtsc();
  for (int i = 0; i < nthreads; i++) { // create threads
    thread_data[i].thread_no = i;
    thread_data[i].iteration = iteration/nthreads;
    pthread_create(&threads[i], NULL, worker, (void *)&thread_data[i]);
  }
  for (int i = 0; i < nthreads; i++) pthread_join(threads[i], NULL);
  end = rdtsc();
  elapsed = end - start;
  printf("iteration=%d nthreads=%d Elapsed Time (cycles): %llu\n",
       iteration, nthreads, elapsed);
  return 0;
}
```

```
gcc -o rdtsc.o  -c rdtsc.c
gcc -o opsth.o  -c opsth.c
```

```
gcc -static -o  opsth  rdtsc.o opsth.o

$ ./opsth  100000000  8
thread no=0 iteration=12500000
thread no=4 iteration=12500000
[...]
thread no=5 iteration=12500000
iteration=100000000 nthreads=8 Elapsed Time (cycles): 66048096
```

- 쓰레드 개수를 변경하면서 테스트해 보면 다음과 같다. (iteration=100000000, 테스트 시스템의 코어는 16개이고 코어당 2개의 쓰레드를 지원, 총32개 쓰레드). 하드웨어에서 지원되는 쓰레드 개수 보다 많으면 성능향상이 없음

nthreads	elapsed cycles	speed up
1	508842000	1.00
2	255154082	1.99
4	129041978	3.94
8	66763582	7.62
12	53081956	9.59
16	33881004	15.02
20	32228414	15.79
24	28647558	17.76
28	23787820	21.39
32	21185288	24.02
36	32137946	15.83
40	33082210	15.38
44	27494846	18.51
48	27446610	18.54



[그림 2] 쓰레드 개수에 따른 성능향상

3. SIMD 프로그래밍

- 병렬처리를 위한 SIMD(single instruction multiple data)를 지원하는 컴퓨터 구조는 지속적으로 발전되었다. X86 에서는 MMX(MultiMedia eXensions), SSE(Streaming SIMD Extensions), AVX(Advanced Vector Extensions), ARM 에서는 NEON, SVE 등이 존재함. GPU 등 병렬처리를 가속화하는 장치들에 사용된다.
- AMD와 Intel에서 SIMD가 지원되는 아키텍쳐를 정리하면 다음과 같다.

[표 1] AMD와 Intel의 아키텍쳐 비교

구분	AMD	Intel
MMX	AMD K6 (1997) [10]	Pentium (1997) [11]
SSE 1	Athlon 64 (2003)[12]	Pentium III (1999)
SSE 2	Athlon 64 (2003)	Pentium 4 (2000)
SSE 3	Athlon 64 Venice (2004)	Pentium 4 Prescott (2004)
SSSE 3	Bobcat, Bulldozer (2011)	Penryn (2007), Bonnell (2008)
SSE 4.1	Jaguar (2013), Bulldozer (2011)	Penryn (2007), Silvermont (2013)
FMA	Piledriver (2012)	Haswell (2013)
AVX 1	Jaguar (2013), Bulldozer (2011)	Sandy Bridge (2011)
AVX 2	Excavator (2015)	Haswell (2013)

3.1 SSE

- SSE(Streaming SIMD Extensions)은 인텔이 1999년 처음 출시한 명령어 집합으로 병렬 처리 지원을 위한 위한 것이다. SSE1, SSE2, SSE3, SSSE3(Supplemental SSE3) 등 세대별로 차이점은 다음과 같다.

[표 2] SSE 세대별 비교

세대	설명
SSE (1999년)	- XMM으로 알려진 128비트 레지스터를 추가. x86에서는 8개, x86_64에서는 16개의 레지스터가 사용가능 - 32비트 float 데이터를 지원. 정수연산은 지원하지 않음. 정수연산은 기존 MMX에서 지원
SSE2 (2001년)	- XMM 레지스터 8개 (128비트, SSE1과 동일) - 32비트 float, 64비트 float, 8/16/32/64 비트 정수를 지원. (MMX를 완전히 대체함)
SSE3 (2004년)	- XMM 레지스터 8개 (128비트, SSE1과 동일) - 지원 데이터 타입도 SSE2와 동일 - 복소수 연산을 위한 가로(수평) 연산 명령 추가, 멀티 쓰레드 환경에서 메모리 접근 최적화, 일부 새로운 부동소수점 변화 명령 추가
SSSE3 (2006년)	- 레지스터는 SSE1과 동일 - 지원 데이터 타입은 SSE3와 동일 - 16가지의 새로운 명령어를 추가함

10) https://en.wikipedia.org/wiki/AMD_K6
11) https://en.wikipedia.org/wiki/Pentium
12) https://en.wikipedia.org/wiki/Athlon_64

SSE4.1 (2006년)	- SSE4는 총 54개의 명령으로 구성, 이중에서 47개의 명령이 SSE4.1로 분류됨
SSE4.2 (2008년)	- STTNI(String and Text New Instruction)을 포함 문자열 검색과 비교를 위한 명령어가 추가됨. 이것은 XML 문서를 파싱하는 속도와 관련됨.

- float 타입의 배열을 덧셈하는 코드를 SSE 확장을 활용하여 작성. 배열의 크기를 충분히 크게 하였을때 각각 실행시간을 비교

```
// vecadd.c
#include <xmmintrin.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "rdtsc.h"
#ifndef N
#define N 128
#endif
static void init_buffer (float *buf, int n, float value) {
  for (int i = 0; i < n; i++) {
    buf[i] = value;
  }
}
static void print_buffer (float *buf, int n) {
  printf("first 10 elements :");
  for (int i = 0; i < 10; i++) {
    printf("%3.1f ", buf[i]);
  }
  printf("\n");
}
#ifdef __SSE__
void vector_add1 (float* a, float* b, float* result, int n) {
  assert(n % 4 == 0);
  for (int i = 0; i < n; i += 4) { // SSE 를 이용
    __m128 va = _mm_loadu_ps(&a[i]);
    __m128 vb = _mm_loadu_ps(&b[i]);
    __m128 vresult = _mm_add_ps(va, vb);
    _mm_storeu_ps(&result[i], vresult);
  }
}
#endif
void vector_add2 (float a[], float b[], float result[], int n) {
  for (int i = 0; i < n; i++) { // SSE를 이용하지 않음
    result[i] = a[i] + b[i];
  }
}
int main() {
  int i;
  uint64_t start, end;
  long long unsigned int elapsed;
  static float va[N], vb[N], r1[N], r2[N];

  init_buffer(va, N, 100);
```

```
  init_buffer(vb, N, 2);
  printf("va = "); print_buffer(va, N);
  printf("vb = "); print_buffer(vb, N);

#ifdef __SSE__
  start = rdtsc();
  vector_add1(va, vb, r1, N);
  end = rdtsc();
  elapsed = end - start;
  printf("with sse Elapsed Time (cycles): %llu\n", elapsed);
#endif
  start = rdtsc();
  vector_add2(va, vb, r2, N);
  end = rdtsc();
  elapsed = end - start;
  printf("without sse Elapsed Time (cycles): %llu\n", elapsed);
  printf("r1 = "); print_buffer(r1, N);
  printf("r2 = "); print_buffer(r2, N);
  return 0;
}
```

```
$ gcc -g  -o vecadd  vecadd.c rdtsc.o  --save-temps -DN=100000000
$ ./vecadd
va = first 10 elements :100.0 100.0 100.0 100.0 100.0 100.0 100.0 100.0 100.0 100.0
vb = first 10 elements :2.0 2.0 2.0 2.0 2.0 2.0 2.0 2.0 2.0 2.0
with sse Elapsed Time (cycles): 480959668
without sse Elapsed Time (cycles): 741395838
r1 = first 10 elements :102.0 102.0 102.0 102.0 102.0 102.0 102.0 102.0 102.0 102.0
r2 = first 10 elements :102.0 102.0 102.0 102.0 102.0 102.0 102.0 102.0 102.0 102.0
 ; SSE를 활용한 경우 실행시간이 약65% (480959668/741395838=0.648) 로 감소됨
```

※ 기본적으로는 SSE가 활성화 되지만. SSE없이 컴파일 하려면 -mno-sse 옵션을 사용할 수 있음. 이 경우 x87 FPU를 이용함을 알 수 있음.

```
gcc  -o vecadd.withsse  vecadd.c  rdtsc.o  --save-temps
gcc  -o vecadd.nosse    vecadd.c  rdtsc.o  --save-temps  -mno-sse
vi -d  vecadd.nosse-vecadd.s   vecadd.withsse-vecadd.s
```

```
// vecadd.withsse-vecadd.s
[...]
vector_add2:
[...]
.L16:
        movl    -4(%rbp), %eax
        cltq
        leaq    0(,%rax,4), %rdx
        movq    -24(%rbp), %rax
        addq    %rdx, %rax
        movss   (%rax), %xmm1
        movl    -4(%rbp), %eax
        cltq
        leaq    0(,%rax,4), %rdx
        movq    -32(%rbp), %rax
        addq    %rdx, %rax
        movss   (%rax), %xmm0
```

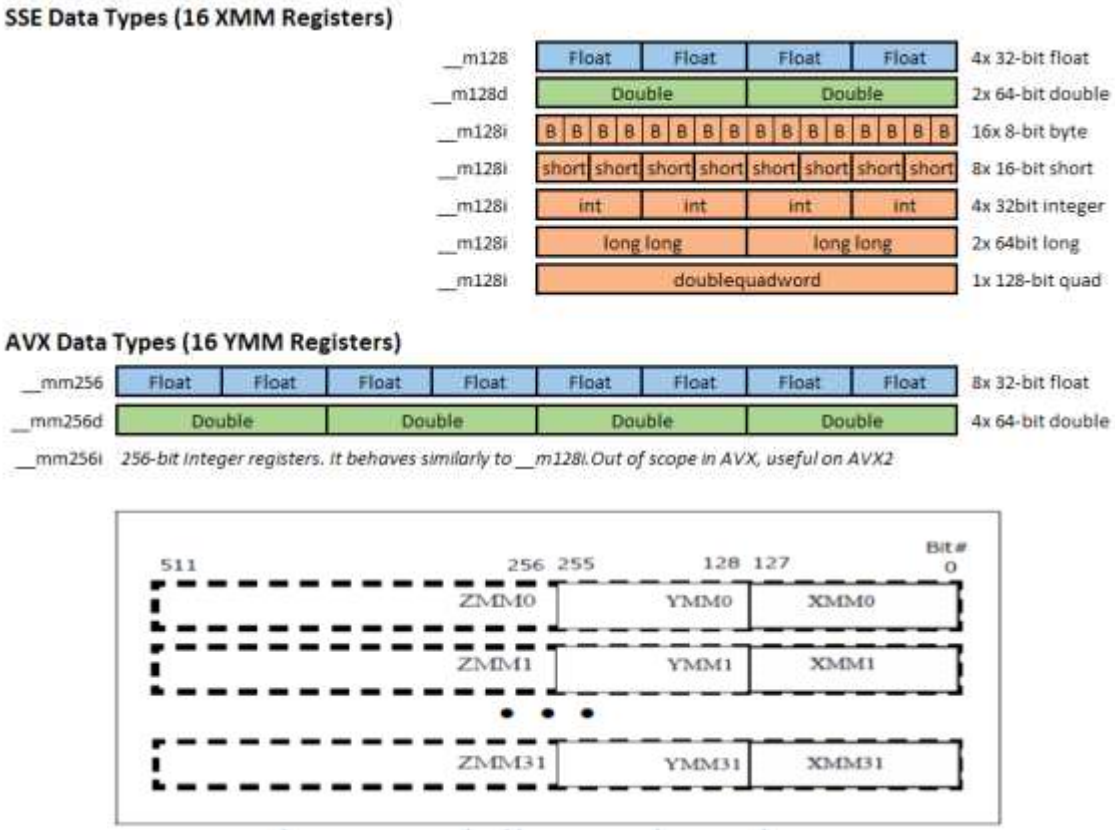
```
// vecadd.nosse-vecadd.s
[...]
vector_add2:
[...]
.L9:
        movl    -4(%rbp), %eax
        cltq
        leaq    0(,%rax,4), %rdx
        movq    -24(%rbp), %rax
        addq    %rdx, %rax
        flds    (%rax)
        movl    -4(%rbp), %eax
        cltq
        leaq    0(,%rax,4), %rdx
        movq    -32(%rbp), %rax
        addq    %rdx, %rax
        flds    (%rax)
```

```
        movl    -4(%rbp), %eax
        cltq
        leaq    0(,%rax,4), %rdx
        movq    -40(%rbp), %rax
        addq    %rdx, %rax
        addss   %xmm1, %xmm0
        movss   %xmm0, (%rax)
        addl    $1, -4(%rbp)
```

```
        movl    -4(%rbp), %eax
        cltq
        leaq    0(,%rax,4), %rdx
        movq    -40(%rbp), %rax
        addq    %rdx, %rax
        faddp   %st, %st(1)
        fstps   (%rax)
        addl    $1, -4(%rbp)
```

3.2 AVX

- AVX(Advanced Vector Extensions)은 고성능 컴퓨팅 및 멀티미디어 응용 프로그램에서 사용되는 SIMD(single instruction multiple data)명령어임. Intel 뿐만 아니라 AMD 프로세서에서도 지원된다. 2011년 Intel Sandy Bridge 아키텍쳐에서 처음으로 지원됨. AVX2는 2013년 Haswell 프로세서부터 지원됨. AVX-512는 2013년 발표되었고, 2016년 출시된 XeonPhi Kinghts Landing(KNL) 프로세서부터 지원됨.
- AVX-512 명령어는 몇 가지 그룹으로 분류됨. 프로세서 종류에 따라 지원되는 그룹이 다름.
- SSE, AVX, AVX2, AVX256 연산 레지스터



[그림 3] SIMD 연산 레지스터

[표 3] AVX 명령 그룹

그룹	설명
AVX512 F	Foundation 대부분의 32비트 및 64비트 부동 소수점 SSE-SSE4.1과 AVX/AVX2 명령어를 EVEX 코딩 방식으로 확장. 512비트 레지스터
AVX512 CD	Conflict Detection Instructions (KNL 프로세서에서 지원)
AVX512 ER	exponent & reciprocal instructions
AVX512 PF	prefetch instructions
AVX512 VL	vector length
AVX512 BW	byte & word instructions
AVX512 DQ	double & quadword instructions
AVX512 IFMA	fused multiply add
AVX512 VBMI	Byte Manipulation Instructions
AVX512 VNNI	neural network instructions

AVX512 GFNI	galois field new instructions: 갈루아 필드를 계산
AVX512 VAES	AES 암호화 명령 지원
AVX512 VBMI2	Vector Byte Manipulation Instructions 2
AVX512 BITALG	bit count/shiffle
AVX512 BF16	bfloat16 instructions
AVX512 FP16	fp16 support

- AVX256 프로그램 예제[13)]

  _mm256_set_ps[14)]는 256비트(32바이트) 벡터를 8개의 float32 값으로 초기화함.

  _mm256_sub_ps[15)]는 float32 벡터 단위의 뺄셈을 수행함. 내부적으로 VSUBPS 명령을 호출.

```
// hello_avx.c
#include <immintrin.h>
#include <stdio.h>
#include "rdtsc.h"
int main() {
   uint64_t start, end;
   long long unsigned int elapsed;

  // 8개의 float32(single precision) 값을 256비트 벡터에 할당함
  __m256 evens = _mm256_set_ps (2.0, 4.0, 6.0, 8.0, 10.0, 12.0, 14.0, 16.0);
  __m256 odds  = _mm256_set_ps (1.0, 3.0, 5.0, 7.0, 9.0, 11.0, 13.0, 15.0);

  start = rdtsc();
  __m256 result = _mm256_sub_ps (evens, odds); // 두 벡터의 차이(subtract)를 구함
  end = rdtsc();
  elapsed = end - start;
  printf("Elapsed Time (cycles): %llu\n", elapsed);

  float* f = (float*)&result; // 결과를 출력
  printf("%f %f %f %f %f %f %f %f \n", f[0], f[1], f[2], f[3], f[4], f[5], f[6], f[7]);
  return 0;
}
```

```
gcc  -mavx  -g  -o hello_avx.o  -c hello_avx.c
gcc  -static -o hello_avx  hello_avx.o  rdtsc.o
```

```
$ ./hello_avx
Elapsed Time (cycles): 96
1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000
```

```
$ objdump  -j .text --disassemble=main  hello_avx
[...]
    1362:       e8 71 01 00 00          call   14d8 <rdtsc>
    1367:       48 89 85 10 ff ff ff    mov    %rax,-0xf0(%rbp)
    136e:       c5 fc 28 85 50 ff ff    vmovaps -0xb0(%rbp),%ymm0
    1375:       ff
    1376:       c5 fc 29 45 90          vmovaps %ymm0,-0x70(%rbp)
    137b:       c5 fc 28 85 70 ff ff    vmovaps -0x90(%rbp),%ymm0
    1382:       ff
```

13) https://yunmorning.tistory.com/31
14) https://www.intel.com/content/www/us/en/docs/cpp-compiler/developer-guide-reference/2021-8/mm256-set-ps.html
15) https://www.intel.com/content/www/us/en/docs/cpp-compiler/developer-guide-reference/2021-8/mm256-sub-ps.html

```
    1383:       c5 fc 29 45 b0          vmovaps %ymm0,-0x50(%rbp)
    1388:       c5 fc 28 45 90          vmovaps -0x70(%rbp),%ymm0
    138d:       c5 fc 5c 45 b0          vsubps -0x50(%rbp),%ymm0,%ymm0
    1392:       c5 fc 29 85 30 ff ff    vmovaps %ymm0,-0xd0(%rbp)
    1399:       ff
    139a:       e8 39 01 00 00          call   14d8 <rdtsc>
[...]
```

※ AVX 명령은 GCC 컴파일러 4.6버전(2011년) 이후 부터 지원되기 시작함.[16] 컴파일을 위해 -mavx 플래그가 필요함.

※ objdump 결과로 확인된 어셈블리코드는 다음과 같다. (예시)

```
1376: c5 fc 29 45 90          vmovaps %ymm0,-0x70(%rbp)
      ----- -- -- --
c5 fc : 2-byte VEX prefix. 두번째 바이트의 비트[2]가 1이면 256-bit vector 연산을 의미함
29 : opcode (opcode 29는 MOVAPS (move aligned packed single precision FP value)를 의미함)
45 : ModR/M (Mod=01b Reg=000b R/M=101b 는 EffectiveAddress=[EBP]+disp8 0번레지스터를 의미함
90 : displacement 로 64-비트 모드에서 sign-extended되어 90H=-112=-70H를 의미함
```

- /usr/lib/gcc/x86_64-linux-gnu/11/include/ 경로에 정의된다. 네이밍 규칙으로 타입명은 __m256, 함수명은 _mm256 으로 시작됨. 아래 typedef 문의 의미는 __m256이라는 새로운 타입을 정의할때 부동 소수점(float) 타입으로 구성되며, 벡터의 전체 길이가 32바이트로 한다는 의미임. float 는 4바이트 이므로, __m256 타입은 8개의 float 요소로 구성됨.

```
  /usr/lib/gcc/x86_64-linux-gnu/11/include/avxintrin.h
typedef float __m256 __attribute__ ((__vector_size__ (32),  __may_alias__));
 /usr/lib/gcc/x86_64-linux-gnu/11/include/avx512fintrin.h
typedef float __m512 __attribute__ ((__vector_size__ (64), __may_alias__));
```

```
// mmsize.c
#include <immintrin.h>
#include <stdio.h>
int main() {
  printf("size of __m256 %ld \n", sizeof(__m256));
  printf("size of __m512 %ld \n", sizeof(__m512));
  return 0;
}
$ gcc -o mmsize mmsize.c
$ ./mmsize
size of __m256 32
size of __m512 64
```

- 위 프로그램을 gdb로 모니터링 한 결과는 다음과 같다.

```
$ gdb  hello_avx
(gdb) break  main
Breakpoint 1 at 0x401745: file hello_avx.c, line 5.
(gdb) run
(gdb) set pagination off
(gdb) disas main         어셈블리 코드에서 vsubps 명령을 확인
[...]
```

16) https://gcc.gnu.org/releases.html

```
   0x000000000040193e <+505>:   call   0x401ab4 <rdtsc>
   0x0000000000401943 <+510>:   mov    %rax,-0xf0(%rbp)
   0x000000000040194a <+517>:   vmovaps -0xb0(%rbp),%ymm0
   0x0000000000401952 <+525>:   vmovaps %ymm0,-0x70(%rbp)
   0x0000000000401957 <+530>:   vmovaps -0x90(%rbp),%ymm0
   0x000000000040195f <+538>:   vmovaps %ymm0,-0x50(%rbp)
   0x0000000000401964 <+543>:   vmovaps -0x70(%rbp),%ymm0
   0x0000000000401969 <+548>:   vsubps -0x50(%rbp),%ymm0,%ymm0
   0x000000000040196e <+553>:   vmovaps %ymm0,-0xd0(%rbp)
   0x0000000000401976 <+561>:   call   0x401ab4 <rdtsc>
[...]
//  디스플레이할 register를 설정함
 display $ymm0.v16_bfloat16
 display $ymm1.v16_bfloat16
(gdb) b *0x0000000000401969     // vsubps 명령에서 break를 설정하고 실행
(gdb) continue
[...]
  // vsubps 실행전 ymm0 값 ==> {16.0, 14.0, 12.0, 10.0, 8.0, 6.0, 4.0, 2.0}
1: $ymm0.v16_bfloat16 = {0, 16, 0, 14, 0, 12, 0, 10, 0, 8, 0, 6, 0, 4, 0, 2}
2: $ymm1.v16_bfloat16 = {0, 7, 0, 5, 0, 3, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0}
(gdb) si    // vsubps 명령 실행 (ymm0 값이 [1.0, 1.0, ..., 1.0] 으로 바뀜 (0x3f800000 =1.0)
1: $ymm0.v16_bfloat16 = {0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1}
2: $ymm1.v16_bfloat16 = {0, 7, 0, 5, 0, 3, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0}
(gdb) print  /x $ymm0.v8_int32[0]
$8 = 0x3f800000  // 0x3f8 ==> 십진수 1.0
(gdb) c
[...]
```

- bfloat16 내적 연산 예제[17]

  _mm512_cvtne2ps_pbh [18] ; convert packed single-precision(32-bit) FP elements in two vectors a and b to packed BF16(16-bit) FP elements, and store the results in single vector dst.

```
// bf16ex.c
#include <immintrin.h>
#include <stdio.h>
#include <math.h>
#include "rdtsc.h"

void hexDump(char *name, void* data, size_t size) {
  const unsigned char* byteData = (const unsigned char*)data;
  for (size_t i = 0; i < size; i++) {
    if (i % 32 == 0) printf("\n%s %04zx: ", name, i);
    if (i % 4 == 3) printf("%02x ", byteData[i]);
    else printf("%02x", byteData[i]);
  }
  printf("\n");
}
```

17) https://www.intel.com/content/www/us/en/developer/articles/technical/intel-deep-learning-boost-new-instruction-bfloat16.html

18) https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#text=_mm512_cvtne2ps_pbh%2520&ig_expand=2671,1898

```
int main() {
  uint64_t start;
  float op1_f32[16], op2_f32[16], op3_f32[16]; // float=32bit * 16 = 512bits

  // register variables
  __m512   v1_f32, v2_f32, v3_f32;
  __m512bh v1_f16, v2_f16, v3_f16;

  start = rdtsc();
  for (int i = 0; i < 16; i++) {
    op1_f32[i]  = i+1;
    op2_f32[i]  = i+32;
    op3_f32[i]  = 1.0;
  }

  start = rdtsc();
  // 길이가 16인 float32 배열을 레지스터에 로딩 (총 512비트)
  v1_f32  = _mm512_loadu_ps(op1_f32); // ps : packed single-precision floating-point
  v2_f32  = _mm512_loadu_ps(op2_f32);
  v3_f32  = _mm512_loadu_ps(op3_f32);
  hexDump("v1_f32 ", &v1_f32, sizeof(v1_f32));
  hexDump("v2_f32 ", &v2_f32, sizeof(v1_f32));
  hexDump("v3_f32 ", &v3_f32, sizeof(v3_f32));

  start = rdtsc();
  // 길이가 16인 2개의 float32 벡터를, 길이가 32인 bf16 백터로 변환
  v1_f16 = _mm512_cvtne2ps_pbh (v1_f32, v2_f32);
  v2_f16 = _mm512_cvtne2ps_pbh (v1_f32, v2_f32);
  v3_f16 = _mm512_cvtne2ps_pbh (v3_f32, v3_f32);
  hexDump("v1_f16 ", &v1_f16, sizeof(v1_f16));
  hexDump("v2_f16 ", &v2_f16, sizeof(v2_f16));

  start = rdtsc();
  // bf16 벡터 2개를 곱하여 누적. FMA
  v3_f32 = _mm512_dpbf16_ps (v3_f32, v1_f16, v2_f16);
  hexDump("v3_f32 ", &v3_f32, sizeof(v3_f32));

  return 0;
}
```

```
$ gcc  -march=native    -g  -o bf16ex.o -c bf16ex.c  --save-temps
$ gcc  -static -o bf16ex  bf16ex.o  rdtsc.o  -Wl,-Map=bf16ex.map
```

```
$ ./bf16ex
; v1 = [1, 2, 3, 4, 5, ..., 16]
v1_f32  0000: 0000803f 00000040 00004040 00008040 0000a040 0000c040 0000e040 00000041
v1_f32  0020: 00001041 00002041 00003041 00004041 00005041 00006041 00007041 00008041
; v2 = [32, 33, 34, ..., 47]
v2_f32  0000: 00000042 00000442 00000842 00000c42 00001042 00001442 00001842 00001c42
v2_f32  0020: 00002042 00002442 00002842 00002c42 00003042 00003442 00003842 00003c42
; v3 = [1, 1, 1, ..., 1]
v3_f32  0000: 0000803f 0000803f 0000803f 0000803f 0000803f 0000803f 0000803f 0000803f
v3_f32  0020: 0000803f 0000803f 0000803f 0000803f 0000803f 0000803f 0000803f 0000803f
; v1_f16, v2_f16 은 동일하며 v1, v2를 merge하여 bf16 으로 변환한 것임
       4200=32 4204=33 4208=34...
```

```
v1_f16  0000: 00420442 08420c42 10421442 18421c42 20422442 28422c42 30423442 38423c42
v1_f16  0020: 803f0040 40408040 a040c040 e0400041 10412041 30414041 50416041 70418041

v2_f16  0000: 00420442 08420c42 10421442 18421c42 20422442 28422c42 30423442 38423c42
v2_f16  0020: 803f0040 40408040 a040c040 e0400041 10412041 30414041 50416041 70418041
; v3_f32 는 v1_f16 과 v2_f16을 곱하여 누적한 것으로, 예를 들어
 첫번째 값 45042000 = 2114d 로 32*32 + 33*33 + 1 와 같다.
 마지막 값 43f10000 = 482d = 15*15 + 16*16 +1
v3_f32  0000: 00200445 00e01445 00a02645 00603945 00204d45 00e06145 00a07745 00308745
v3_f32  0020: 0000c040 0000d041 00007842 0000e442 00003643 00008543 0000b743 0000f143
```

```
$ gdb -ex "break main" -ex "run" -ex "set pagination off"  bf16ex
[...]
   0x0000000000401b12 <+713>:   vdpbf16ps %zmm2,%zmm1,%zmm0
; 위 명령 실행 직전
(gdb) x/i $pc
=> 0x401b12 <main+713>: vdpbf16ps %zmm2,%zmm1,%zmm0

(gdb) print $zmm1.v32_bfloat16
$4 = {32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 1, 2, 3, 4, 5, 6, 7, 8,
9, 10, 11, 12, 13, 14, 15, 16}
(gdb) print $zmm2.v32_bfloat16
$6 = {32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 1, 2, 3, 4, 5, 6, 7, 8,
9, 10, 11, 12, 13, 14, 15, 16}

; 실행 후
(gdb) si
50        return (__m512bh)__builtin_ia32_cvtne2ps2bf16_v32hi(__A, __B);

(gdb) print $zmm0.v16_float
$11 = {2114, 2382, 2666, 2966, 3282, 3614, 3962, 4326, 6, 26, 62, 114, 182, 266, 366, 482}
```

※ Intrinsics Guide에 나와 있는 _mm512_dpbf16_ps [19]동작은 다음과 같다. A, B는 BF16타입이고 길이가 32이므로, 각 배열 요소를 곱하여 연속된 2개요소를 더하고, 결과 값에 누적함.

```
DEFINE make_fp32(x[15:0]) {
    y.fp32  := 0.0
    y[31:16] := x[15:0]
    RETURN y
}
dst := src
FOR j := 0 to 15
    dst.fp32[j] += make_fp32(a.bf16[2*j+1]) * make_fp32(b.bf16[2*j+1])
    dst.fp32[j] += make_fp32(a.bf16[2*j+0]) * make_fp32(b.bf16[2*j+0])
ENDFOR
dst[MAX:512] := 0
```

- 정수 곱셈 (vpmaddwd)

_mm512_madd_epi16 [20] : multipy packed signed 16-bit integers in a and b, producing

19) https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#text=_mm512_dpbf16_ps&ig_expand=1892,1898,4103,1898,2671

20) https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#ig_expand=1892,1898,4103,1898,2671,4204,95,4203,4203&text=_mm512_madd_epi16

signed 32-bit integer

```
// vpmaddwd.cpp
#include <immintrin.h>
#include <iostream>
void print_m512i(__m512i vec) {
    alignas(64) int32_t result[16];
    _mm512_store_epi32(result, vec);
    for (int i = 0; i < 16; ++i) {
        std::cout << result[i] << " ";
    }
    std::cout << std::endl;
}
int main() {
    alignas(64) int16_t a[32] = {
         1,  2,  3,  4,  5,  6,  7,  8, 9, 10, 11, 12, 13, 14, 15, 16,
        17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32 };
    alignas(64) int16_t b[32] = {
         1,  1,  1,  1,  1,  1,  1,  1, 1,  1,  1,  1,  1,  1,  1,  1,
         2,  2,  2,  2,  2,  2,  2,  2, 2,  2,  2,  2,  2,  2,  2,  2 };
    // load vector
    __m512i vec_a = _mm512_load_epi32(a); // extended packed integer
    __m512i vec_b = _mm512_load_epi32(b);
    // _mm512_madd_epi16 : multiply and add
    __m512i  result = _mm512_madd_epi16 (vec_a, vec_b);
    print_m512i(result);
    return 0;
}
```

```
$ g++ -g -static -o vpmaddwd vpmaddwd.cpp  -march=native
$ ./vpmaddwd
3 7 11 15 19 23 27 31 70 78 86 94 102 110 118 126
```

```
$ gdb vpmaddwd
(gdb) disas main
[...]
   0x00005555555555aa <+744>:   vpmaddwd %zmm2,%zmm1,%zmm0{%k1}
; 위 명령 실행 직전
(gdb) print $zmm1.v32_int16
$5 = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 2
5, 26, 27, 28, 29, 30, 31, 32}
(gdb) print $zmm2.v32_int16
$6 = {1 <repeats 16 times>, 2 <repeats 16 times>}
(gdb) si
main () at vpmaddwd.cpp:26
26          __m512i result = _mm512_madd_epi16(vec_a, vec_b);
; 실행 후 ex) [0] 1*1 + 2*1 => 3  [15] 31*2 + 32*2 => 126
(gdb) print $zmm0.v16_int32
$8 = {3, 7, 11, 15, 19, 23, 27, 31, 70, 78, 86, 94, 102, 110, 118, 126}
```

※ 함수명 뒤에 붙는 접미사의 의미는 다음과 같다.

- epi : extended packed integer, 정수형 타입. 8비트, 16비트, 32비트, 64비트
- epu : extended packed unsigned integer, 비부호정 수형 타입. 8비트, 16비트, 32비트, 64비트
- ps : packed single-precision floating-point. 단정밀도 실수, float 16비트

- pd : packed double-precision floating-point. 배정밀도 실수. double 32비트

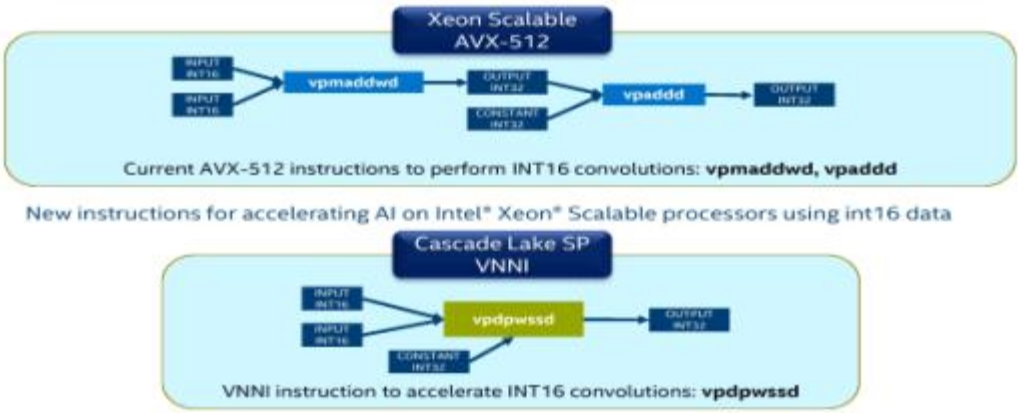
- 정수 덧셈(vpaddd)

```
// vpaddd.cpp
#include <immintrin.h>
#include <iostream>

void print_m512i(__m512i vec) {
    alignas(64) int32_t result[16];
    _mm512_store_epi32(result, vec);
    for (int i = 0; i < 16; ++i) {
        std::cout << result[i] << " ";
    }
    std::cout << std::endl;
}

int main() {
  alignas(64) int32_t a[16] = {
      1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16 };
  alignas(64) int32_t b[16] = {
      16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };
  alignas(64) int32_t c[16] = {
      11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26};

  __m512i vec_a = _mm512_load_epi32(a);
  __m512i vec_b = _mm512_load_epi32(b);
  __m512i vec_c = _mm512_load_epi32(c);

  __m512i result;
  result = _mm512_add_epi32 (vec_a, vec_b); // add two vectors
  print_m512i(result);

  __mmask16 k = 0x00ff;   // 하위 8개 비트만 설정 ==> 하위 8개 요소만 덧셈(a+b) 수행
  result = _mm512_mask_add_epi32 (vec_c, k, vec_a, vec_b);
  print_m512i(result);
  return 0;
}
```

```
$  g++ -g -static -o vpaddd vpaddd.cpp -march=native
$ ./vpaddd
17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17
17 17 17 17 17 17 17 17 19 20 21 22 23 24 25 26
 ; mask 버전의 경우 하위 8개만 덧셈이 되고, 상위 8개는 vec_c가 그대로 복사됨
$ gdb vpaddd
[...]
(gdb) disas main
[...]
   0x000000000040498a <+684>:   vpaddd %zmm0,%zmm1,%zmm0
; 위 명령 실행전
(gdb) print $zmm1.v16_int32
$3 = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16}
(gdb) print $zmm0.v16_int32
$4 = {16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1}
```

```
(gdb) ni
main () at vpaddd.cpp:25
25          __m512i result = _mm512_add_epi32(vec_a, vec_b);
(gdb) print $zmm0.v16_int32
$5 = {17 <repeats 16 times>}
```

- AVX512-VNNI(variable neural network instructions) 연산은 AI 및 딥러닝 작업을 가속화하기 위해 도입한 명령어 세트임. 백터의 곱셈-누적(MAC, Multiply-Accumulate) 연산을 효율적으로 수행할 수 있어 신경망 합성곱 연산에 사용할 수 있음. 내부적으로 `vpdpwssd(int16)`, `vpdpbusd(int8)` 연산. 인텔 캐스케이드 레이크 제온 프로세서에 처음으로 적용됨.



[그림 4] AVX-512 VNNI 연산

```
// vpdpwssd.cpp
#include <immintrin.h>
#include <iostream>
void print_m512i(__m512i vec) {
    alignas(64) int32_t result[16];
    _mm512_store_epi32(result, vec);
    for (int i = 0; i < 16; ++i) {
        std::cout << result[i] << " ";
    }
    std::cout << std::endl;
}
int main() {
    alignas(64) int16_t a[32] = {
         1,  2,  3,  4,  5,  6,  7,  8, 9, 10, 11, 12, 13, 14, 15, 16,
        17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32 };
    alignas(64) int16_t b[32] = {
        32, 31, 30, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18, 17,
        16, 15, 14, 13, 12, 11, 10,  9, 8,  7,  6,  5,  4,  3,  2,  1 };
    alignas(64) int32_t c[16] = {
        100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100 };

    __m512i vec_a = _mm512_load_epi32(a);
    __m512i vec_b = _mm512_load_epi32(b);
    __m512i vec_c = _mm512_load_epi32(c);
    __m512i result = _mm512_dpwssds_epi32 (vec_c, vec_a, vec_b);
```

```
    print_m512i(result);
    return 0;
}
```

```
$ g++ -g -static  -o vpdpwssd  vpdpwssd.cpp  -march=native
$ ./vpdpwssd
194 306 402 482 546 594 626 642 642 626 594 546 482 402 306 194
```

```
$ gdb vpdpwssd
[...]
(gdb) disas main
[...]
   0x0000000000404aa8 <+970>:   vpdpwssds %zmm2,%zmm1,%zmm0
; 위 명령 실행전
(gdb) print $zmm0.v16_int32
$1 = {100 <repeats 16 times>}
(gdb) print $zmm1.v32_int16
$2 = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 2
5, 26, 27, 28, 29, 30, 31, 32}
(gdb) print $zmm2.v32_int16
$3 = {32, 31, 30, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 1
0, 9, 8, 7, 6, 5, 4, 3, 2, 1}
(gdb) si
main () at vpdpwssd.cpp:25
25          __m512i result = _mm512_dpwssds_epi32(vec_c, vec_a, vec_b);
; 실행후. ex) [0] 1*32 + 2*31 + 100 => 194  [8] 17*16 + 18*15 + 100 => 642
(gdb) print $zmm0.v16_int32
$4 = {194, 306, 402, 482, 546, 594, 626, 642, 642, 626, 594, 546, 482, 402, 306, 194}
```

- 성능 비교

```
// vnni.c
#include <immintrin.h>
#include <stdio.h>
#include <stdlib.h>
#include "rdtsc.h"

void print_m512i(__m512i vec) {
  int32_t result[16]  __attribute__((aligned(64)));
  _mm512_store_epi32(result, vec);
  for (int i = 0; i < 16; ++i) {
    printf("%d ", result[i]);
  }
  printf("\n");
}
int main() {
  int i;
  uint64_t start, end;
  long long unsigned int elapsed;
  int16_t a[32]   __attribute__((aligned(64))) = {
       1,  2,  3,  4,  5,  6,  7,  8, 9, 10, 11, 12, 13, 14, 15, 16,
      17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32 };
  int16_t b[32]  __attribute__((aligned(64))) = {
       1,  1,  1,  1,  1,  1,  1,  1, 1,  1,  1,  1,  1,  1,  1,  1,
```

```
      2,  2,  2,  2,  2,  2,  2,  2, 2,  2,  2,  2,  2,  2,  2,  2 };
  int32_t c[16]  __attribute__((aligned(64))) = {
      100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100 };

  __m512i vec_a, vec_b, vec_c, tmp;
  __m512i result;

  start = rdtsc();
  vec_a  = _mm512_load_epi32(a); // a, b:  16bit x 32
  vec_b  = _mm512_load_epi32(b);
  for (i = 0; i < N; i++) { ; multiplay, add 를 따로 계산
    tmp    = _mm512_madd_epi16(vec_a, vec_b); // tmp : 32bit x 16
    result = _mm512_add_epi32(vec_c, tmp); // c = c + tmp
  }
  end = rdtsc();
  elapsed = end - start;
  printf("muladd Elapsed Time (cycles): %llu\n", elapsed);
  print_m512i(result);

  start = rdtsc();
  for (i = 0; i < N; i++) {  ; vnni 계산
    result = _mm512_dpwssds_epi32(vec_c, vec_a, vec_b);
  }
  end = rdtsc();
  elapsed = end - start;
  printf("dpwssds Elapsed Time (cycles): %llu\n", elapsed);
  print_m512i(result);

  return 0;
}
```

```
gcc -g -static -o vnni.o  -c vnni.c -march=native -DN=1000000000
gcc -static -o vnni  vnni.o rdtsc.o
```

```
$ ./vnni
muladd Elapsed Time (cycles): 4716518288
3 7 11 15 19 23 27 31 70 78 86 94 102 110 118 126
dpwssds Elapsed Time (cycles): 3256439740
3 7 11 15 19 23 27 31 70 78 86 94 102 110 118 126
 ; 실행시간 차이는 3256439740/4716518288= 0.69  약 70% 임을 알수 있음
```

- 벡터 비교.

  _mm512_cmp_ps_mask [21] : compare fp32 elements in a and b.

```
// avx512fintrin.h
 typedef unsigned short __mmask16;
// avxintrin.h
[...]
/* Equal (ordered, non-signaling)  */
#define _CMP_EQ_OQ      0x00
/* Less-than (ordered, signaling)  */
```

21) https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#text=_mm512_cmp_ps&ig_expand=828

```
#define _CMP_LT_OS      0x01
[...]
```

```
// compare.c
#include <immintrin.h>
#include <stdlib.h>
#include <stdio.h>

void print_m512i(__m512 vec) {
  float result[16] __attribute__((aligned(64)));
  _mm512_store_ps(result, vec);
  for (int i = 0; i < 16; ++i) {
    printf("%f ", result[i]);
  }
  printf("\n");
}
int main() {
  float a[16] __attribute__((aligned(64)))
     = { 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16 };
  float b[16] __attribute__((aligned(64)))
     = { 16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1 };
  float c[16] __attribute__((aligned(64)))
     = { 8,8,8,8, 0,0,0,0, 0,0,0,0, 8,8,8,8 };
  __m512 vec_a = _mm512_loadu_ps(a);
  __m512 vec_b = _mm512_loadu_ps(b);
  __m512 vec_c = _mm512_loadu_ps(c);
  __mmask16 mask ;
  mask = _mm512_cmp_ps_mask (vec_a, vec_b, _CMP_LT_OS);
  printf("mask : %04X \n", mask);   // a가 b 보다 작은지 비교(하위 8비트는 true)

  mask = _mm512_cmp_ps_mask (vec_b, vec_c, _CMP_GT_OS);
  printf("mask : %04X \n", mask); // b가 c보다 큰지 비교(상위 4비트는 false)
  return 0;
}
```

```
$ gcc -g -static -o compare compare.c -march=native
$ ./compare
mask : 00FF
mask : 0FFF
```

```
$ gdb compare
[...]
첫 번째 비교 명령
   0x0000000000401c04 <+935>:   vcmpltps 0x80(%rsp),%zmm0,%k0{%k1}
; 위 명령 실행전
(gdb) print $zmm0.v16_float
$1 = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16} // vec_a
(gdb) print /x $k0
$2 = 0x10000000
(gdb) print /x $k1
$3 = 0xffff
(gdb) print $rsp+0x80
$4 = (void *) 0x7fffffffe100
(gdb) x/16x 0x7fffffffe100   // vec_b
0x7fffffffe140:  0x41800000  0x41700000  0x41600000  0x41500000
0x7fffffffe150:  0x41400000  0x41300000  0x41200000  0x41100000
```

```
0x7fffffffe160:  0x41000000  0x40e00000  0x40c00000  0x40a00000
0x7fffffffe170:  0x40800000  0x40400000  0x40000000  0x3f800000
; 실행 후
(gdb) si
0x0000000000401c0d      22        mask = _mm512_cmp_ps_mask (vec_a, vec_b, _CMP_LT_OS);
(gdb) print /x $k0
$5 = 0xff

두 번째 비교 명령
   0x0000000000401c42 <+997>:   vcmpgtps 0xc0(%rsp),%zmm0,%k3{%k2}
; 실행전
(gdb) print /x $k3
$6 = 0x0
(gdb) print /x $k2
$7 = 0xffff
(gdb) print $zmm0.v16_float
$8 = {16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1} // vec_b
(gdb) print $rsp+0xc0
$9 = (void *) 0x7fffffffe140
(gdb) x/16x 0x7fffffffe140    // vec_c
0x7fffffffe180:  0x41080000  0x41080000  0x41080000  0x41080000
0x7fffffffe190:  0x00000000  0x00000000  0x00000000  0x00000000
0x7fffffffe1a0:  0x00000000  0x00000000  0x00000000  0x00000000
0x7fffffffe1b0:  0x41080000  0x41080000  0x41080000  0x41080000
(gdb) si
0x0000000000401c4b  33    mask = _mm512_cmp_ps_mask (vec_b, vec_c, _CMP_GT_OS);
; 실행후
(gdb) print /x $k3
$10 = 0xfff
```

※ AVA-512 에서는 8개의 opmask(오퍼레이션 마스크) 레지스터(k0-k7)를 지원한다. 레지스터의 너비는 MAX_KL(64비트) 임. 8개중 7개(k1-k7) 레지스터는 EVEX 인코딩 명령어를 위해 사용된다. k0 레지스터는 특별한 레지스터로 보통 모든 비트가 1로 설정된 상태를 의미함.

- 벡터 비교를 위한 코드를 인라인 어셈블리로 작성하면 아래와 같다. 제약자(constraint) 문자 “x"는 벡터 레지스터를, "Yk"는 마스크 레지스터를 의미함.

```
// compare2.c
#include <immintrin.h>
#include <stdio.h>
int main() {
  __m512 vec_a = _mm512_set_ps(1,2,3,4, 5,6,7,8, 1,2,3,4, 5,6,7,8);
  __m512 vec_b = _mm512_set_ps(3,3,3,3, 3,3,3,3, 3,3,3,3, 3,3,3,3);
  __mmask16 mask1 = 0xFF00;
  __mmask16 mask2;
// vec_a가 vec_b 보다 작은지 비교, mask1에 의해서 상위 8개 요소만 비교대상으로함
  __asm__ volatile ( "vcmpltps %[value_b], %[value_a], %[mask2]%{%[mask1]%}"
    : [mask2] "=Yk" (mask2)
    : [value_b] "x"(vec_b),
      [value_a] "x"(vec_a),
      [mask1] "Yk" (mask1)
```

```
  );
  printf("(mask2): 0x%04x\n", mask2);
  return 0;
}
```

```
$ gcc -g -static -o compare2  compare2.c  -march=native
$ ./compare2
(mask2): 0xc000
```

```
$ gdb compare2
[...]
   0x0000000000401aee <+937>:   vcmpltps %zmm0,%zmm1,%k1{%k2}
; 위 명령 실행전
(gdb) print /x $k1
$1 = 0x8401100
(gdb) print /x $k2
$2 = 0xff00
(gdb) print $zmm0.v16_float
$3 = {3 <repeats 16 times>}
(gdb) print $zmm1.v16_float
$4 = {8, 7, 6, 5, 4, 3, 2, 1, 8, 7, 6, 5, 4, 3, 2, 1}

; 실행후
(gdb) print /x $k1
$5 = 0xc000
```

3.3 AMX

- 인텔 고급 매트릭스 확장(Advanced Matrix Extensions, AMX)[22]는 인공지능 및 기계학습을 워크로드를 위한 아키텍쳐로 2020년에 발표되었고, 2023년에 출시된 Sapphire Rapids 프로세서에 적용됨. 타일(tile)이라고 부르는 2차원 구조의 레지스터(TMUL, tile matrix multiply unit)를 단위로 연산이 가능함.
- 1KB 크기의 2차원 레지스터 6개로 구성됨

그룹	설명
AMX BF16	bfloat16 support
AMX INT8	tile 8-bit integer support
AMX TILE	tile architecture support

- AMX 명령

명령	설명
LDTILECFG	Load tile configuration
STTILECFG	Store tile configuration
TDPBF16PS	Dot product of BF16 tiles accumulated into packed single precision tile.
TDPBSSD	Dot product of signed bytes with dword accumulation.
TDPBSUD	Dot product of signed/unsigned bytes with dword accumulation.
TDPBUSD	Dot product of unsigned/signed bytes with dword accumulation.
TDPBUUD	Dot product of unsigned bytes with dword accumulation.
TILELOADD	Load data into tile.
TILELOADDTI	Load data into tile with hint to optimize data caching.
TILERELEASE	Release tile.
TILESTORED	Store tile.
TILEZERO	Zero tile.

<table>
<tr><th>명령</th><th>설명</th></tr>
<tr><td>_tile_loadconfig
23)</td><td>void _tile_loadconfig (const void * mem_addr)
- 메모리 주소에서 타일 설정(64바이트)을 읽음
- 설정 내용은 팔래트 유형, 행당 바이트 수 및 행 수를 표시, palette_id 가 0이면 타일 구성과 타일 데이터 모두에 대한 초기화 상태를 나타내며 타일은 0이 됨.
- 잘못된 구성으로 인해 #GP (General Protection fault) 오류가 발생함
<table>
<tr><th>오프셋</th><th>내용</th></tr>
<tr><td>0</td><td>palette_id</td></tr>
<tr><td>1</td><td>startRow(8b)</td></tr>
<tr><td>2-15</td><td>reverved (must be zero)</td></tr>
<tr><td>16-17</td><td>tile0.colb - bytes_per_row</td></tr>
<tr><td>18-19</td><td>tile1.colb</td></tr>
<tr><td>20-21</td><td>tile2.colb</td></tr>
<tr><td>...</td><td>...</td></tr>
<tr><td>30-31</td><td>tile7.colb</td></tr>
<tr><td>32-47</td><td>reverved (must be zero)</td></tr>
<tr><td>48</td><td>tile0.rows</td></tr>
<tr><td>49</td><td>tile1.rows</td></tr>
</table>
</td></tr>
</table>

22) https://en.wikichip.org/wiki/x86/amx

	50 tile2.rows ... ... 55 tile7.rows 56-63 reverved (must be zero)
_tile_loadd	void _tile_loadd (__tile dst, const void * base, int stride) - 타일의 행 데이터를 명시된 메모리 주소 base와 stride에 의해 대상 타일 dst로 로드함
_tile_stored	void _tile_stored (__tile src, void * base, int stride) - 이전에 "_tile_loadconfig"를 통해 구성한 타일 구성을 사용하여 "src"로 지정된 타일을 "base" 주소 및 "stride"로 지정된 메모리에 저장
_tile_dpbssd	void _tile_dpbssd (__tile dst, __tile a, __tile b) - 소스/대상 누산기를 사용하여 타일에서 바이트의 내적을 계산 - "a"의 부호 있는 8비트 정수로 구성된 4개의 인접한 쌍 그룹과 "b"의 해당 부호 있는 8비트 정수를 곱하여 중간 32비트 결과 4개를 생성 - 4개의 결과를 "dst"의 해당 32비트 정수와 합산하고 32비트 결과를 다시 타일 "dst"에 저장

- 하드웨어에서 AMX를 지원하는지 여부는 cpuid 명령을 이용한다.

```
$ cpuid -1 ¦ grep -i AMX
      AMX-BF16: tile bfloat16 support          = true
      AMX-TILE: tile architecture support      = true
      AMX-INT8: tile 8-bit integer support     = true
$ cat /proc/cpuinfo
[...]
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflu
sh mmx fxsr sse sse2 ss ht syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon rep_good nopl
xtopology nonstop_tsc cpuid aperfmperf tsc_known_freq pni pclmulqdq monitor ssse3 fma cx16 pdcm
pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand hypervisor
lahf_lm abm 3dnowprefetch invpcid_single ssbd ibrs ibpb stibp ibrs_enhanced fsgsbase tsc_adjust
bmi1 avx2 smep bmi2 erms invpcid avx512f avx512dq rdseed adx smap avx512ifma clflushopt clwb av
x512cd sha_ni avx512bw avx512vl xsaveopt xsavec xgetbv1 xsaves avx_vnni avx512_bf16 wbnoinvd id
a arat avx512vbmi umip pku ospke waitpkg avx512_vbmi2 gfni vaes vpclmulqdq avx512_vnni avx512_b
italg tme avx512_vpopcntdq rdpid cldemote movdiri movdir64b md_clear serialize amx_bf16 avx512_
fp16 amx_tile amx_int8 flush_l1d arch_capabilities
[...]
```

- 타일 설정

```
// amx1.c (loadconfig)
#include <immintrin.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <sys/syscall.h>
#include <unistd.h>
#include <stdbool.h>
```

23) https://www.intel.com/content/www/us/en/docs/cpp-compiler/developer-guide-reference/2021-10/tile-loadconfig.html

```
#define MAX_ROWS 16
#define MAX_COLS 64
#define ARCH_GET_XCOMP_PERM     0x1022
#define ARCH_REQ_XCOMP_PERM     0x1023
#define XFEATURE_XTILECFG       17
#define XFEATURE_XTILEDATA      18

// Define tile config data structure
typedef struct __tile_config {
  uint8_t palette_id;
  uint8_t start_row;
  uint8_t reserved_0[14];
  uint16_t colsb[16];
  uint8_t rows[16];
} __tilecfg;

// Initialize tile config
static void init_tile_config (__tilecfg *cfg) {
  cfg->palette_id = 1;  // 팔레트 id
  cfg->start_row = 0;
  // 사용할 타일의 행,열의 수를 정의
  cfg->rows[0]  = MAX_ROWS; cfg->colsb[0] = MAX_ROWS;
  cfg->rows[1]  = MAX_ROWS; cfg->colsb[1] = MAX_COLS;
}

// syscall을 사용하여 XFEATURE_XTILEDATA 기능을 사용 권한을 OS에게 요청
static bool set_tiledata_use() {
  if (syscall(SYS_arch_prctl, ARCH_REQ_XCOMP_PERM, XFEATURE_XTILEDATA)) {
    fprintf(stderr, "\n Fail to do XFEATURE_XTILEDATA \n\n"); return false;
  } else {
    printf("Successfully set permission for XFEATURE_XTILEDATA.\n");
  }
  return true;
}

int main() {
  __tilecfg tile_data = {0}; // 설정 데이터 0으로 초기화됨
  if (!set_tiledata_use()) exit(-1); // 커널에 AMX 사용 권한을 요청
  init_tile_config (&tile_data); // 설정 정보를 로드
  _tile_loadconfig (&tile_data); // AMX 설정을 초기화

  _tile_release (); // AMX 사용을 해제
}
```

```
$ gcc -g -static  -mamx-tile -o amx1  amx1.c
$ ./amx1
Successfully set permission for XFEATURE_XTILEDATA.

$ gdb amx1
$ disas main
[...]
   0x00000000004018cf <+142>:   ldtilecfg (%rax)
[...]
```

※ GCC에서는 11버전 부터 AMX를 지원함

- 타일 설정에 문제가 있을 경우 #GP fault 가 발생함. 다음과 같이 시그널 핸들러를 등록하여 처리가 가능하다

```
// amx2.c (amx1.c 에서 수정) signal handler
[...]
// signal_handler
#include <signal.h>
#include <setjmp.h>

jmp_buf jump_buffer;
void signal_handler (int signal) {
  if (signal == SIGSEGV) {
    printf("Caught segmentation fault (SIGSEGV)!\n");
    longjmp(jump_buffer, 1);
  }
}

// Initialize tile config
static void init_tile_config (__tilecfg *cfg) {
  cfg->palette_id = 1;  // 팔레트 id
  cfg->start_row = 0;
  cfg->reserved_0[0] = 1; // reserved 값은 0이어야 하나, 고의적으로 1로 설정하여 테스트
                          // SIGSEGV 가 발생함
  // 사용할 타일의 행,열의 수를 정의
  cfg->rows[0]  = MAX_ROWS; cfg->colsb[0] = MAX_ROWS;
  cfg->rows[1]  = MAX_ROWS; cfg->colsb[1] = MAX_COLS;
}
int main() {
  if (!set_tiledata_use()) exit(-1);
  signal(SIGSEGV, signal_handler); // register a signal handler

  __tilecfg tile_data = {0};
  init_tile_config (&tile_data);
  if (setjmp(jump_buffer) == 0) {
    _tile_loadconfig (&tile_data);
  } else {
    fprintf(stderr, "some configuration error.\n");
    exit(1);
  }
  _tile_release ();
}
```

```
$ gcc -g -static  -mamx-tile -o amx2  amx2.c
$ ./amx2
Successfully set permission for XFEATURE_XTILEDATA.
Caught segmentation fault (SIGSEGV)!
some configuration error.
```

- 데이터 초기화. 매트릭스 A, B, C를 초기한다. A와 B는 int8, C는 int32임

```
// amx3.c (amx2.c 에서 수정) tile_load
[...]

#define MAX_ROWS 4
```

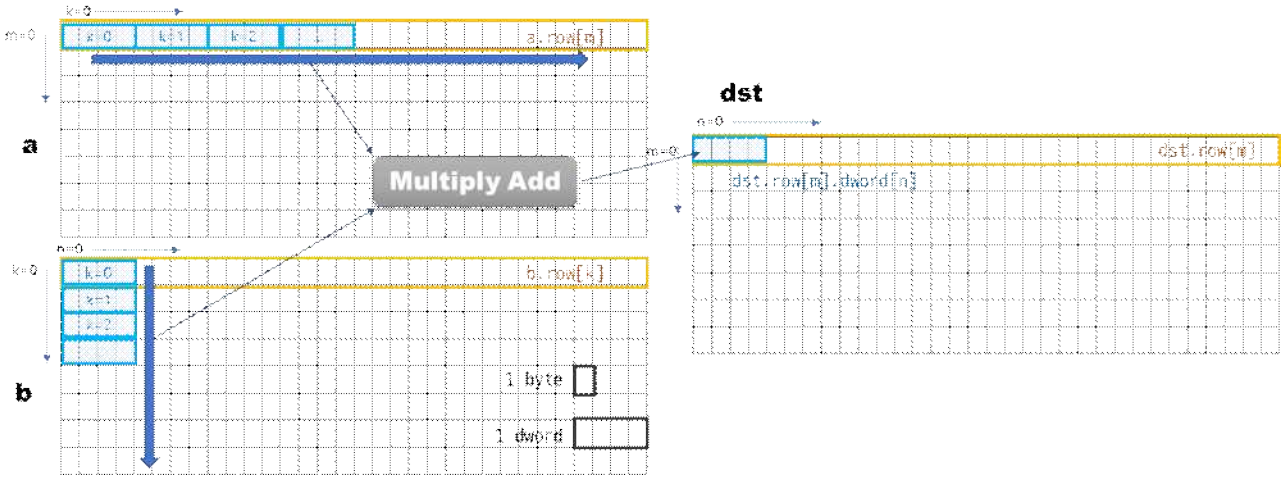
```
#define MAX_COLS 16
#define MAX      1024
#define STRIDE   MAX_COLS

// Initialize int8_t buffer
static void init_buffer8 (int8_t *buf) {
  int rows, colsb, i, j;
  int8_t v;
  rows  = MAX_ROWS;
  colsb = MAX_COLS;
  for (i = 0; i < rows; i++) {
    for (j = 0; j < colsb; j++) {
      if (j % 4 == 0) v = 0; else v = j;
      buf[i * colsb + j] = v;
    }
  }
}
// Initialize int32_t buffer
static void init_buffer32 (int32_t *buf) {
  int rows, colsb, i, j;
  rows  = MAX_ROWS;
  colsb = MAX_COLS;
  int colsb2=colsb/4;
  for (i = 0; i < rows; i++)
    for (j = 0; j < (colsb2); j++)
        buf[i * colsb2 + j] = 0;
}

// Print int8_t buffer
static void print_buffer8(int8_t* buf, int32_t rows, int32_t colsb) {
  printf("print_buffer8 rows=%d colsb=%d\n", rows, colsb);
  for (int i = 0; i < rows; i++) {
    for (int j = 0; j < (colsb); j++) {
       printf("%d ", buf[i * colsb + j]);
    }
    printf("\n");
  }
  printf("\n");
}
// Print int32_t buffer
static void print_buffer32(int32_t* buf, int32_t rows, int32_t colsb) {
  printf("print_buffer32 rows=%d colsb=%d\n", rows, colsb);
  for (int i = 0; i < rows; i++)  {
    for (int j = 0; j < colsb; j++) {
       printf("%d ", buf[i * colsb + j]);
    }
    printf("\n");
  }
  printf("\n");
}
int main() {
[...]
  int8_t src1[MAX];
  int8_t src2[MAX];
```

```
  int32_t res[MAX/4];
  int rows  = MAX_ROWS;
  int colsb = MAX_COLS;
  init_buffer (src1, 0, 1); print_buffer8(src1, rows, colsb);
  init_buffer (src2, 0, 2); print_buffer8(src2, rows, colsb);
  init_buffer32 (res, 100); print_buffer32(res, rows, colsb/4);
  // Load tile rows from memory
  _tile_loadd (2, src1, STRIDE);  // tile 2번에 로드
  _tile_loadd (3, src2, STRIDE);  // tile 3번에 로드
  _tile_loadd (1, res, STRIDE);   // tile 1번에 로드
[...]
$ gcc -g -static  -mamx-tile -o amx3  amx3.c
$ ./amx3
print_buffer8 rows=4 colsb=16
0 1 2 3 0 5 6 7 0 9 10 11 0 13 14 15
0 1 2 3 0 5 6 7 0 9 10 11 0 13 14 15
0 1 2 3 0 5 6 7 0 9 10 11 0 13 14 15
0 1 2 3 0 5 6 7 0 9 10 11 0 13 14 15

print_buffer8 rows=4 colsb=16
0 1 2 3 0 5 6 7 0 9 10 11 0 13 14 15
0 1 2 3 0 5 6 7 0 9 10 11 0 13 14 15
0 1 2 3 0 5 6 7 0 9 10 11 0 13 14 15
0 1 2 3 0 5 6 7 0 9 10 11 0 13 14 15

print_buffer32 rows=4 colsb=4
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
```

- _tile_dpbssd [24]의 동작은 다음과 같다.

```
DEFINE DPBD(x, y) {
    tmp1 := SignExtend32(x.byte[0]) * SignExtend32(y.byte[0])
    tmp2 := SignExtend32(x.byte[1]) * SignExtend32(y.byte[1])
    tmp3 := SignExtend32(x.byte[2]) * SignExtend32(y.byte[2])
    tmp4 := SignExtend32(x.byte[3]) * SignExtend32(y.byte[3])
    RETURN tmp1 + tmp2 + tmp3 + tmp4
}
FOR m := 0 TO dst.rows - 1
    FOR k := 0 TO (a.colsb / 4) - 1
        FOR n := 0 TO (dst.colsb / 4) - 1
            dst.row[m].dword[n] += DPBD(a.row[m].dword[k], b.row[k].dword[n])
        ENDFOR
    ENDFOR
    write_row_and_zero(dst, m, tmp, dst.colsb)
ENDFOR
zero_upper_rows(dst, dst.rows)
zero_tileconfig_start()
```

24) https://www.intel.com/content/www/us/en/docs/cpp-compiler/developer-guide-reference/2021-10/tile-dpbssd.html



[그림 5] 타일 행렬 곱셈 연산

- 매트릭스 곱셈(TMUL INT8)

```
// amx4.c (amx3.c 에서 수정) tmul
[...]

int main() {
[...]
  // Load tile rows from memory
  _tile_loadd (2, src1, STRIDE);
  _tile_loadd (3, src2, STRIDE);
  _tile_loadd (1, res, STRIDE);
  // tile 2와 3을 곱하여 1에 저장
  _tile_dpbssd (1, 2, 3);
  _tile_stored (1, res, STRIDE);
  print_buffer32(res, rows, colsb/4);
  _tile_release ();
}
$ gcc -g -static  -mamx-tile -o amx4  amx4.c
$ ./amx4
[...]

print_buffer32 rows=4 colsb=4
200 584 968 1352
200 584 968 1352
200 584 968 1352
200 584 968 1352
```

※ 예를 들어 dst[m=0][n=2] =968 은 다음과 같이 계산된다.

m=0,n=2	A.row[0].dword[0]	A.row[0].dword[1]	A.row[0].dword[2]	A.row[0].dword[3]
dword	0 1 2 3	0 5 6 7	0 9 10 11	0 13 14 15
	B.row[0].dword[2]	B.row[1].dword[2]	B.row[2].dword[2]	B.row[3].dword[2]
dword	0 9 10 11	0 9 10 11	0 9 10 11	0 9 10 11
dot product	0+9+20+33	0+45+60+77	0+81+100+121	0+117+140+165
total sum	0+9+20+33+0+45+60+77+0+81+100+121+0+117+140+165=968			

- 성능 측정. TSC 카운터를 이용하여 곱셉연산에 소요되는 클럭을 측정.

```
// amx5.c
#include "rdtsc.h"

#ifndef MAX_ROWS
#define MAX_ROWS 4
#endif
#ifndef MAX_COLS
#define MAX_COLS 16
#endif
int main() {
[...]
  uint64_t start, end;
  long long unsigned int elapsed;
  start = rdtsc();
  _tile_dpbssd (1, 2, 3); // Compute dot-product of bytes in tiles
  end = rdtsc();
  elapsed = end - start;
  printf("_tile_dpbssd Elapsed Time (cycles): %llu\n", elapsed);
[...]
$ gcc -g -static  -mamx-tile -o amx5  amx5.c  rdtsc.o -DMAX_ROWS=16 -DMAX_COLS=64
$ ./amx5
[...]
_tile_dpbssd Elapsed Time (cycles): 190
[...]
$ clang -g -static  -march=native  -o amx5  amx5.c  rdtsc.o -DMAX_ROWS=16 -DMAX_COLS=64
$ ./amx5
[...]
_tile_dpbssd Elapsed Time (cycles): 190
[...]
```

- 매트릭스 곱셈(TMUL BFLOAT16), int8 버전과 차이점은 int8을 4개씩 묶어서 int32에 합산한 것과 달리. bfloat16은 2개씩 묶어 float32에 합산한다. bfloat16 값을 저장하기위한 구조체와 타입변경을 위한 함수를 필요로 함.

```
// amx6.c
[...]
typedef struct {
  uint16_t value;
} bfloat16;

bfloat16 float_to_bfloat16(float f) {
  uint32_t int_value = *(uint32_t*)&f;
  uint16_t bfloat_value = int_value >> 16;
  bfloat16 result;
  result.value = bfloat_value;
  return result;
}
float bfloat16_to_float(bfloat16 bf) {
  uint32_t int_value = bf.value << 16;
  float result = *(float*)&int_value;
  return result;
}

// Initialize bf16 buffer
```

```
static void init_buffer_bf16 (bfloat16 *buf) {
  int rows, colsb, i, j;
  bfloat16 v;
  rows  = MAX_ROWS;
  colsb = MAX_COLS;
  int colsb2 = colsb/2;
  for (i = 0; i < rows; i++) {
    for (j = 0; j < colsb/2; j++) {
      if (j % 4 == 0) v = float_to_bfloat16(1.0);
      else v = float_to_bfloat16(j);
      buf[i * colsb2 + j] = v;
    }
  }
}
// Initialize float buffer
static void init_buffer32 (float *buf) {
  int rows, colsb, i, j;
  rows  = MAX_ROWS;
  colsb = MAX_COLS;
  int colsb2=colsb/4;
  for (i = 0; i < rows; i++)
    for (j = 0; j < (colsb2); j++)
        buf[i * colsb2 + j] = 0.0;
}

// Print bf16 buffer
static void print_buffer_bf16(bfloat16 *buf, int32_t rows, int32_t colsb) {
  printf("print_buffer_bf16 rows=%d colsb=%d\n", rows, colsb);
  bfloat16 v;
  float f;
  for (int i = 0; i < rows; i++) {
    for (int j = 0; j < (colsb); j++) {
       v = buf[i * colsb + j];
       f = bfloat16_to_float(v);
       printf("%3.1f ", f);
    }
    printf("\n");
  }
  printf("\n");
}
// Print float32 buffer
static void print_buffer32(float *buf, int32_t rows, int32_t colsb) {
  float v;
  printf("print_buffer32 rows=%d colsb=%d\n", rows, colsb);
  for (int i = 0; i < rows; i++)  {
    for (int j = 0; j < colsb; j++) {
       v = buf[i * colsb + j];
       printf("%3.1f ", v);
    }
    printf("\n");
  }
  printf("\n");
}
```

```
int main() {
[...]
  bfloat16 src1[MAX/2];
  bfloat16 src2[MAX/2];
  float res[MAX/4];
  int rows  = MAX_ROWS;
  int colsb = MAX_COLS;

  init_buffer_bf16 (src1); print_buffer_bf16(src1, rows, colsb/2);
  init_buffer_bf16 (src2); print_buffer_bf16(src2, rows, colsb/2);
  init_buffer32 (res);     print_buffer32(res, rows, colsb/4);

  // Load tile rows from memory
  _tile_loadd (2, src1, STRIDE);
  _tile_loadd (3, src2, STRIDE);
  _tile_loadd (1, res,  STRIDE);

  uint64_t start, end;
  long long unsigned int elapsed;
  start = rdtsc();
  _tile_dpbf16ps (1, 2, 3); // Compute dot-product
  end = rdtsc();
  elapsed = end - start;
  printf("dp Elapsed Time (cycles): %llu\n", elapsed);
  _tile_stored (1, res, STRIDE);
  print_buffer32(res, rows, colsb/4);
  _tile_release ();
}
```

```
$ gcc  -g -static  -march=native -o amx6  amx6.c  rdtsc.o -DMAX_ROWS=8 -DMAX_COLS=32
$ ./amx6
print_buffer_bf16 rows=8 colsb=16
1.0 1.0 2.0 3.0 1.0 5.0 6.0 7.0 1.0 9.0 10.0 11.0 1.0 13.0 14.0 15.0
1.0 1.0 2.0 3.0 1.0 5.0 6.0 7.0 1.0 9.0 10.0 11.0 1.0 13.0 14.0 15.0
1.0 1.0 2.0 3.0 1.0 5.0 6.0 7.0 1.0 9.0 10.0 11.0 1.0 13.0 14.0 15.0
1.0 1.0 2.0 3.0 1.0 5.0 6.0 7.0 1.0 9.0 10.0 11.0 1.0 13.0 14.0 15.0
1.0 1.0 2.0 3.0 1.0 5.0 6.0 7.0 1.0 9.0 10.0 11.0 1.0 13.0 14.0 15.0
1.0 1.0 2.0 3.0 1.0 5.0 6.0 7.0 1.0 9.0 10.0 11.0 1.0 13.0 14.0 15.0
1.0 1.0 2.0 3.0 1.0 5.0 6.0 7.0 1.0 9.0 10.0 11.0 1.0 13.0 14.0 15.0
1.0 1.0 2.0 3.0 1.0 5.0 6.0 7.0 1.0 9.0 10.0 11.0 1.0 13.0 14.0 15.0

print_buffer_bf16 rows=8 colsb=16
1.0 1.0 2.0 3.0 1.0 5.0 6.0 7.0 1.0 9.0 10.0 11.0 1.0 13.0 14.0 15.0
1.0 1.0 2.0 3.0 1.0 5.0 6.0 7.0 1.0 9.0 10.0 11.0 1.0 13.0 14.0 15.0
1.0 1.0 2.0 3.0 1.0 5.0 6.0 7.0 1.0 9.0 10.0 11.0 1.0 13.0 14.0 15.0
1.0 1.0 2.0 3.0 1.0 5.0 6.0 7.0 1.0 9.0 10.0 11.0 1.0 13.0 14.0 15.0
1.0 1.0 2.0 3.0 1.0 5.0 6.0 7.0 1.0 9.0 10.0 11.0 1.0 13.0 14.0 15.0
1.0 1.0 2.0 3.0 1.0 5.0 6.0 7.0 1.0 9.0 10.0 11.0 1.0 13.0 14.0 15.0
1.0 1.0 2.0 3.0 1.0 5.0 6.0 7.0 1.0 9.0 10.0 11.0 1.0 13.0 14.0 15.0
1.0 1.0 2.0 3.0 1.0 5.0 6.0 7.0 1.0 9.0 10.0 11.0 1.0 13.0 14.0 15.0

print_buffer32 rows=8 colsb=8
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
[...]
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0

_tile_dpbssd Elapsed Time (cycles): 62
```

```
print_buffer32 rows=8 colsb=8
100.0 264.0 356.0 664.0 612.0 1064.0 868.0 1464.0
100.0 264.0 356.0 664.0 612.0 1064.0 868.0 1464.0
100.0 264.0 356.0 664.0 612.0 1064.0 868.0 1464.0
100.0 264.0 356.0 664.0 612.0 1064.0 868.0 1464.0
100.0 264.0 356.0 664.0 612.0 1064.0 868.0 1464.0
100.0 264.0 356.0 664.0 612.0 1064.0 868.0 1464.0
100.0 264.0 356.0 664.0 612.0 1064.0 868.0 1464.0
100.0 264.0 356.0 664.0 612.0 1064.0 868.0 1464.0
```

※ 예를 들어 dst[m=0][n=7] =1464 는 다음과 같이 계산된다.

m=0,n=7	A.row[0] k=0,...,7			
	bf16[0] bf16[1]	bf16[2] bf16[3]	...	bf16[14] bf16[15]
bf16	1.0 1.0	2.0 3.0	...	14.0 15.0
	B.row[k] k=0,...,7			
	bf16[14] bf16[15]	bf16[14] bf16[15]	...	bf16[14] bf16[15]
bf16	14.0 15.0	14.0 15.0	...	14.0 15.0
dot product	14+15	28+45		196+225
total sum	1464			

4. 최적화

- 컴파일러에 의한 코드 최적화를 잘 이용하면 더 빠르고 효율적인 실행가능한 프로그램을 만들 수 있음. GCC에서는 -O 옵션에 의해 최적화를 수행하며 최적화 수준을 다르게 컴파일 할 수 있음. 최적화 수준이 높을 수록 컴파일 시간이 늘어남.

최적화 옵션	설명
-O0	- 최적화를 수행하지 않음 (default) - 소스코드의 각 명령은 재배열 없이 직접 변환된다. 디버깅 단계에서 사용
-O1	- 일반적인 기본적인 최적화를 수행함
-O2	- 명령 스케줄링을 포함한 추가적인 최적화를 적용. 속도와 공간의 균형이 필요하지 않은 최적화만 사용되므로, 실행파일의 크기는 늘어나지 않음
-O3	- 실행속도를 높일수 있지만, 크기도 늘어날 수 음.
-Os	- 실행파일의 크기를 줄이는 최적화를 선택, 메모리나 디스크 공간이 제한된 임베디드 시스템에 적합

[표 4] 컴파일러 최적화 옵션

최적화 수준	설명
고수준	- 루프 변환 (Loop Transformations): 루프 언롤링, 루프 분할, 루프 합병, 루프 interchange 등. - 데이터 흐름 분석 (Data Flow Analysis): 변수의 생명 주기 분석, 사용되지 않는 코드 제거. - 함수 인라인화 (Function Inlining): 자주 호출되는 작은 함수를 호출하지 않고 직접 코드에 삽입. - 상수 전파 (Constant Propagation): 상수를 사용하는 표현식을 상수 자체로 대체.
중간 수준	- 배열 범위 검사 제거 (Bounds Checking Elimination): 배열 인덱스 범위를 미리 확인하여 런타임 검사 제거. - 공통 서브 표현식 제거 (Common Subexpression Elimination): 반복되는 계산을 제거. - 복제 및 이동 (Code Motion and Cloning): 루프 외부로 코드 이동, 복제하여 실행 시간 단축. - 코드 블록 병합 (Basic Block Merging): 동일한 코드 블록을 병합하여 중복 제거.
저수준	- 레지스터 할당 (Register Allocation): 변수와 중간 계산 결과를 레지스터에 저장하여 메모리 접근 최소화. - 명령어 선택 및 스케줄링 (Instruction Selection and Scheduling): 타겟 아키텍처의 효율적인 명령어 사용. - 파이프라인 최적화 (Pipeline Optimization): CPU 파이프라인의 병목을 최소화. - 벡터화 (Vectorization): SIMD 명령어를 사용하여 병렬 처리.

- 아래 예제를 이용하여 최적화 기능을 살펴본다.

```
// vec1.c
#include <stdio.h>
#include <stdlib.h>
#include "rdtsc.h"
#ifndef N
```

```
#define N 200
#endif

void init_buffer(float *vec, int n) {
  for (int i = 0; i < n; ++i) {
    vec[i] = 1.0f;
  }
}
void buffer_print(float *buf, int n) {
  float v;
  printf("buffer_print n=%d \n", n);
  for (int i = 0; i < n; i++)  {
    v = buf[i];
    printf("%3.1f ", v);
  }
  printf("\n");
}
int main() {
  float a[N], b[N], c[N], result[N];
  init_buffer(a, N);
  init_buffer(b, N);
  init_buffer(c, N);

  uint64_t start, end;
  long long unsigned int elapsed;
  start = rdtsc();
  for (int j = 0; j < N; ++j) {
    result[j] = a[j] + b[j] * c[j];
  }
  end = rdtsc();
  elapsed = end - start;
  printf("loop Elapsed Time (cycles): %llu\n", elapsed);
  buffer_print(result, N);
  return 0;
}
```

```
$ gcc -o rdtsc.o  -c rdtsc.c
$ gcc -g -static -o vec1.level0  --save-temps  vec1.c  rdtsc.o
$ ./vec1.level0
loop Elapsed Time (cycles): 1090
buffer_print n=200
2.0 2.0 2.0 2.0 2.0 [...] 2.0
```

```
// vec1-vec1.s
[...]
        call    rdtsc@PLT
        movq    %rax, -3240(%rbp)
        movl    $0, -3244(%rbp) ; for loop 시작
        jmp     .L8
.L9:
        movl    -3244(%rbp), %eax
        cltq
        movss   -3216(%rbp,%rax,4), %xmm1 ; a[j] -> xmm1
        movl    -3244(%rbp), %eax
```

```
        cltq
        movss   -2416(%rbp,%rax,4), %xmm2 ; b[j] -> xmm2
        movl    -3244(%rbp), %eax
        cltq
        movss   -1616(%rbp,%rax,4), %xmm0 ; c[j] -> xmm0
        mulss   %xmm2, %xmm0   ;  multiply --> xmm0
        addss   %xmm1, %xmm0   ;  add --> xmm0
        movl    -3244(%rbp), %eax
        cltq
        movss   %xmm0, -816(%rbp,%rax,4) ;  xmm0 --> result[j]
        addl    $1, -3244(%rbp)
.L8:
        cmpl    $199, -3244(%rbp)
        jle     .L9
        call    rdtsc@PLT
[...]
```

```
; 최적화 옵션 -O1
$ gcc  -O1  -static -o vec1.level1  --save-temps  vec1.c  rdtsc.o
$ less vec1-vec1.s
[...]
        call    rdtsc@PLT
        movq    %rax, %rbx
        movl    $0, %eax
        leaq    1600(%rsp), %rcx
        movq    %rsp, %rdx
.L11:
        movss   800(%rsp,%rax), %xmm0
        mulss   (%rax,%rcx), %xmm0  ; 메모리주소 (rax+rcx)의 값을 xmm0에 곱함
        addss   (%rax,%rdx), %xmm0  ; 메모리주소 (rax+rdx)의 값을 xmm0에 더함
        movss   %xmm0, 2400(%rsp,%rax)
        addq    $4, %rax
        cmpq    $800, %rax
        jne     .L11
        call    rdtsc@PLT
[...]
```

※ -fopt-info 옵션은 컴파일 과정에서 최적화 정보를 출력함.

```
$ gcc  -O1 -fopt-info  -static -o vec1  --save-temps  vec1.c  rdtsc.o
vec1.c:22:3: optimized:   Inlining printf/15 into buffer_print/40 (always_inline).
vec1.c:20:5: optimized:   Inlining printf/15 into buffer_print/40 (always_inline).
vec1.c:17:3: optimized:   Inlining printf/15 into buffer_print/40 (always_inline).
vec1.c:39:3: optimized:   Inlining printf/15 into main/41 (always_inline).
```

- 최적화 하지 않는 경우와 최적화 옵션 -O1 을 적용한 경우 수행시간 차이는 다음과 같다. (반복실행시 약간씩 차이가 있음)

```
$ ./vec1.level0
loop Elapsed Time (cycles): 1168
$ ./vec1.level1
loop Elapsed Time (cycles): 382
```

- 최적화 레벨2

```
$ gcc  -O2  -fopt-info   -static -o vec1.level2  --save-temps  vec1.c  rdtsc.o
vec1.c:22:3: optimized:   Inlining printf/15 into buffer_print/40 (always_inline).
vec1.c:20:5: optimized:   Inlining printf/15 into buffer_print/40 (always_inline).
vec1.c:17:3: optimized:   Inlining printf/15 into buffer_print/40 (always_inline).
vec1.c:39:3: optimized:   Inlining printf/15 into main/41 (always_inline).
vec1.c:28:3: optimized:  Inlining init_buffer/39 into main/41.
vec1.c:27:3: optimized:  Inlining init_buffer/39 into main/41.
vec1.c:26:3: optimized:  Inlining init_buffer/39 into main/41.
```

```
$ less vec1.level2-vec1.s
[...]
        call    rdtsc@PLT
        movq    %rax, %r14
        xorl    %eax, %eax
        .p2align 4,,10
        .p2align 3
.L15:
        movss   (%rbx,%rax), %xmm0
        mulss   (%r12,%rax), %xmm0
        addss   0(%r13,%rax), %xmm0
        movss   %xmm0, 0(%rbp,%rax)
        addq    $4, %rax
        cmpq    $800, %rax
        jne     .L15
        call    rdtsc@PLT
[...]
```

- 최적화 레벨3. --unroll-loops 옵션을 함께 적용

```
$ gcc  -O3 --unroll-loops  -fopt-info   -static -o vec1.level3  --save-temps  vec1.c  rdtsc.o
[...]
vec1.c:11:21: optimized: loop vectorized using 16 byte vectors
vec1.c:11:21: optimized: loop with 2 iterations completely unrolled (header execution count 645
30389)
vec1.c:12:12: optimized: loop unrolled 7 times
vec1.c:18:21: optimized: loop unrolled 7 times
vec1.c:34:21: optimized: loop vectorized using 16 byte vectors
vec1.c:11:21: optimized: loop vectorized using 16 byte vectors
vec1.c:11:21: optimized: loop vectorized using 16 byte vectors
vec1.c:11:21: optimized: loop vectorized using 16 byte vectors
vec1.c:35:15: optimized: loop unrolled 9 times
vec1.c:12:12: optimized: loop unrolled 9 times
vec1.c:12:12: optimized: loop unrolled 9 times
vec1.c:12:12: optimized: loop unrolled 9 times
```

```
$ less  vec1.level3-vec1.s
[...]
        call    rdtsc@PLT
        xorl    %esi, %esi
        movq    %rax, %r14
.L92:
        movaps  (%r12,%rsi), %xmm1
        movaps  16(%r12,%rsi), %xmm2
        mulps   (%rbx,%rsi), %xmm1   ; 10번의 mul을 같은 루프에서 수행 (xmm1~xmm8)
```

```
        movaps  32(%r12,%rsi), %xmm3
        mulps   16(%rbx,%rsi), %xmm2
        movaps  48(%r12,%rsi), %xmm4
        mulps   32(%rbx,%rsi), %xmm3
        movaps  64(%r12,%rsi), %xmm5
        mulps   48(%rbx,%rsi), %xmm4
        movaps  80(%r12,%rsi), %xmm6
        mulps   64(%rbx,%rsi), %xmm5
        movaps  96(%r12,%rsi), %xmm7
        mulps   80(%rbx,%rsi), %xmm6
        movaps  112(%r12,%rsi), %xmm8
        mulps   96(%rbx,%rsi), %xmm7
        addps   0(%r13,%rsi), %xmm1
        mulps   112(%rbx,%rsi), %xmm8
        addps   16(%r13,%rsi), %xmm2
        addps   32(%r13,%rsi), %xmm3
        addps   48(%r13,%rsi), %xmm4
        addps   64(%r13,%rsi), %xmm5
        addps   80(%r13,%rsi), %xmm6
        movaps  %xmm1, 0(%rbp,%rsi)
        addps   96(%r13,%rsi), %xmm7
        addps   112(%r13,%rsi), %xmm8
        movaps  %xmm2, 16(%rbp,%rsi)
        movaps  %xmm3, 32(%rbp,%rsi)
        movaps  %xmm4, 48(%rbp,%rsi)
        movaps  %xmm5, 64(%rbp,%rsi)
        movaps  %xmm6, 80(%rbp,%rsi)
        movaps  %xmm7, 96(%rbp,%rsi)
        movaps  %xmm8, 112(%rbp,%rsi)
        movaps  128(%r12,%rsi), %xmm9
        movaps  144(%r12,%rsi), %xmm10
        mulps   128(%rbx,%rsi), %xmm9
        mulps   144(%rbx,%rsi), %xmm10
        addps   128(%r13,%rsi), %xmm9
        addps   144(%r13,%rsi), %xmm10
        movaps  %xmm9, 128(%rbp,%rsi)
        movaps  %xmm10, 144(%rbp,%rsi)
        addq    $160, %rsi
        cmpq    $800, %rsi
        jne     .L92
        call    rdtsc@PLT
[...]
```

- 루프 언롤링 하지 않기. 컴파일러 지시어를 사용하여 특정 루프에 대한 언롤링을 제한할 수 있음

```
형식 : #pragma GCC unroll N
   루프를 N번 언롤링하도록 컴파일러에게 지시
```

```
// vec2.c  (vec1.c 에서 수정)
[...]
int main() {
[...]
  start = rdtsc();
  #pragma GCC unroll 0     // 루프 언롤링을 비활성화
```

```
  for (int j = 0; j < N; ++j) {
    result[j] = a[j] + b[j] * c[j];
  }
[...]
$ gcc  -O3 --unroll-loops  -o vec2.level3  --save-temps  vec2.c  rdtsc.o
$ less vec2.level3-vec2.s
[...]
```

[참고자료]

- Intel Intrinsics Guide https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html
- Intel AVX512-FP16 Architecture Specification, June 2021, Rev 1.0
- Intel oneAPI DPC++/C++ Compiler Documentation https://www.intel.com/content/www/us/en/developer/tools/oneapi/dpc-compiler-documentation.html
- Code Sample: Intel AMX-Intrinsics Functions https://www.intel.com/content/www/us/en/developer/articles/code-sample/advanced-matrix-extensions-intrinsics-functions.html
- Code Sample: Deep Learning Boost Instruction bflat16 Intrinsic Functions https://www.intel.com/content/www/us/en/developer/articles/technical/intel-deep-learning-boost-new-instruction-bfloat16.html
- Intel Architecture Instruction Set Extensions and Future Features https://www.intel.com/content/www/us/en/content-details/819680/intel-architecture-instruction-set-extensions-programming-reference.html
- 모던 C++, SIMD 병렬화 https://www.youtube.com/watch?v=EQciEvjcAr0
- IEEE-754 Flating Point Converter https://www.h-schmidt.net/FloatConverter/IEEE754.html
- SIMD for C++ Developers, 2021 http://const.me/articles/simd/simd.pdf
- GNU C inline asm input constraint for AVX512 mask registers https://stackoverflow.com/questions/55946103/gnu-c-inline-asm-input-constraint-for-avx512-mask-registers-k1-k7
- What is SSE and AVX? https://www.codingame.com/playgrounds/283/sse-avx-vectorization/what-is-sse-and-avx
- Options That Control Optimization https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html
- x86 Assembly Language Reference Manual, ORACLE, 2018 https://docs.oracle.com/cd/E53394_01/pdf/E54851.pdf

이 보고서는 2024년도 한국과학기술정보연구원(KISTI)의
기본사업으로 수행된 연구입니다.
과제번호: (KISTI) K24-L02-C06-S01
과제명: 초고성능컴퓨팅 공동활용을 위한 통합 환경 개발 및 구축

비매품/무료
95560
9 788929 416959
ISBN 978-89-294-1695-9 (PDF)