Using Vector Instructions through Intrinsic functions

Single Instruction Stream Multiple Data Stream (SIMD)

Computer Organization and Design

https://software.intel.com/sites/landingpage/IntrinsicsGuide/

Computer Architecture

- Assembly Instructions

  mov r1, r2

  add r1, r2, r3

- Scalar vs Vector Instructions

- Extensions to support parallelism in instructions

  Intel Intrinsics

  https://software.intel.com/sites/landingpage/IntrinsicsGuide/

Motivation

- Compiler translation not optimal
- Book introduces Parallelism : Subword parallelism
- Book content lacking some details for programming exercises.
- Vector parallelism: using hardware instructions to target data-parallel execution

Two Parts

- Part 1
  - Conceptual ideas on SIMD instructions
  - Few intrinsic functions to load, store and compute

- Part 2
  - How to translate a C loop using intrinsics
  - Aligned memory allocation
  - Demonstration on Intel processors
  - Quiz, lab problems and HW assignments with solution

Parallel Assembly level Instructions

Single Instruction Multiple Data

Extensions to Assembly Instruction Set

- Streaming SIMD Extensions (SSE)
- Intel Advanced Vector Extensions (AVX)
- AVX-512

Single Instruction Multiple Data

- SIMD

  e.g., add multiple pairs of elements using one instruction

- Vectorization or vector parallelism

- Parallel Operations via an instruction and wide registers

Single Instruction Multiple Data (SIMD)

- Special Instructions that can do multiple operations in one go.
  Pairwise add example:

A	1	2	3	4
B	2	1	2	0
+	3	3	5	4

Sequential:

Register r1, r2, ..

1) add(r1, #1, #2)
2) add(r2, #2, #1)
3) add(r3, #3, #2)
4) add(r4, #4, #0)

Single Instruction Multiple Data (SIMD)

- Special Instructions that can do multiple operations in one go.

  Pairwise add example:

A	1	2	3	4
B	2	1	2	0
+	3	3	5	4

Sequential:

1) add(r1, #1,#2), 2) add(r2, #2,#1), 3) add(r3, #3,#2), 4) add(r4, #4,#0)

VS

Parallel Addition: using 1 vector add instruction (VAdd)

Load A with {1, 2, 3, 4} and Load B with {2, 1, 2, 0}

VAdd(result, A, B)

Width of data types in C

- Float = 32 bits
- Double = 64 bits

CPU Registers

- General purpose CPU registers are 32 bits or 64 bits wide

- Special Wide SIMD registers
  128, 256, 512 bits

Example: AVX supports 256 bit registers (aka YMM register)

Many YMM registers available in x86 instruction set

Each YMM register can store multiple 32-bit single-precision floating point numbers

Streaming SIMD Extensions (SSE)

- Wide Registers Intel CPU

128 bits shown here

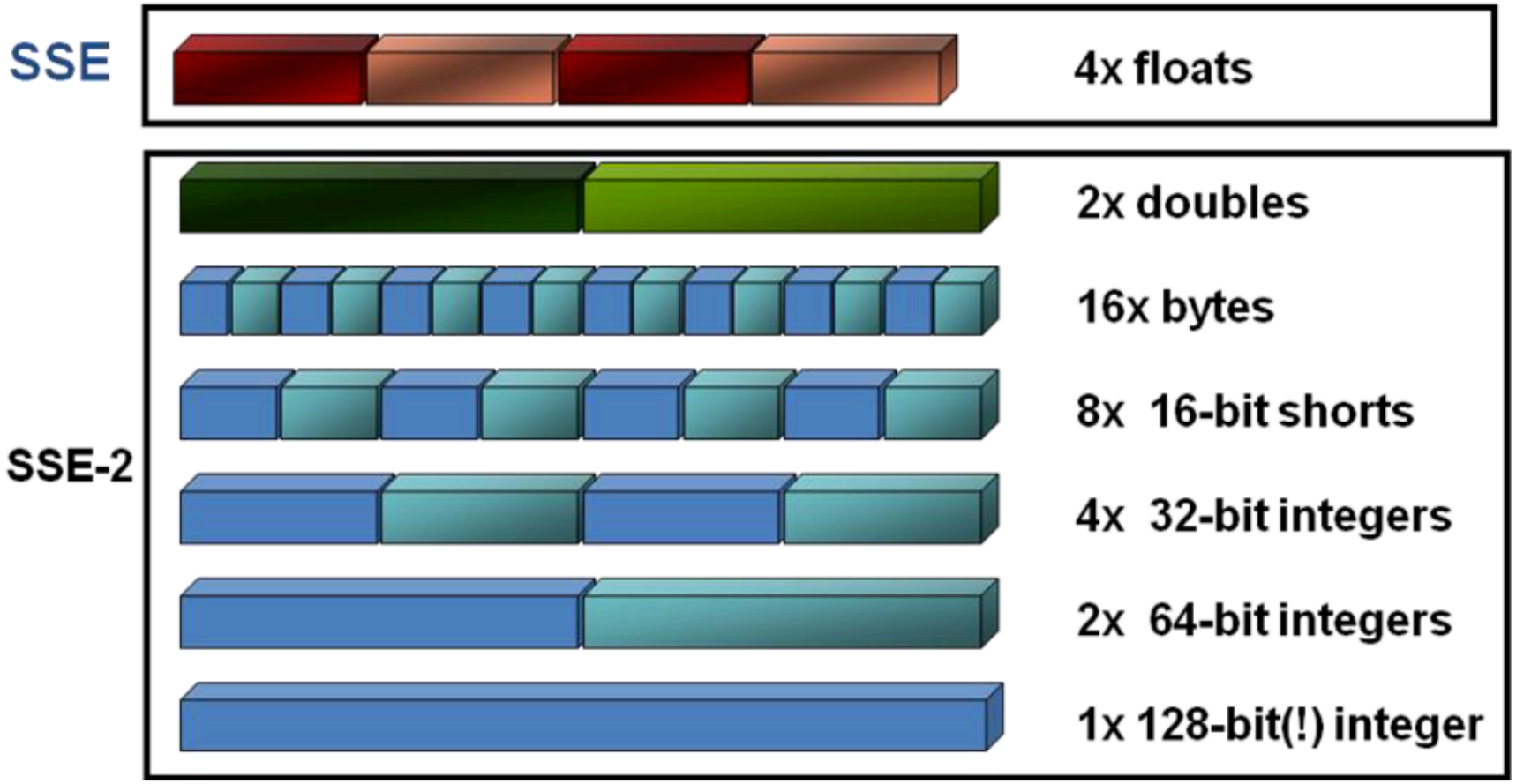
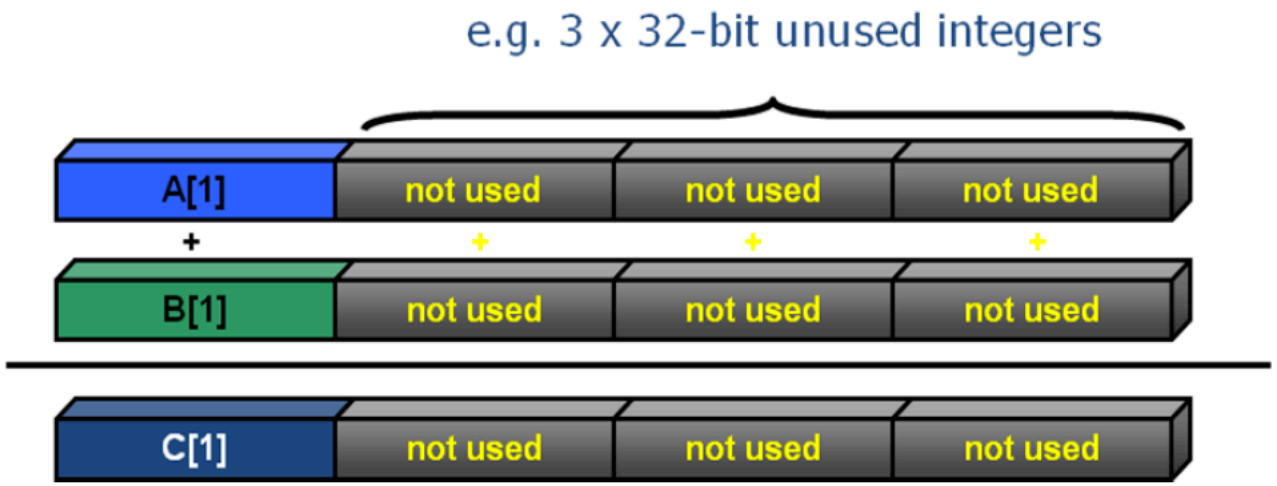


Image Source: Intel Vectorization Guide

```
for(i=0;i<=MAX;i++)
    C[i]=A[i]+B[i];
```

Without vectorization



With vectorization

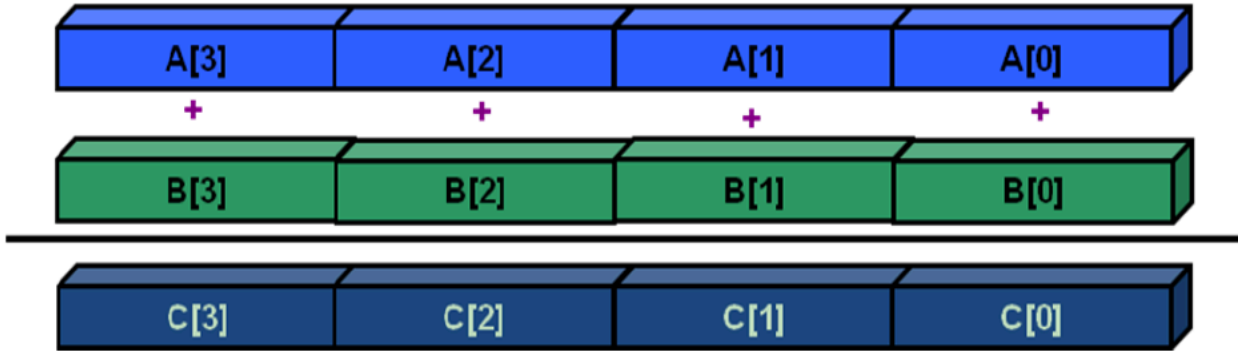


Image Source: Intel Vectorization Guide

Translating scalar code to vector code

- How to use SIMD instructions in C code?
- We will use Intel Intrinsic functions instead of assembly instructions
- Loop unrolling helps in translation

Special Assembly Instructions

- Multimedia extensions – processing pixels in an image
- ARM extensions – Neon
- X86 extensions – SSE, SSE2, AVX, etc for Intel and AMD

Special vector registers

- SIMD register

- SSE2 = xmm0, xmm1, ..

  128 bits wide

- AVX = ymm0, ymm1, ..

  256 bits wide

SIMD register Questions

In 128 bit SIMD register, how many integers, floats can be stored ?

In 128 bit SIMD register, how many double precision numbers can be stored ?

Single Instruction Multiple Data Width

- World's fastest ARM processor Fugaku by Fujitsu has a SIMD register that is 2048 bits wide.

How many double precision floating point data can be stored in one SIMD register?

How many characters can be stored in one SIMD register?

How many pixels? One pixel is 3 bytes for red, green and blue.

Hint: 1 char = 1 byte = 8 bits

1 double precision floating point = 8 bytes = 8*8 = 64 bits

Pop quiz

Assume YMM register of width 256 bits.

How many integers can fit in a YMM register?

Assume an array has 32 doubles, double arr[32].

How many such registers can be utilized to process the array using SIMD?

Answer

Question: Assume an array has 32 doubles, double arr[32].

How many such registers can be utilized to process the array using SIMD?

- A double precision variable takes 64 bits of storage.
- YMM register is 256 bits wide
- Each YMM register can hold 256/64 = 4 doubles.
- 32 doubles in the array would require 8 YMM registers.

Data types in intrinsic functions

- Use data types instead of register

SIMD Register	Data types
YMM register Example: ymm0	__m256d two underscores _ _ d is for double precision (64 bits) __m256d var; var can be initialized with say {0,0,0,0}

Intrinsic Data Types

#include<immintrin.h>

Regular data types	Intrinsic data types
int	__m256i
float	__m256
double	__m256d

```
for(i=0;i<=MAX;i++)
        c[i]=a[i]+b[i];
```

Code transformation – Loop unrolling

```
for( i = 0; i <= MAX; i = i+4)
{
        c[i+0] =  a[i+0] +  b[i+0];
        c[i+1] = a[i+1]  +  b[i+1];
        c[i+2] = a[i+2]  +  b[i+2];
        c[i+3] = a[i+3]  +  b[i+3];
}
```

```
for(i=0;i<=MAX;i++)
      c[i]=a[i]+b[i];
```

32-bit registers r0, r1, r2

```
Load r1, &a[0]      // load 1 integer
Load r2, &b[0]      // load 1 integer
Add r0, r1, r2      // 1 addition
Store r0, &c[0]     // store 1 integer
```

```
for( i = 0; i <= MAX; i = i+4)
{
      c[i]   =  a[i]  +  b[i];
      c[i+1] = a[i+1]  +  b[i+1];
      c[i+2] = a[i+2]  +  b[i+2];
      c[i+3] = a[i+3]  +  b[i+3];
}
```

128-bit registers R0, R1, R2

```
VLoad R1, &a[0]    // loads 4 ints
VLoad R2, &b[0]    // loads 4 ints
VAdd R0, R1, R2    // 4 additions
VStore R0, &c[0]   // loads 4 ints
```

Unroll the loop

```
int a[12];
for(int i = 0; i < 12; i++)
{
        if(a[i] % 2 == 0)
           printf (i);
}
```

```
int a[12];
for(int i = 0; i < 12; i = i+4)
{
       if(a[i] % 2 == 0)
          printf (i);

       if(a[i+1] % 2 == 0)
          printf (i+1);

       if(a[i+2] % 2 == 0)
          printf (i+2);

      if(a[i+3] % 2 == 0)
          printf (i+3);
}
```

```
int A[256];      int B[256];          int C[256]

for(int i = 0; i < 256; i++)
    C[i] = A[i] * B[i];
```

A	1	2																										**100**
B	10	11																										2
C	10	22																										200

How to use vector instructions?

How many loop iterations?

Many ways to vectorize

- Assembly instructions – e.g., *VMULPD*

- Intrinsics – like C functions

  Convenience: No need to write assembly

```
_mm256_add_pd(x1, x2)        // parallel additions
_mm256_mul_pd(x1, x2)        // parallel multiplications
```

e.g.,

```
__m256d ans = _mm256_mul_pd(__m256d m1, __m256d m2);
```

Link: https://software.intel.com/sites/landingpage/IntrinsicsGuide/

Intrinsics vs assembly instructions

- `__m256d c0 = _mm256_load_pd(memory address);`

  Assembly level

  ```
  vmovapd (%r11),%ymm0   //Load 4 elements of C into %ymm0
  ```

- `_mm256_store_pd( address, c0);`

  Assembly level

  ```
  vmovapd %ymm0,(%r11)   //Store %ymm0 into 4 C elements
  ```

Vector Load practice question

```
double *arr = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0};
__m256d c0 = _mm256_load_pd(arr);
```

What does c0 contain?

- 1.0
- 1.0, 2.0 , 3.0, 4.0
- address of arr
- 1.0, 2.0 , 3.0, 4.0, 5.0, 6.0, 7.0, 8.0

Pop quiz

```
double *arr = {1.0, 2.0 , 3.0, 4.0, 5.0, 6.0, 7.0, 8.0};

__m256d a0 = _mm256_load_pd(arr);
__m256d a4 = _mm256_load_pd(arr + 4);
__m256d sum =  _mm256_add_pd( a0, a4);

double arr[4] = {0, 0, 0, 0};
_mm256_store_pd(arr, sum);
```

arr = &arr[0]

arr+4 = &arr[4]

What does the arr contain?

a) 0, 0, 0, 0
b) 6.0, 8.0, 10.0, 12.0
c) 1.0, 2.0 , 3.0, 4.0
d) 5.0, 6.0, 7.0, 8.0

Writing code using intrinsics

```
void add(double *a, double *b, double *c)
{
        c[i] = a[i]  +   b[i];
}
```

It helps to visualize the unrolled loop

Because 4 iterations can be taken care in a single step

Assume double data type (8 bytes = 64 bits)

void add(double *a, double *b, double *c)

C code	Using Intel Intrinsics

for(i = 0; i <= MAX; i = i+4)
{
c[i] = a[i] + b[i];
c[i+1] = a[i+1] + b[i+1];
c[i+2] = a[i+2] + b[i+2];
c[i+3] = a[i+3] + b[i+3];
}
	for(i = 0; i <= MAX; i = i+4)
	{
	__m256d a0 = _mm256_load_pd(&a[i]);
	__m256d b0 = _mm256_load_pd(&b[i]);
	__m256d c0 = _mm256_add_pd(a0, b0);
	_mm256_store_pd(&c[i], c0);
	}

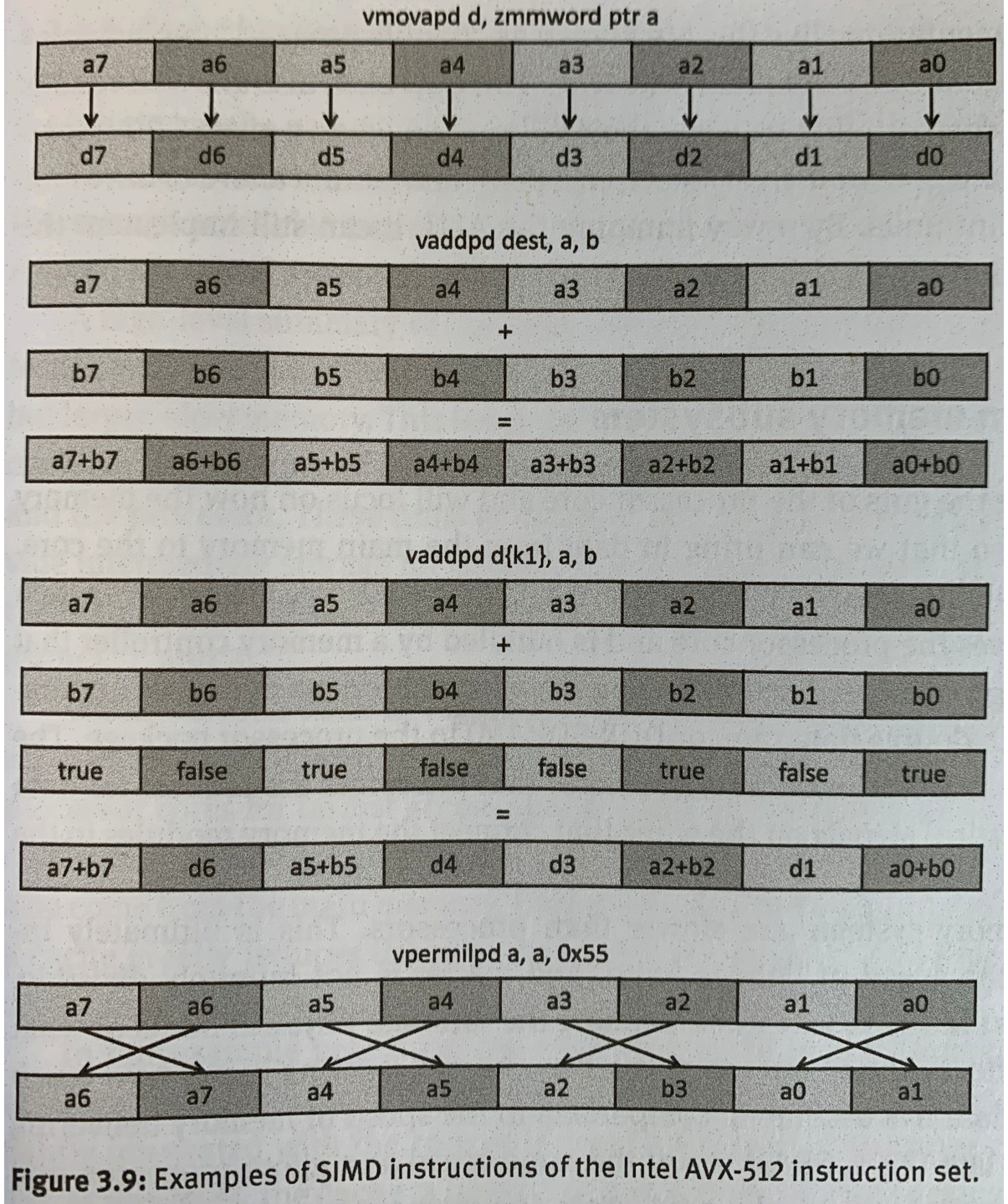


Figure 3.9: Examples of SIMD instructions of the Intel AVX-512 instruction set.

Image Credit:

High Performance Parallel

Runtimes Design and Implementation

Recap

- Data types in intrinsics
  using vector data types instead of registers

- Functions in intrinsics
  using C-like functions instead vector assembly instructions

- Code transformation/translation

- Loop unrolling