Vector Processing

SSE/AVX & NEON

Jeong-Gun Lee

Dept. of Computer Engineering, Hallym University

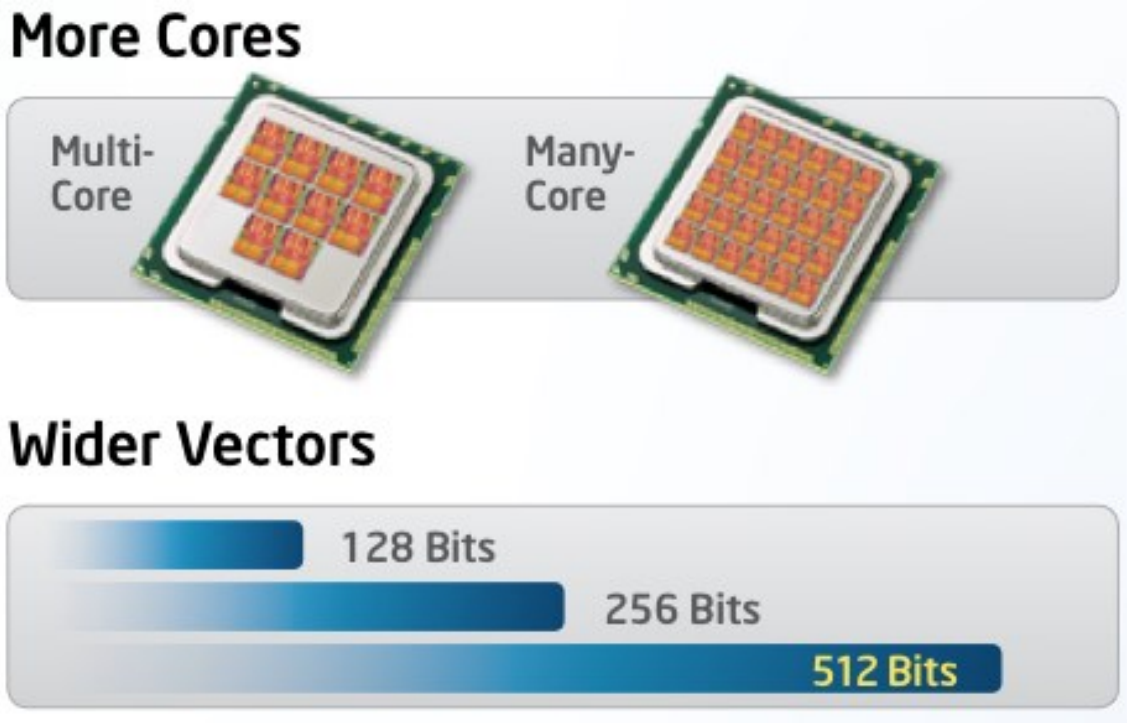
Email: Jeonggun.Lee@hallym.ac.kr

Outline

- Scalar registers vs. vector registers
  - Intel AVX vector extensions
- Vectorization (SIMDization) of scalar code
- Selected AVX vector instructions

Terms

- SIMD: Single Instruction Multiple Data
- MMX: MultiMedia eXtensions
- SSE: Streaming SIMD Extensions
- AVX: Advanced Vector eXtensions (From Sandy Bridge)



SIMD Vectorization

- To sum the values of 2 arrays, a conventional CPU needs one add operation ("+") per array index:

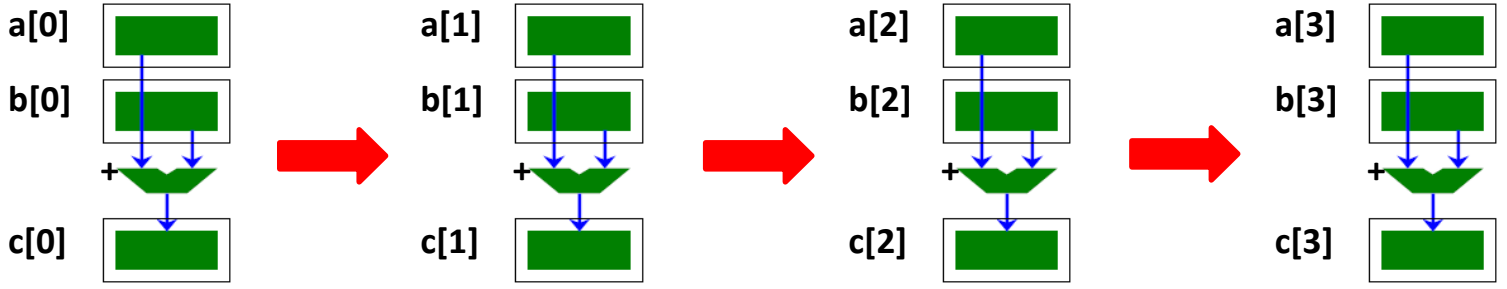
```
double a[4] = {1.0, 2.0, 3.0, 4.0};
double b[4] = {1.0, 2.0, 3.0, 4.0};
double c[4];

c[0] = a[0] + b[0];
c[1] = a[1] + b[1];
c[2] = a[2] + b[2];
c[3] = a[3] + b[3];
```

sequential array sum

```
double a[4] = {1.0, 2.0, 3.0, 4.0};
double b[4] = {1.0, 2.0, 3.0, 4.0};
double c[4];
int i;

for(i=0; i < 4; i++) {
   c[i] = a[i] + b[i];
}
```

array sum using loop

- Reason: a register of a conventional CPU can only hold only 1 data item at a time (such a register is called a **scalar register**):



SIMD Vectorization (cont.)

- Vector processors have *large registers that can **hold multiple values** of the **same data type***.

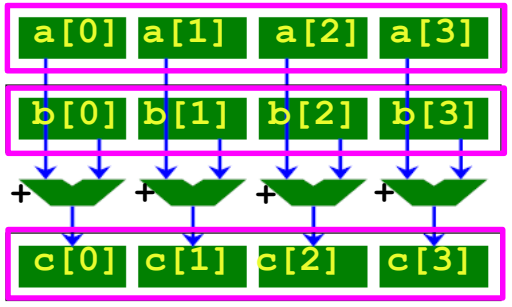
```
double a[4] = {1.0, 2.0, 3.0, 4.0};
double b[4] = {1.0, 2.0, 3.0, 4.0};
double c[4];

c[0] = a[0] + b[0];
c[1] = a[1] + b[1];
c[2] = a[2] + b[2];
c[3] = a[3] + b[3];
```

sequential array sum

```
__m256d a = {1.0, 2.0, 3.0, 4.0};
__m256d b = {1.0, 2.0, 3.0, 4.0};
__m256d c;

c = _mm256_add_pd(a,b);
```

data-parallel array sum using vectors

- An Intel AVX register can hold ***4 double values*** at once:
- Called a **vector** of 4 doubles
  - every **double** value is a **vector element**
- **_mm256_add_pd()** operates on vectors
  - In one go, individual elements from vectors a and b are added and stored in vector c (**data parallelism!**)



SIMD Vectorization (cont.)

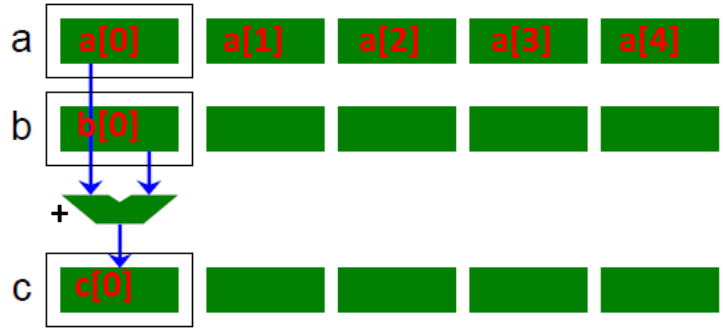
- Instead of arrays of integers, we can define arrays of vectors of integers:

```
double a[4*MAX], b[4*MAX];
double c[4*MAX];
int ctr;

for (ctr = 0; ctr < 4*MAX; ctr++) {
    c[ctr] = a[ctr] + b[ctr];
}
```

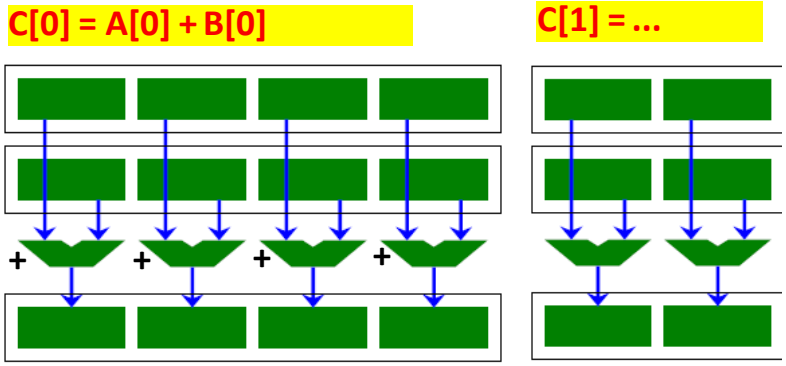
scalar code

```
#define MAX ...
__m256d A[MAX], B[MAX], C[MAX];
int ctr;

for (ctr = 0; ctr < MAX; ctr++) {
   C[ctr] = _mm256_add_pd(A[ctr], B[ctr];
}
```

vector code



scalar register

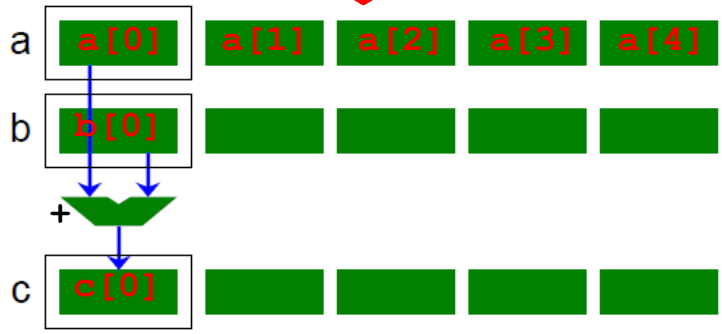


vector register

SIMD Vectorization (cont.)

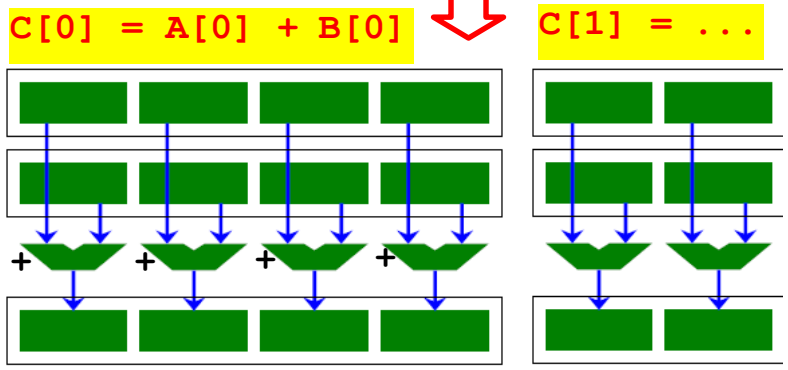
```
double a[4*MAX], b[4*MAX];
double c[4*MAX];
int ctr;

for (ctr = 0; ctr < 4*MAX; ctr++) {
    c[ctr] = a[ctr] + b[ctr];
}
```

scalar code




```
#define MAX ...
__m256d A[MAX], B[MAX], C[MAX];
int ctr;

for (ctr = 0; ctr < MAX; ctr++) {
   C[ctr] = _mm256_add_pd(A[ctr], B[ctr];
}
```

vector code



- Vector code uses single instruction, but multiple data items (vector elements)
  - called Single Instruction Multiple Data (SIMD)
- Scalar code operates on 1 data item per operation
  - called Single Instruction Single Data (SISD)

SIMD Vectorization (cont.)

```
double a[4*MAX], b[4*MAX];
double c[4*MAX];
int ctr;

for (ctr = 0; ctr < 4*MAX; ctr++) {
    c[ctr] = a[ctr] + b[ctr];
}
```

```
#define MAX ...
__m256d A[MAX], B[MAX], C[MAX];
int ctr;

for (ctr = 0; ctr < MAX; ctr++) {
   C[ctr] = _mm256_add_pd(A[ctr], B[ctr];
}
```

With “MAX 5000000”

```
jeong-gun@lana:~/VECTOR/TEST$ time ./seq1

real    1m4.615s
user    1m4.492s
sys     0m0.128s
```

64 sec

```
jeong-gun@lana:~/VECTOR/TEST$ time ./vec1

real    0m43.476s
user    0m41.748s
sys     0m1.728s
```

43 sec

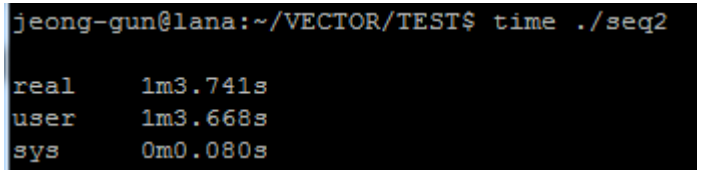
~ 1.48 Speedup < 4

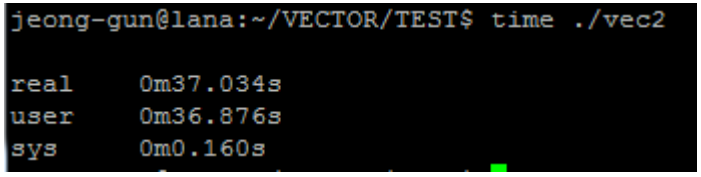
SIMD Vectorization (cont.)

```
float a[4*MAX], b[4*MAX];
float c[4*MAX];
int ctr;

for (ctr = 0; ctr < 4*MAX; ctr++) {
    c[ctr] = a[ctr] + b[ctr];
}
```

```
#define MAX ...
__m256 A[MAX], B[MAX], C[MAX];
int ctr;

for (ctr = 0; ctr < MAX; ctr++) {
   C[ctr] = _mm256_add_ps(A[ctr], B[ctr];
}
```

With “MAX 5000000”





63 sec

36 sec

~ 1.75 Speedup < 8

Lab

```
// vec1.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include <immintrin.h>
#include <time.h>

#define MAX 5000000

/*
__m256d va[MAX];
__m256d vb[MAX];

void main() {
int ctr, i;

clock_t startTime = clock();

for (ctr = 0; ctr<1000; ctr++) {
for (i = 0; i<MAX; i++) {
va[i] = _mm256_add_pd(va[i], vb[i]);
}
}
printf("Elapase Time : %d\n", clock() - startTime);
}
*/

double va[4 * MAX];
double vb[4 * MAX];

void main() {
int i, ctr;

clock_t startTime = clock();

for (ctr = 0; ctr<1000; ctr++) {
for (i = 0; i<4 * MAX; i++) {
va[i] = va[i] + vb[i];
}
}
printf("Elapase Time : %d\n", clock() - startTime);

}
```

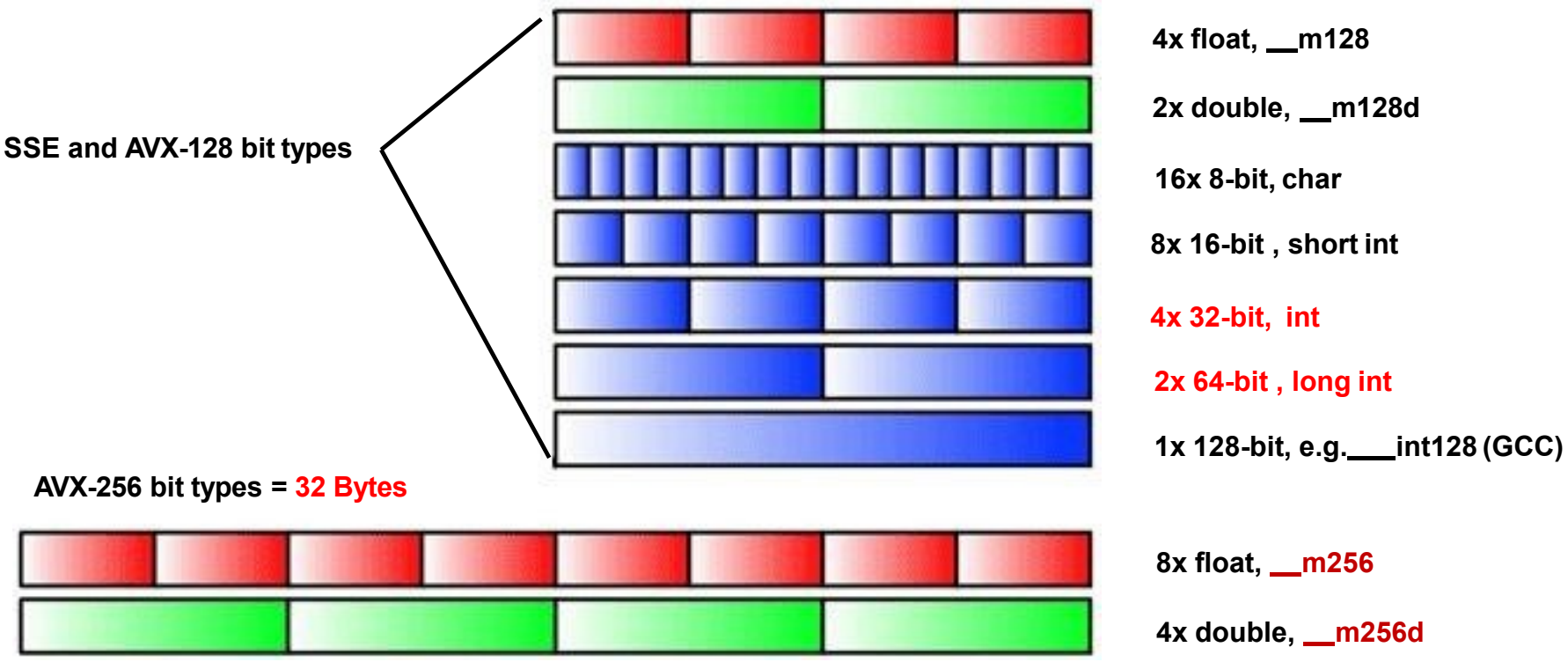
Branch: master ▾ GeomexSoft / 02_simd_lab / simple_test /

jeonggunlee committed on GitHub Rename vec2.c to vec2_float.c

..	
scalar.c	Add files via upload
seq1_double.c	Rename seq1.c to seq1_double.c
seq2.s	Add files via upload
seq2_float.c	Rename seq2float.c to seq2_float.c
test.txt	Add files via upload
vec1_double.c	Rename vec1.c to vec1_double.c
vec2.s	Add files via upload
vec2_float.c	Rename vec2.c to vec2_float.c
vector.c	Add files via upload
vector.s	Add files via upload
vector_unroll.c	Add files via upload

Different AVX and SSE vector types

- AVX was introduced with the Intel **Sandy Bridge** architecture (2011)
- Vector registers even span to **512** and might be further extended to **1024 bits**
- In our discussion, we won't cover previous (128-bit SSE) types



Vector Processing Instructions

```
jeong-gun@lana: ~/VECTOR/TEST
jeong-gun@lana:~/VECTOR/TEST$ gcc seq2.c -S
jeong-gun@lana:~/VECTOR/TEST$ gcc vec2.c -S -mavx
```

jeong-gun@lana: ~/VECTOR/TEST

```
        .file   "seq2.c"
        .comm   va,80000000,32
        .comm   vb,80000000,32
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $0, -4(%rbp)
        jmp     .L2
.L5:
        movl    $0, -8(%rbp)
        jmp     .L3
.L4:
        movl    -8(%rbp), %eax
        cltq
        movss   va(,%rax,4), %xmm1
        movl    -8(%rbp), %eax
        cltq
        movss   vb(,%rax,4), %xmm0
        addss   %xmm1, %xmm0
        movl    -8(%rbp), %eax
        cltq
        movss   %xmm0, va(,%rax,4)
        addl    $1, -8(%rbp)
.L3:
        cmpl    $19999999, -8(%rbp)
        jle     .L4
        addl    $1, -4(%rbp)
.L2:
        cmpl    $999, -4(%rbp)
        jle     .L5
        nop
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.4) 5.4.0 20160609"
        .section        .note.GNU-stack,"",@progbits
~
~
"seq2.s" 47 lines, 770 characters
```

jeong-gun@lana: ~/VECTOR/TEST

```
        .file   "vec2.c"
        .comm   va,160000000,32
        .comm   vb,160000000,32
        .text
        .globl  main
        .type   main, @function
main:
.LFB3448:
        .cfi_startproc
        leaq    8(%rsp), %r10
        .cfi_def_cfa 10, 0
        andq    $-32, %rsp
        pushq   -8(%r10)
        pushq   %rbp
        .cfi_escape 0x10,0x6,0x2,0x76,0
        movq    %rsp, %rbp
        pushq   %r10
        .cfi_escape 0xf,0x3,0x76,0x78,0x6
        movl    $0, -88(%rbp)
        jmp     .L2
.L6:
        movl    $0, -84(%rbp)
        jmp     .L3
.L5:
        movl    -84(%rbp), %eax
        cltq
        salq    $5, %rax
        addq    $vb, %rax
        movl    -84(%rbp), %eax
        cltq
        salq    $5, %rax
        addq    $va, %rax
        vmovaps (%rax), %ymm1
        vmovaps %ymm1, -80(%rbp)
        vmovaps %ymm0, -48(%rbp)
        vmovaps -80(%rbp), %ymm0
        vaddps  -48(%rbp), %ymm0, %ymm0
        movl    -84(%rbp), %eax
        cltq
        vmovaps %ymm0, (%rax)
        addl    $1, -84(%rbp)
.L3:
        cmpl    $4999999, -84(%rbp)
        jle     .L5
        addl    $1, -88(%rbp)
.L2:
"vec2.s" 63 lines, 1096 characters
```

Vector Processing Instructions

```
jeong-gun@lana: ~/VECTOR/TEST
jeong-gun@lana:~/VECTOR/TEST$ gcc seq2.c -S
jeong-gun@lana:~/VECTOR/TEST$ gcc vec2.c -S -mavx
jeong-gun@lana:~/VECTOR/TEST$
```

```
jeong-gun@lana: ~/VECTOR/TEST
        .file   "seq2.c"
        .comm   va,80000000,32
        .comm   vb,80000000,32
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $0, -4(%rbp)
        jmp     .L2
.L5:
        movl    $0, -8(%rbp)
        jmp     .L3
.L4:
        movl    -8(%rbp), %eax
        cltq
        movss   va(,%rax,4), %xmm1
        movl    -8(%rbp), %eax
        cltq
        movss   vb(,%rax,4), %xmm0
        addss   %xmm1, %xmm0
        movl    -8(%rbp), %eax
        cltq
        movss   %xmm0, va(,%rax,4)
        addl    $1, -8(%rbp)
.L3:
        cmpl    $19999999, -8(%rbp)
        jle     .L4
        addl    $1, -4(%rbp)
.L2:
        cmpl    $999, -4(%rbp)
        jle     .L5
        nop
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.4) 5.4.0 20160609"
        .section        .note.GNU-stack,"",@progbits
~
~
"seq2.s" 47 lines, 770 characters
```

ADDSS

Add Scalar Single-Precision Floating-Point Values

Opcode	Mnemonic	Description
F3 0F 58 /r	ADDSS xmm1, xmm2/m32	Add the low single-precision floating-point value from xmm2/m32 to xmm1.

Description

Adds the low single-precision floating-point values from the source operand (second operand) and the destination operand (first operand), and stores the single-precision floating-point result in the destination operand. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register. The three high-order doublewords of the destination operand remain unchanged.

Operation

```
Destination[0..31] = Destination[0..31] + Source[0..31];
//Destination[32..127] remain unchanged
```

Vector Processing Instructions

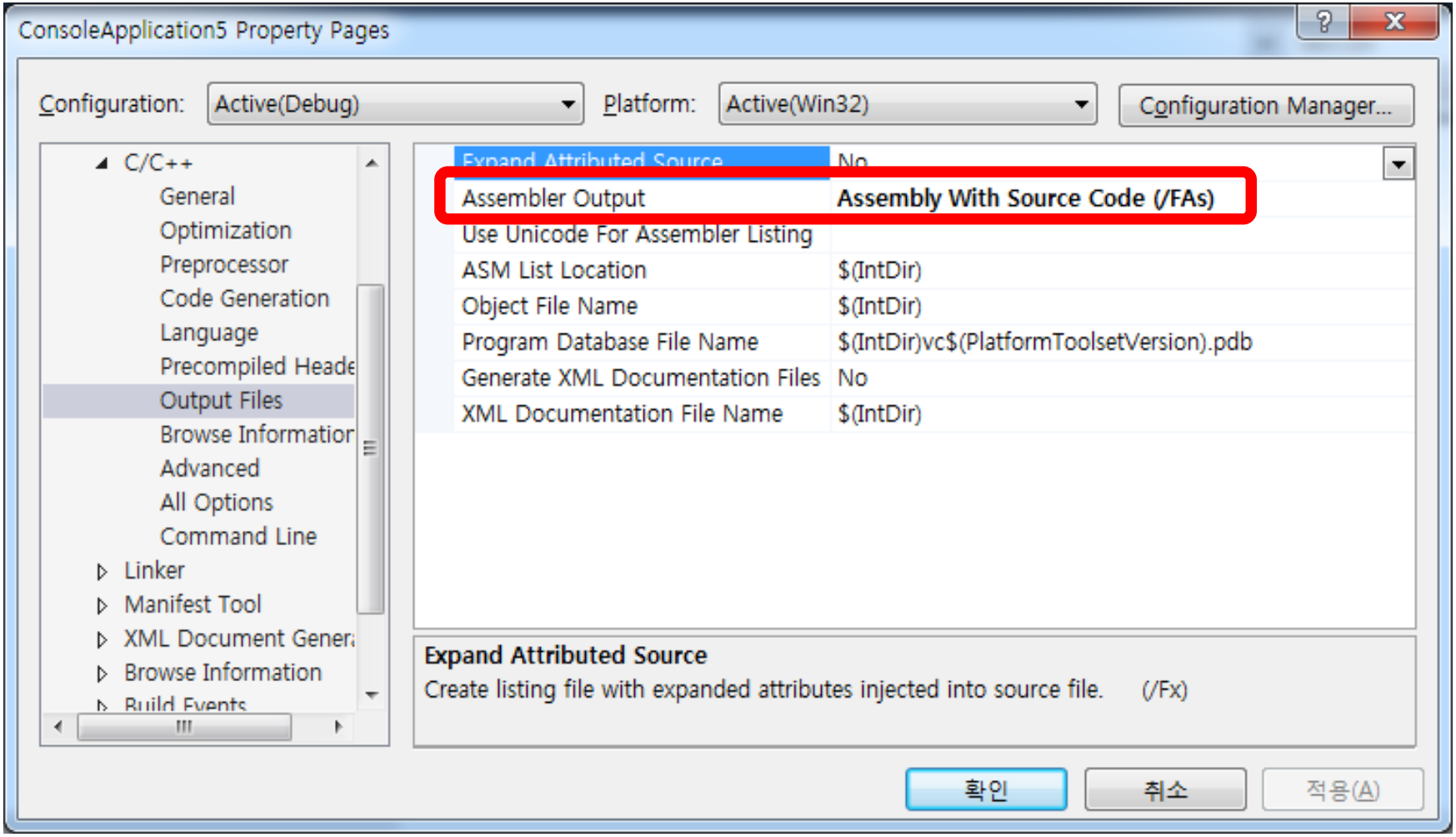
```
jeong-gun@lana: ~/VECTOR/TEST
jeong-gun@lana:~/VECTOR/TEST$ gcc seq2.c -S
jeong-gun@lana:~/VECTOR/TEST$ gcc vec2.c -S -mavx
jeong-gun@lana:~/VECTOR/TEST
```

ADDPS—Add Packed Single-Precision Floating-Point Values

Opcode/Instruction	Op/En	64/32-bit Mode	CPUID Feature Flag	Description
0F 58 /r ADDPS *xmm1, xmm2/m128*	RM	V/V	SSE	Add packed single-precision floating-point values from *xmm2/m128* to *xmm1* and stores result in xmm1.
VEX.NDS.128.0F.WIG 58 /r VADDPS xmm1,xmm2, xmm3/m128	RVM	V/V	AVX	Add packed single-precision floating-point values from xmm3/mem to xmm2 and stores result in xmm1.
VEX.NDS.256.0F.WIG 58 /r VADDPS ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Add packed single-precision floating-point values from ymm3/mem to ymm2 and stores result in ymm1.

```
jeong-gun@lana: ~/VECTOR/TEST
        .file   "vec2.c"
        .comm   va,160000000,32
        addq    $vb, %rax
        movl    -84(%rbp), %eax
        cltq
        salq    $5, %rax
        addq    $va, %rax
        vmovaps (%rax), %ymm1
        vmovaps %ymm1, -80(%rbp)
        vmovaps %ymm0, -48(%rbp)
        vmovaps -80(%rbp), %ymm0
        vaddps  -48(%rbp), %ymm0, %ymm0
        movl    -84(%rbp), %eax
        cltq
        vmovaps %ymm0, (%rax)
        addl    $1, -84(%rbp)
.L3:
        cmpl    $4999999, -84(%rbp)
        jle     .L5
        addl    $1, -88(%rbp)
.L2:
"vec2.s" 63 lines, 1096 characters
```

Vector Processing Instructions



Different AVX and SSE vector types (cont.)

- We can have vectors of 8 floats:

```
__m256 a, b, c;
c = _mm256_add_ps(a,b); //add 8 float values in one instruction!
```

- Or vecto... of doubles:

```
__m256d a, b, c;
c = _mm256_add_pd(a,b); //add 4 double values in one instruction!
```

- _mm256_add_pd() is called an **intrinsic**. An intrinsic is a procedure provided by the compiler. The compiler will replace the intrinsic with one or more CPU assembly instructions.
  - Note: here the suffix 'pd' denotes the type of the operand(s), see next slides.

AVX Intrinsic Naming Convention

```
_mm256_op_suffix (data_type param1, data_type param2, data_type param3)
```

AVX 256 bit register prefix

operation (add, sub, ...)

type of operand to operate on
first letter is 'p' (packed, meaning 'vector') or 's' (scalar) types according to table below

Suffix	Meaning
s/d	single/double precision float value
ps/pd/sd	packed single, packed double or scalar double
epi32	extended packed 32-bit signed integer
si256	scalar 256-bit integer

Note: 'single' means 'single precision', i.e., the 'float' type, 32-bits wide 'double' mea

ns 'double precision', i.e., the 'double' type, 64-bits wide

AVX Intrinsic Naming Convention (cont.)

packed (=vector)

_mm256_mul_ps

single precision (='float')

packed (=vector)

_mm256_mul_pd

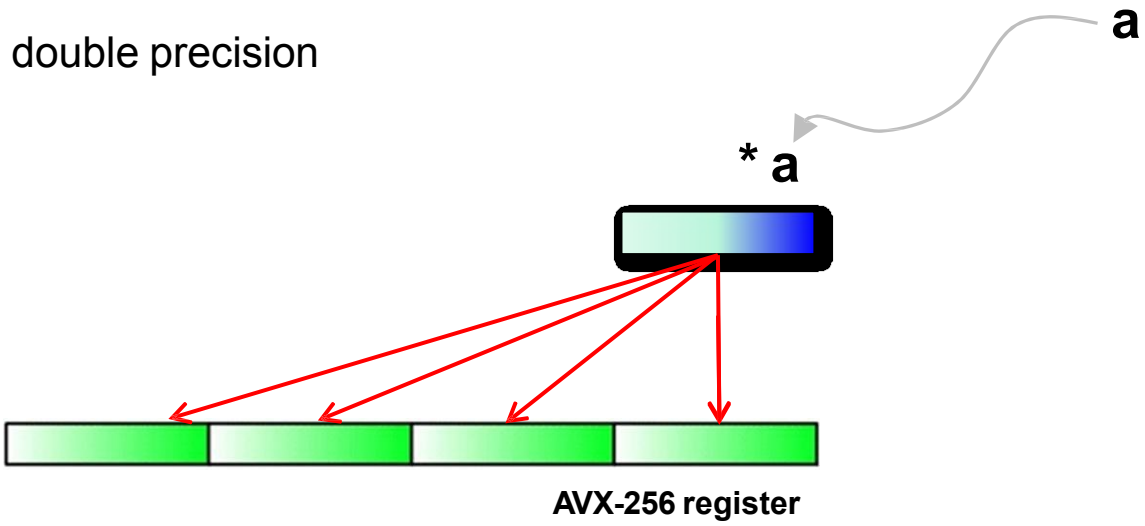
double precision ('double')

Broadcasting a scalar value across a register

scalar

```
_mm256_broadcast_sd (double const* a)
```

double precision



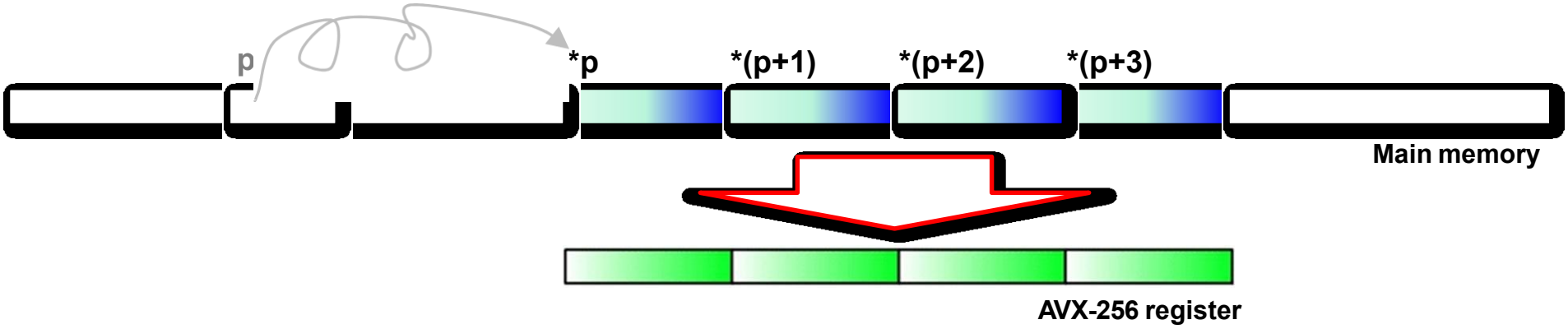
Used ***to load a scalar value from memory across all elements of vector register***, single precision version exists, too.

Loading consecutive scalars into register

packed

```
_mm256_load_pd (double const* p)
```

double precision



Used to load 4 scalar values from memory to register. Single precision version also available.

SIMD Vectorization

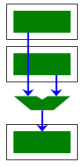
- An AVX vector register can hold 8 **float** variables at a time.
  - max. 8-fold speedup over computation in scalar registers:

```
#define MAX (5000 * 8)

float a [MAX];
float b [MAX];

void main() {
    int i, ctr;
    for (ctr=0; ctr<100; ctr++) {
        for(i=0; i<MAX; i++) {
            a[i] = a[i] + b[i];
        }
    }
}
```

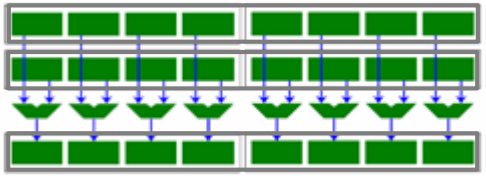
scalar sum

```
#include <immintrin.h>

#define MAX (5000)

__m256 va [MAX];
__m256 vb [MAX];

void main(void) {
   int i, ctr;
   for (ctr=0; ctr<100; ctr++) {
      for(i=0; i<MAX; i++) {
         //GCC will select intrinsics:
         va[i] = va[i] + vb[i];
      }
   }
}
```

SIMD sum



scalar register, processin g 1 pair of floats per loop iteration



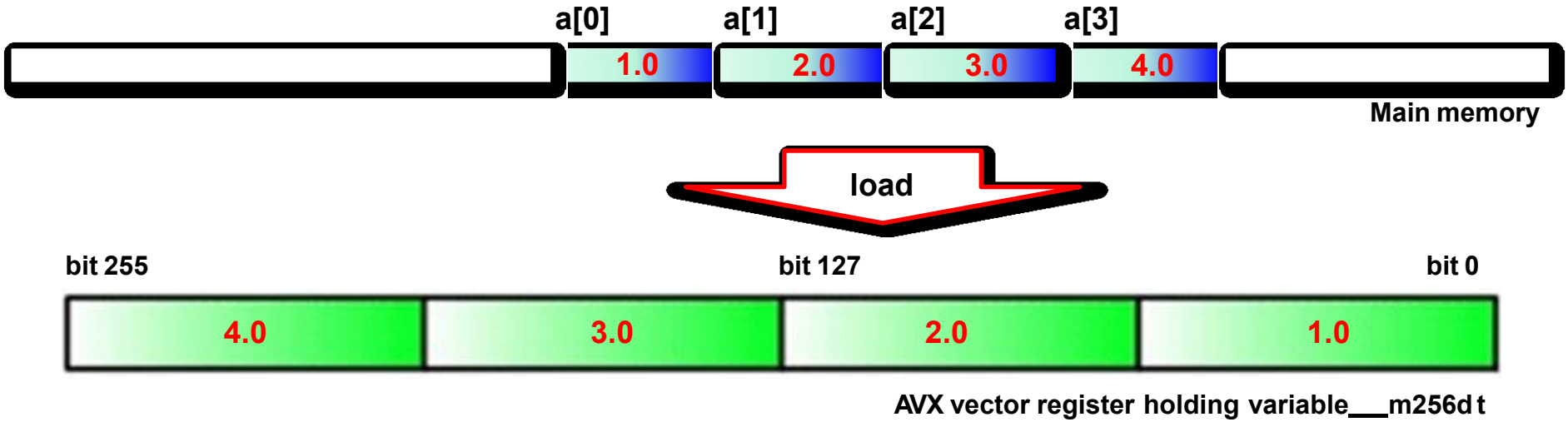
vector registers, processing 8 float pairs per loop iterati on

SIMD Vectorization (cont.)

- Try out the examples on the previous page!
- GCC >= 4.4 can generate code for AVX, if told to do so:
  - gcc **–mavx** simd_sum.c -o simd_sum
- Note: only Intel Sandybridge CPUs support AVX
  - on other CPUs, an '**illegal instruction**' exception will occur

Memory and Register Layout

```
double a[4] = { 1.0,  2.0,  3.0,  4.0 };
__m256d t = _mm256_load_pd (a);
__m256d t = _mm256_set_pd(4.0, 3.0, 2.0, 1.0);
```

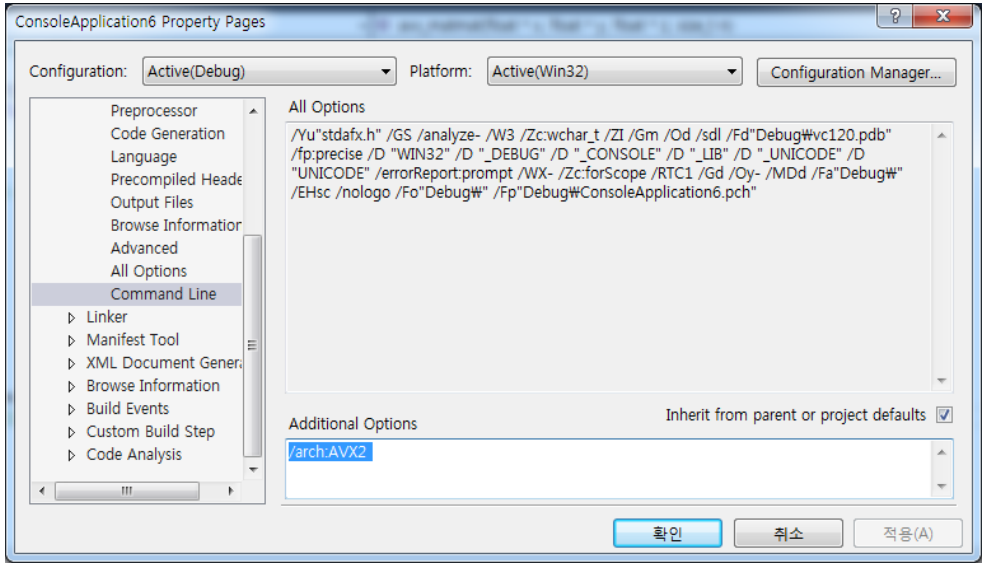


Note: The packed values inside of the AVX register are represented in right-to-left order! Same as using an ____mm256_set_pd intrinsic.

AVX2 – FMA !

- -mavx –mfma in Linux

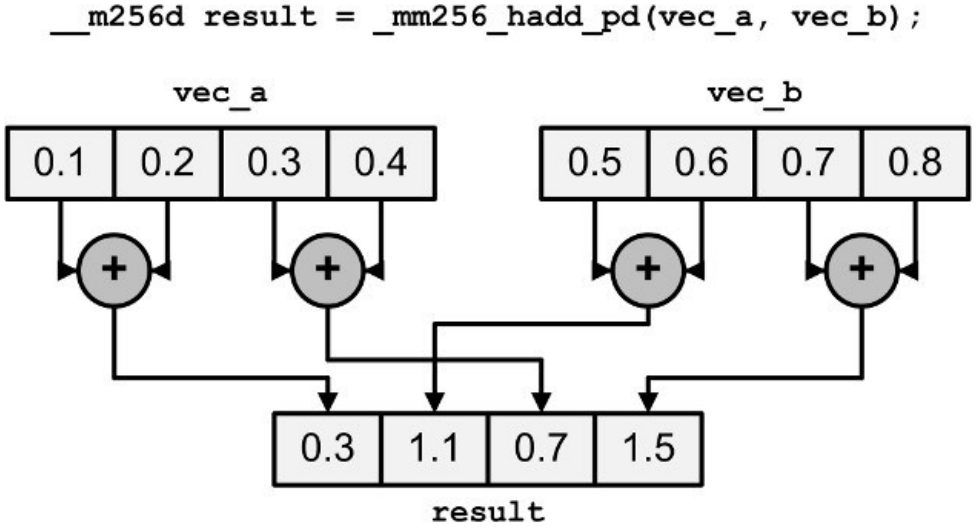
```
for(size_t k = 0; k < n; k+=8){
        a = _mm256_load_ps(x+n*i+k);
        b = _mm256_load_ps(y_temp+k);
        acc = _mm256_add_ps(acc, _mm256_mul_ps(a,b));
        //acc = _mm256_fmadd_ps(a, b, acc);
            …
}
```



Assignment !

- Assignment
  - **Vector Addition with AVX**
  - **Matrix-Vector Multiplication with AVX**
  - **Matrix-Matrix Multiplication with AVX**

- _mm256_setzero_ps()
- _mm256_load_ps
- _mm256_set_ps
- _mm256_add_
- _mm256_mul_ps
- _mm256_hadd_ps
- _mm256_store_ps



Assignment !

- One more thing to remember
  - Memory Alignment for AVX operations
  - The memory for AVX load/store should be **ALIGNED** carefully with 32 bytes width

align

- Directs the compiler to align the variable to a specified boundary

Windows* OS:
__declspec(align(*n*))

Linux* OS:
__attribute__((aligned(*n*)))
__attribute__((align(*n*)))

```
__declspec(align(32)) float vecCon[8];
```

Assignment !

- One more thing to remember
  - Memory Alignment for AVX operations
  - The memory for AVX load/store should be **ALIGNED** carefully with 32 bytes width


```
size_t n = 1024;
// AVX requires 32 bytes (256 bit) aligned memory
float* x = (float*)_mm_malloc(n*n*sizeof(float),32);
float* y = (float*)_mm_malloc(n*n*sizeof(float),32);
float* z = (float*)_mm_malloc(n*n*sizeof(float),32);
```

```
// Don't forget to deallocate memory !
_mm_free(x)
_mm_free(y)
_mm_free(z)
```

Further information

- Our treatment of AVX intrinsics was not exhaustive
  - many more intrinsics exist
    - see the interactive guide at http://software.intel.com/en-us/avx/
    - see the Intel IDF presentation at http://software.intel.com/file/2915

- Vector extensions provided by other CPU architectures
  - ARM embedded processors (smartphones, tablets etc): **NEON**
    - 8-64 bit integer, single-precision floating point,
    - 64/128 bit wide
  - PowerPC (used by IBM & formerly Apple): Altivec
  - AMD: 3DNow!, AVX, SSE
  - Cell BE: Vector/SIMD multimedia extension technology

SIMD and its relation to thread-level parallelism

- To utilize the AVX-provided data-parallelism, we can
  - manually vectorize using assembly language
  - manually vectorize in C/C++ using intrinsics (what we did)
  - write scalar C/C++ code and hope for vectorization by compiler
- The data-parallelism provided by SIMD-instructions is **fine-grained**
  - on the level of single CPU instructions => instruction-level parallelism (ILP)
- Parallelism provided by threads much **coarser-grained**
  - task/data/pipeline parallelism can be expressed

Extra

- Videos
  - https://www.youtube.com/watch?v=QhC4kcXLPww
  - https://www.youtube.com/watch?v=zeJ-kce1VLw
  - https://www.youtube.com/watch?v=DXPfE2jGqg0

OK! Let's do Assignment (a.k.a. Lab)