Coding for SingleInstructionMultipleData processing

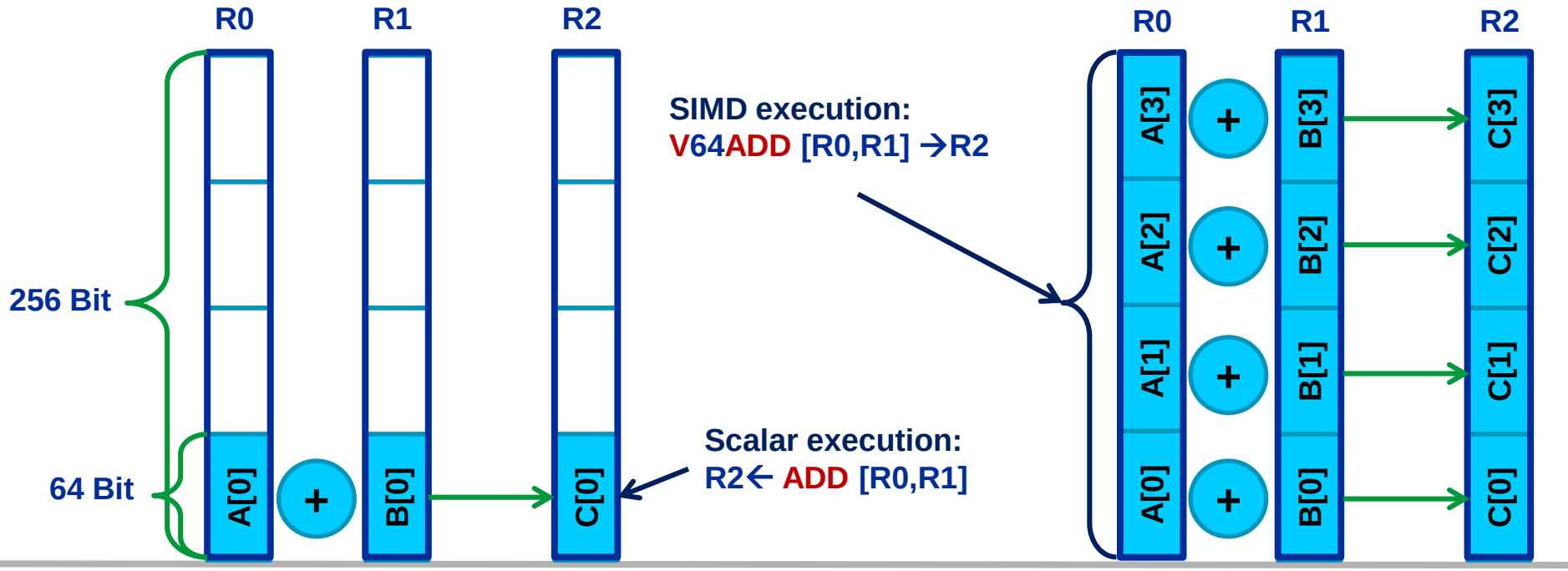
Basics

Compiler options

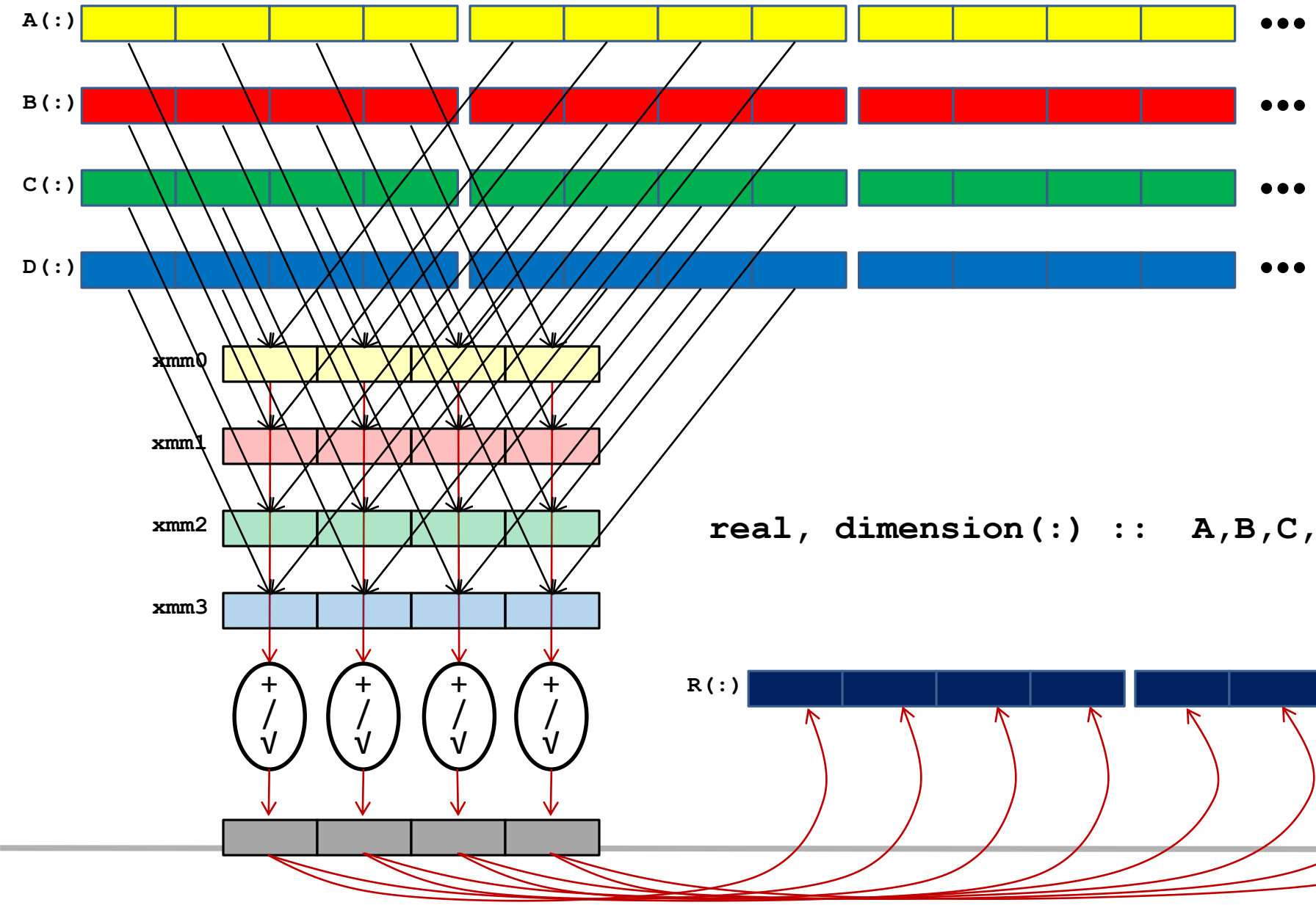
Reading x86 assembly language

SIMD processing – Basics

- **Single Instruction Multiple Data (SIMD) operations allow the concurrent execution of the same operation on "wide" registers.**
- **x86 SIMD instruction sets:**
  - SSE: register width = 128 Bit → 2 double precision floating point operands
  - AVX: register width = 256 Bit → 4 double precision floating point operands
- **Adding two registers holding double precision floating point operands**

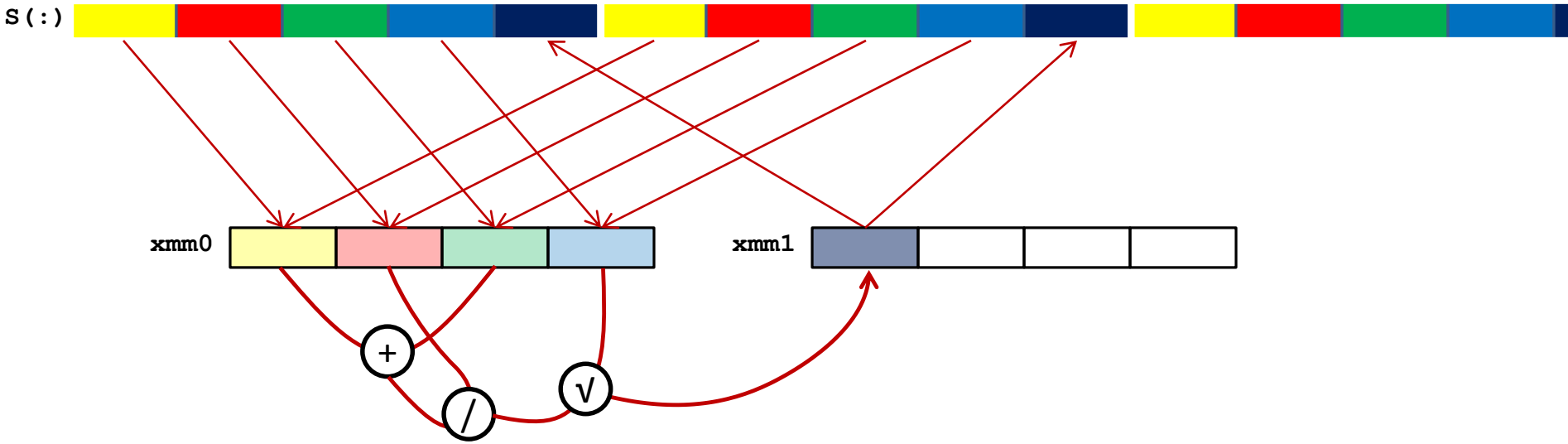


SIMD style structure of arrays processing




```
real, dimension(:) ::  A,B,C,D,R
```

SIMD style of array of structures processing




```
type struct
  real*4  A,B,C,D,R
end type struct

type(struct), dimension(:) :: S
```

SIMD processing – Basics

- Steps (done by the compiler) for "SIMD processing"

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

"Loop unrolling"

```
for(int i=0; i<n;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];}
//remainder loop handling
```

Load 256 Bits starting from address of `A[i]` to register `R0`

Add the corresponding 64 Bit entries in `R0` and `R1` and store the 4 results to `R2`

Store `R2` (256 Bit) to address starting at `C[i]`

```
LABEL1:
        VLOAD R0 ← A[i]
        VLOAD R1 ← B[i]
        V64ADD[R0,R1] → R2
        VSTORE R2 → C[i]
        i←i+4
        i<(n-4)? JMP LABEL1
//remainder loop handling
```

SIMD processing – Basics

- **No SIMD vectorization for loops with data dependencies:**

```
for(int i=0; i<n;i++)
        A[i]=A[i-1]*s;
```

- **"Pointer aliasing" may prevent SIMDfication**

```
void scale_shift(double *A, double *B, double *C, int n) {
        for(int i=0; i<n; ++i)
           C[i] = A[i] + B[i];
}
```

  - C/C++ allows that `A` → `&C[-1]` and `B` → `&C[-2]`
    → `C[i] = C[i-1] + C[i-2]`: **dependency → No SIMD**

  - **If "pointer aliasing" is not used, tell it to the compiler:**
    - `-fno-alias` (Intel), `-Msafeptr` (PGI), `-fargument-noalias` (gcc)
    - `restrict` keyword (C only!):
      `void f(double restrict *a, double restrict *b) {…}`

Vectorization compiler options (Intel)

- **The compiler will vectorize starting with `-O2`.**
- **To enable specific SIMD extensions use the –x option:**
  - **`-xSSE2`** vectorize for SSE2 capable machines

  Available SIMD extensions:

  `SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX`

  - **`-xAVX`** on Sandy Bridge processors

  Recommended option:
  - **`-xHost`** will optimize for the architecture you compile on

  On AMD Opteron: use plain **`-O3`** as the **`-x`** options may involve CPU type checks.

Vectorization compiler options

- **Controlling non-temporal stores (part of the SIMD extensions)**
  - `-opt-streaming-stores always|auto|never`

`always`	use NT stores, assume application is memory bound (use with caution!)
`auto`	compiler decides when to use NT stores
`never`	do not use NT stores unless activated by source code directive

Vectorization source code directives

- Fine-grained control of loop vectorization
- Use `!DEC$` (Fortran) or `#pragma` (C/C++) sentinel to start a compiler directive

- `#pragma vector always`
  vectorize even if it seems inefficient (hint!)

- `#pragma novector`
  do not vectorize even if possible

- `#pragma vector nontemporal`
  use NT stores when allowed (i.e. alignment conditions are met)

- `#pragma vector aligned`
  specifies that all array accesses are aligned to 16-byte boundaries (DANGEROUS! You must not lie about this!)

User mandated vectorization

- **Since Intel Compiler 12.0 the `simd` pragma is available**
- **`#pragma simd` enforces vectorization where the other pragmas fail**
- **Prerequesites:**
  - Countable loop
  - Innermost loop
  - Must conform to for-loop style of OpenMP worksharing constructs
- **There are additional clauses: `reduction, vectorlength, private`**
- **Refer to the compiler manual for further details**

```
#pragma simd reduction(+:x)
  for (int i=0; i<n; i++) {
     x = x + A[i];
  }
```

- **NOTE: Using the #pragma simd the compiler may generate incorrect code if the loop violates the vectorization rules!**

- **Alignment issues**
  - Alignment of arrays with SSE (AVX) should be on 16-byte (32-byte) boundaries to allow packed aligned loads and NT stores **(for Intel processors)**
    - **AMD has a scalar nontemporal store instruction**
  - Otherwise the compiler will revert to unaligned loads and not use NT stores – even if you say **`vector nontemporal`**
  - Modern x86 CPUs have less (not zero) impact for misaligned LD/ST, but **Xeon Phi relies heavily on it!**
  - How is manual alignment accomplished?
- **Dynamic allocation of aligned memory (`align` = alignment boundary):**

```
#define _XOPEN_SOURCE 600
#include <stdlib.h>

int posix_memalign(void **ptr,
                   size_t align,
                   size_t size);
```

Reading x86 assembly code and exploiting SIMD parallelism

Understanding SIMD execution by inspecting assembly code

SIMD vectorization how-to

Why check the assembly code?

- **Sometimes the only way to make sure the compiler "did the right thing"**
  - Example: "LOOP WAS VECTORIZED" message is printed, but Loads & Stores may still be scalar!

- **Get the assembler code (Intel compiler):**

```
icc -S -masm=intel -O3  -xHost  triad.c  -o a.out
```

- **Disassemble Executable:**

```
objdump -d  ./a.out | less
```

The x86 ISA is documented in:

Intel Software Development Manual (SDM) 2A and 2B
AMD64 Architecture Programmer's Manual Vol. 1-5

Basics of the x86-64 ISA

- Instructions have 0 to 4 operands
- Operands can be registers, memory references or immediates
- Opcodes (binary representation of instructions) vary from 1 to 17 bytes
- There are two syntax forms: Intel (left) and AT&T (right)
- Addressing Mode: BASE + INDEX * SCALE + DISPLACEMENT
- C: `A[i]` equivalent to `*(A+i)` (a pointer has a type: `A+i*8`)

```
movaps [rdi + rax*8+48], xmm3
add rax, 8
js 1b
```

```
movaps    %xmm4, 48(%rdi,%rax,8)
addq      $8, %rax
js        ..B1.4
```

```
401b9f: 0f 29 5c c7 30      movaps %xmm3,0x30(%rdi,%rax,8)
401ba4: 48 83 c0 08         add    $0x8,%rax
401ba8: 78 a6               js     401b50 <triad_asm+0x4b>
```

16 general Purpose Registers (64bit):

`rax, rbx, rcx, rdx, rsi, rdi, rsp, rbp, r8-r15`

alias with eight 32 bit register set:

`eax, ebx, ecx, edx, esi, edi, esp, ebp`

Floating Point SIMD Registers:

`xmm0-xmm15` SSE (128bit) alias with 256-bit registers

`ymm0-ymm15` AVX (256bit)

SIMD instructions are distinguished by:

AVX (VEX) prefix:	`v`
Operation:	`mul, add, mov`
Modifier:	nontemporal (`nt`), unaligned (`u`), aligned (`a`), high (`h`)
Width:	scalar (`s`), packed (`p`)
Data type:	single (`s`), double (`d`)

Case study: Vector Triad (DP) on IvyBridge-EP

```
for (int i = 0; i < length; i++) {
    A[i] = B[i] + D[i] * C[i];
}
```

To get object code use `objdump -d` on object file or executable or compile with `-S`

Assembly code (-O1):

CLANG

```
LBB0_3
movsd   xmm0, [rdx]
mulsd   xmm0, [rcx]
addsd   xmm0, [rsi]
movsd   [rax], xmm0
add     rsi, 8
add     rdx, 8
add     rcx, 8
add     rax, 8
dec     edi
jne     LBB0_3
```

ICC

```
..B1.6:
movsd   xmm0, [r12+rax*8]
mulsd   xmm0, [r13+rax*8]
addsd   xmm0, [r14+rax*8]
movsd   [r15+rax*8], xmm0
inc     rax
cmp     rax, rbx
jl      ..B1.6
```

GCC

```
.L4:
movsd xmm0,[rbx+rax]
mulsd xmm0,[r12+rax]
addsd xmm0,[r13+0+rax]
movsd [rbp+0+rax],xmm0
add rax, 8
cmp rax, r14
jne .L4
```

7 instructions per loop iteration

Case study: Vector Triad (DP) –O3 (Intel compiler)

SSE

```
..B1.19:
        movsd     xmm0,  [r15+rsi*8]
        movsd     xmm3,  [16+r15+rsi*8]
        movsd     xmm5,  [32+r15+rsi*8]
        movsd     xmm7,  [48+r15+rsi*8]
        movhpd    xmm0,  [8+r15+rsi*8]
        movhpd    xmm3,  [24+r15+rsi*8]
        movhpd    xmm5,  [40+r15+rsi*8]
        movhpd    xmm7,  [56+r15+rsi*8]
        mulpd     xmm0,  [r14+rsi*8]
        mulpd     xmm3,  [16+r14+rsi*8]
        mulpd     xmm5,  [32+r14+rsi*8]
        mulpd     xmm7,  [48+r14+rsi*8]
        movsd     xmm2,  [r13+rsi*8]
        movsd     xmm4,  [16+r13+rsi*8]
        movsd     xmm6,  [32+r13+rsi*8]
        movsd     xmm8,  [48+r13+rsi*8]
        movhpd    xmm2,  [8+r13+rsi*8]
        movhpd    xmm4,  [24+r13+rsi*8]
        movhpd    xmm6,  [40+r13+rsi*8]
        movhpd    xmm8,  [56+r13+rsi*8]
        addpd     xmm2, xmm0
        addpd     xmm4, xmm3
        addpd     xmm6, xmm5
        addpd     xmm8, xmm7
        movaps    [rdx+rsi*8], xmm2
        movaps    [16+rdx+rsi*8], xmm4
        movaps    [32+rdx+rsi*8], xmm6
        movaps    [48+rdx+rsi*8], xmm8
        add       rsi, 8
        cmp       rsi, r9
        jb        ..B1.19
```

3.86 instructions per loop iteration

Case study: Vector Triad (DP) –O3 –xHost

AVX

```
..B1.15:
vmovupd    xmm2, [r15+rsi*8]
vmovupd    xmm10, [32+r15+rsi*8]
vmovupd    xmm3, [rdx+rsi*8]
vmovupd    xmm11, [32+rdx+rsi*8]
vmovupd    xmm0, [r14+rsi*8]
vmovupd    xmm9, [32+r14+rsi*8]
vinsertf128 ymm4, ymm2,[16+r15+rsi*8], 1
vinsertf128 ymm12,ymm10,[48+r15+rsi*8],1
vinsertf128 ymm5, ymm3,[16+rdx+rsi*8], 1
vinsertf128 ymm13,ymm11,[48+rdx+rsi*8],1
vmulpd     ymm7, ymm4, ymm5
vmulpd     ymm15, ymm12, ymm13
vmovupd    xmm4, [64+rdx+rsi*8]
vmovupd    xmm12, [96+rdx+rsi*8]
vmovupd    xmm3, [64+r15+rsi*8]
vmovupd    xmm11, [96+r15+rsi*8]
vmovupd    xmm2, [64+r14+rsi*8]
vmovupd    xmm10, [96+r14+rsi*8]
vinsertf128 ymm14,ymm9,[48+r14+rsi*8], 1
vinsertf128 ymm6,ymm0,[16+r14+rsi*8], 1
vaddpd     ymm8, ymm6, ymm7
vaddpd     ymm0, ymm14, ymm15
vmovupd    [r13+rsi*8], ymm8
vmovupd    [32+r13+rsi*8], ymm0
vinsertf128 ymm5, ymm3, [80+r15+rsi*8], 1
vinsertf128 ymm13,ymm11,[112+r15+rsi*8], 1
vinsertf128 ymm6, ymm4,  [80+rdx+rsi*8], 1
vinsertf128 ymm14,ymm12,[112+rdx+rsi*8], 1
vmulpd     ymm8, ymm5, ymm6
vmulpd     ymm0, ymm13, ymm14
vinsertf128 ymm7, ymm2, [80+r14+rsi*8], 1
vinsertf128 ymm15,ymm10,[112+r14+rsi*8], 1
vaddpd     ymm9, ymm7, ymm8
vaddpd     ymm2, ymm15, ymm0
vmovupd    [64+r13+rsi*8], ymm9
vmovupd    [96+r13+rsi*8], ymm2
add        rsi, 16
cmp        rsi, r9
jb         ..B1.15
```

2.44 instructions per loop iteration

Benefit of SIMD limited by “serial fraction” 16-byte wide loads)!

Case study: Vector Triad (DP) –O3 –xHost

`#pragma vector aligned`

SSE

```
..B1.7:
movaps    xmm0, [r13+rcx*8]
movaps    xmm2, [16+r13+rcx*8]
movaps    xmm3, [32+r13+rcx*8]
movaps    xmm4, [48+r13+rcx*8]
mulpd     xmm0, [rbp+rcx*8]
mulpd     xmm2, [16+rbp+rcx*8]
mulpd     xmm3, [32+rbp+rcx*8]
mulpd     xmm4, [48+rbp+rcx*8]
addpd     xmm0, [r12+rcx*8]
addpd     xmm2, [16+r12+rcx*8]
addpd     xmm3, [32+r12+rcx*8]
addpd     xmm4, [48+r12+rcx*8]
movaps    [r15+rcx*8], xmm0
movaps    [16+r15+rcx*8], xmm2
movaps    [32+r15+rcx*8], xmm3
movaps    [48+r15+rcx*8], xmm4
add       rcx, 8
cmp       rcx, rsi
jb        ..B1.7
```

2.38 instructions per loop iteration

AVX

```
..B1.7:
vmovupd   ymm0, [r15+rcx*8]
vmovupd   ymm4, [32+r15+rcx*8]
vmovupd   ymm7, [64+r15+rcx*8]
vmovupd   ymm10,[96+r15+rcx*8]
vmulpd    ymm2, ymm0, [rdx+rcx*8]
vmulpd    ymm5, ymm4, [32+rdx+rcx*8]
vmulpd    ymm8, ymm7, [64+rdx+rcx*8]
vmulpd    ymm11, ymm10, [96+rdx+rcx*8]
vaddpd    ymm3, ymm2, [r14+rcx*8]
vaddpd    ymm6, ymm5, [32+r14+rcx*8]
vaddpd    ymm9, ymm8, [64+r14+rcx*8]
vaddpd    ymm12, ymm11, [96+r14+rcx*8]
vmovupd   [r13+rcx*8], ymm3
vmovupd   [32+r13+rcx*8], ymm6
vmovupd   [64+r13+rcx*8], ymm9
vmovupd   [96+r13+rcx*8], ymm12
add       rcx, 16
cmp       rcx, rsi
jb        ..B1.7
```

1.19 instructions per loop iteration

Case study: Vector Triad (DP) –O3 –xHost
`#pragma vector aligned` on Haswell-EP

AVX + FMA3

```
..B1.7:
vmovupd    ymm2, [r15+rcx*8]
vmovupd    ymm4, [32+r15+rcx*8]
vmovupd    ymm6, [64+r15+rcx*8]
vmovupd    ymm8, [96+r15+rcx*8]
vmovupd    ymm0, [rdx+rcx*8]
vmovupd    ymm3, [32+rdx+rcx*8]
vmovupd    ymm5, [64+rdx+rcx*8]
vmovupd    ymm7, [96+rdx+rcx*8]
vfmadd213pd ymm2, ymm0, [r14+rcx*8]
vfmadd213pd ymm4, ymm3, [32+r14+rcx*8]
vfmadd213pd ymm6, ymm5, [64+r14+rcx*8]
vfmadd213pd ymm8, ymm7, [96+r14+rcx*8]
vmovupd    [r13+rcx*8], ymm2
vmovupd    [32+r13+rcx*8], ymm4
vmovupd    [64+r13+rcx*8], ymm6
vmovupd    [96+r13+rcx*8], ymm8
add        rcx, 16
cmp        rcx, rsi
jb         ..B1.7
```

1.19 instructions per loop iteration

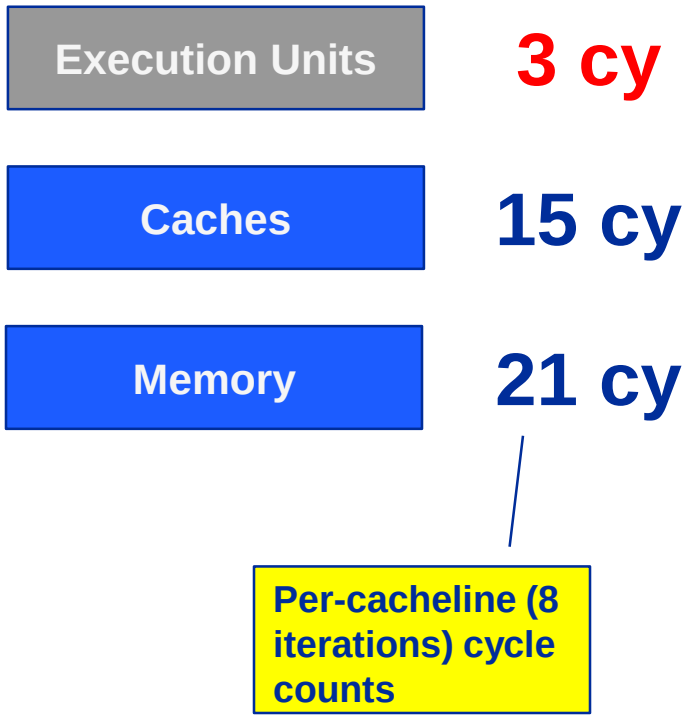
On X86 ISA instruction are converted to so-called µops (elementary ops like load, add, mult). For performance the number of µops is important.

23 uops vs. 27 µops (AVX)

SIMD processing – The whole picture

SIMD influences instruction execution in the core – other runtime contributions stay the same!

AVX example:



Comparing total execution time:

	Execution	Cache	Memory
Scalar	12		
SSE	6	15	21
AVX	3		

Total runtime with data loaded from memory:

Scalar	48
SSE	42
AVX	39

SIMD only effective if runtime is dominated by instructions execution!

How to leverage SIMD: your options

Alternatives:

- The compiler does it for you (but: aliasing, alignment, language)
- Compiler directives (pragmas)
- Alternative programming models for compute kernels (OpenCL, ispc)
- Intrinsics (restricted to C/C++)
- Implement directly in assembler

To use intrinsics the following headers are available:

- `xmmintrin.h` (SSE)
- `pmmintrin.h` (SSE2)
- `immintrin.h` (AVX)
- `x86intrin.h` (all extensions)

```
for (int j=0; j<size; j+=16){
    t0 = _mm_loadu_ps(data+j);
    t1 = _mm_loadu_ps(data+j+4);
    t2 = _mm_loadu_ps(data+j+8);
    t3 = _mm_loadu_ps(data+j+12);
    sum0 = _mm_add_ps(sum0, t0);
    sum1 = _mm_add_ps(sum1, t1);
    sum2 = _mm_add_ps(sum2, t2);
    sum3 = _mm_add_ps(sum3, t3);
}
```

Rules for vectorizable loops

1. Countable
2. Single entry and single exit
3. Straight line code
4. No function calls (exception intrinsic math functions)

Better performance with:

1. Simple inner loops with unit stride
2. Minimize indirect addressing
3. Align data structures (SSE 16 bytes, AVX 32 bytes)
4. In C use the restrict keyword for pointers to rule out aliasing

Obstacles for vectorization:

1. Non-contiguous memory access
2. Data dependencies