

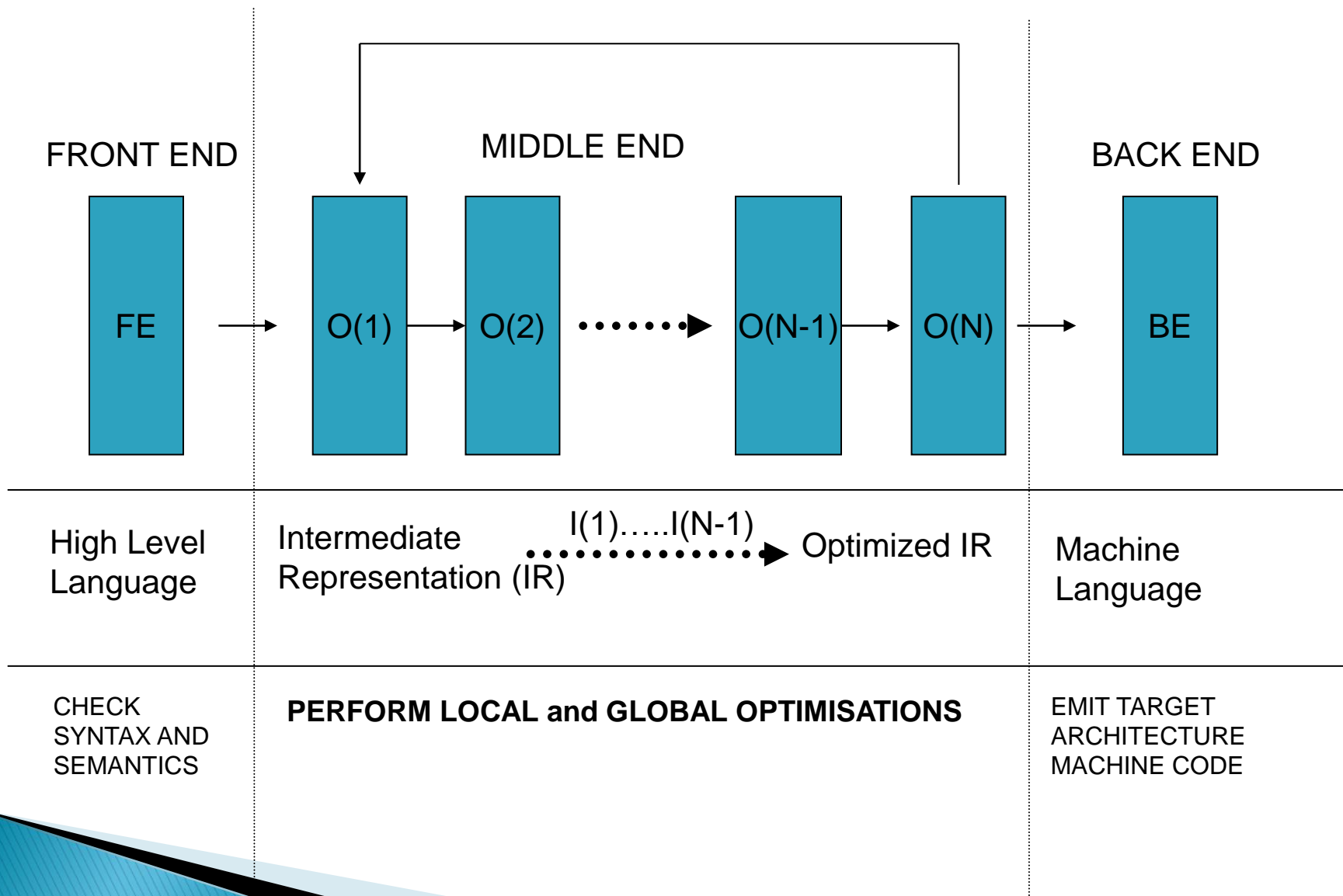
Practical Parallel Programming II

»» Speeding up the sequential program

Jan Lemeire
2021–2022

Compiler optimizations

Architecture of a typical optimizing compiler



Function inlining

Function macros

- ▶ In C/C++
- ▶ example:

```
#define ABS(my_val) ((my_val) < 0) ?  
- (my_val) : (my_val)
```

- ▶ Use of macro: function is replaced by the precompiler
- ▶ But: parameters and types are not checked
- ▶ Alternative: inline functions

Function inlining

- ▶ Instead of a function call, the functions is replace by the code
- ▶ Function call:
 - need of stack storage to pass parameters and store variables
 - A jump to start the function and one to return to the calling code
- ▶ Inlining of the function:
 - duplicate code
 - May lead to more instruction cache misses
- ▶ Use the keyword **`inline`** to inform the compiler that inlining is a good idea

Loop unrolling

Reducing branch frequency

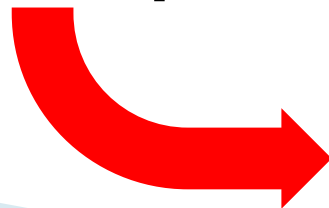
- ▶ One way to make branches faster is to... not branch as much.
- ▶ **Loop unrolling** is a technique to reduce the number of branches. It does this by duplicating the loop body, reducing the number of iterations needed.

```
for(i = 0; i < 100; i++)  
    a[i] = b[i] + c[i];
```

Original loop

Unrolled loop (2X)

```
for(i = 0; i < 100; i += 2){  
    a[i] = b[i] + c[i];  
    a[i+1] = b[i+1] + c[i+1];  
}
```



STATIC LOOP UNROLLING – a trivial translation to MIPS

```
for (i = 1000 ; i > 0 ; i -- ) {  
    x[ i ] = x[ i ] + constant;  
}
```

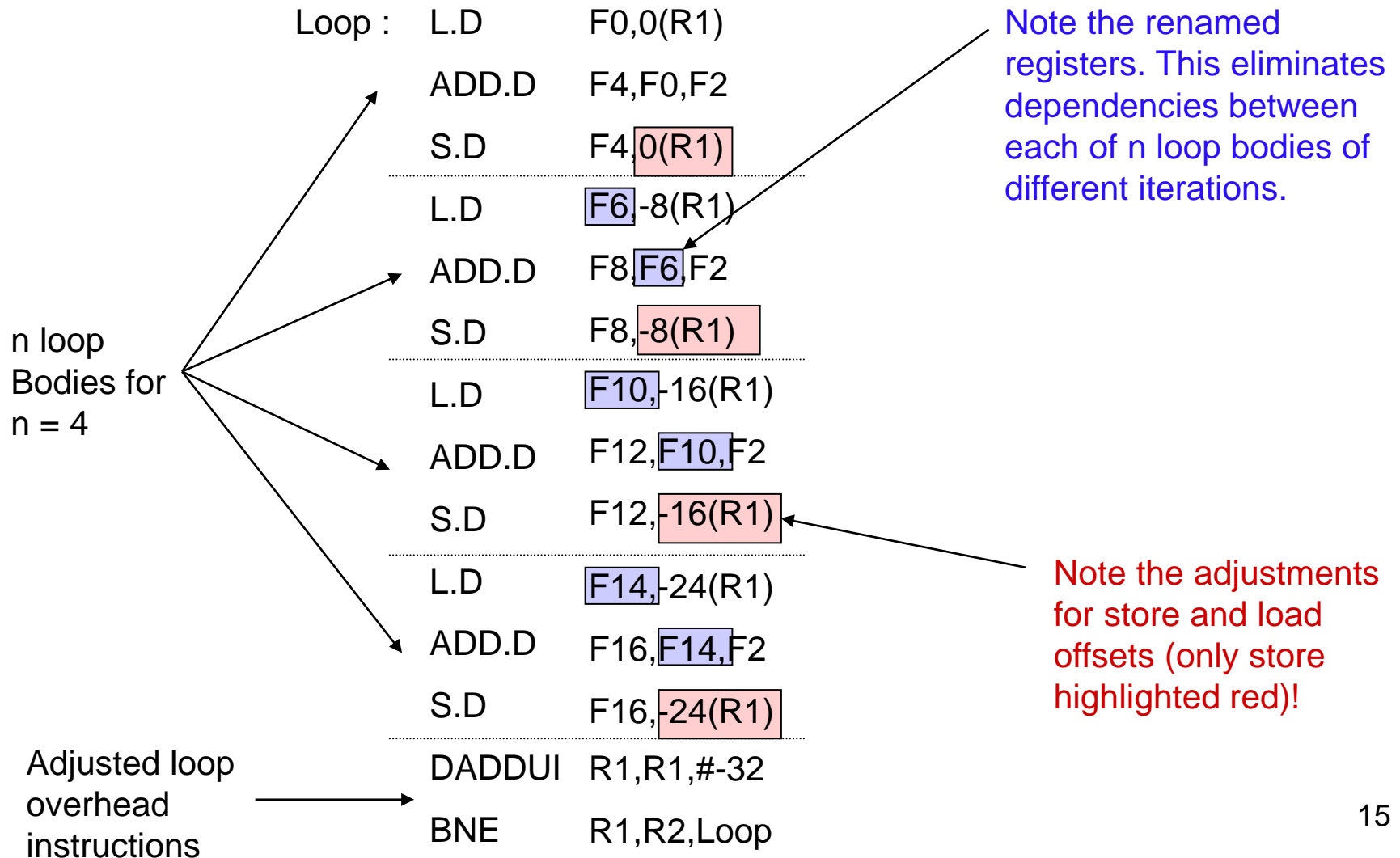
Our example translates into the MIPS assembly code below (**without any scheduling**).

Note the loop independent dependence in the loop ,i.e. $x[i]$ on $x[i]$

```
Loop :  L.D      F0,0(R1)      ; F0 = array elem.  
        ADD.D   F4,F0,F2      ; add scalar in F2  
        S.D     F4,0(R1)      ; store result  
        DADDUI  R1,R1,#-8      ; decrement ptr  
        BNE     R1,R2,Loop     ; branch if R1 !=R2
```

STATIC LOOP UNROLLING – issuing our instructions

→ The unrolled loop from the running example with an unroll factor of $n = 4$ would then be:



How far to unroll?

- ▶ The previous example doubled the code in the loop. Of course we can unroll 3X, 4X, 8X... what are the tradeoffs?
- ▶ **Space vs. time** is the big one.
 - But memory today is big, network connections are fast... is this so much of a problem?
 - Well.....
- ▶ **Caching** is the big bottleneck these days.
 - The bigger the code is, the less of it will fit in the cache.
 - This is bad

Vector instructions

Instruction and Data Streams

		Data Streams	
		Single	Multiple
Instruction Streams	Single	SISD: Intel Pentium 4	SIMD: SSE instructions of x86
	Multiple	MISD: No examples today	MIMD: Intel Xeon e5345

SIMD vector instructions

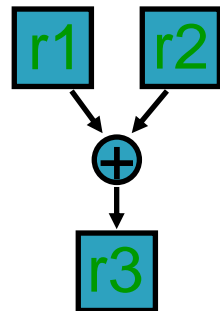
- ▶ Perform an instruction on 4/8/16 floats (special registers) at once
 - E.g., MMX and SSE instructions in x86
 - Multiple data elements in 128-bit wide registers
- ▶ All processors execute the same instruction at the same time
 - Each with different data address, etc.
- ▶ Simplifies synchronization
- ▶ Reduced instruction control hardware
- ▶ Works best for highly data-parallel applications

Vector Processing

- Vector processing exploits data parallelism by performing the same computation on linear arrays of numbers "vectors" using one instruction.

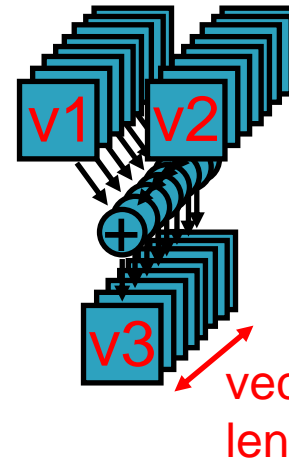
Scalar
ISA
(RISC
or CISC)

SCALAR
(1 operation)



Add.d F3, F1, F2

VECTOR
(N operations)



addv.d v3, v1, v2

Vector
ISA

Up to
Maximum
Vector
Length
(MVL)

Vector Processors

- ▶ Highly pipelined function units
- ▶ Stream data from/to vector registers to units
 - Data collected from memory into registers
 - Results stored from registers to memory
- ▶ Example: Vector extension to MIPS
 - 32×64 -element registers (64-bit elements)
 - Vector instructions (on up to 64 elements!)
 - `lv, sv`: load/store vector
 - `addv.d`: add vectors of double
 - `addvs.d`: add scalar to each element of vector of double
- ▶ Significantly reduces instruction-fetch bandwidth

Example: DAXPY ($Y = a \times X + Y$)

► Conventional MIPS code

```

loop:  l.d    $f0,a($sp)           ;load scalar a
        addiu r4,$s0,#512         ;upper bound of what to load
        l.d    $f2,0($s0)         ;load x(i)
        mul.d   $f2,$f2,$f0        ;a × x(i)
        l.d    $f4,0($s1)         ;load y(i)
        add.d   $f4,$f4,$f2        ;a × x(i) + y(i)
        s.d    $f4,0($s1)         ;store into y(i)
        addiu  $s0,$s0,#8          ;increment index to x
        addiu  $s1,$s1,#8          ;increment index to y
        subu   $t0,r4,$s0         ;compute bound
        bne    $t0,$zero,loop     ;check if done

```

► Vector MIPS code (if X and Y contain 64-element)

```

l.d    $f0,a($sp)           ;load scalar a
lv     $v1,0($s0)           ;load vector x
mulvs.d $v2,$v1,$f0         ;vector-scalar multiply
lv     $v3,0($s1)           ;load vector y
addv.d $v4,$v2,$v3          ;add y to product
sv     $v4,0($s1)           ;store the result

```

Properties of Vector Processors

- ▶ Each result in a vector operation is independent of previous results (Data Parallelism, LLP exploited)
 - => long pipelines used, compiler ensures no dependencies
 - => higher clock rate
- ▶ Vector instructions access memory with known patterns
 - => Highly interleaved memory with multiple banks used to provide the high bandwidth needed and hide memory latency.
 - => Amortize memory latency of over many vector elements
 - => no (data) caches usually used. (Do use instruction cache)
- ▶ A single vector instruction implies a large number of computations (replacing loops or reducing number of iterations needed)
 - => Fewer instructions fetched/executed
 - => Reduces branches and branch problems in pipelines

Vector vs. Scalar

- ▶ Vector architectures and compilers
 - Simplify data-parallel programming
 - Explicit statement of absence of loop-carried dependences
 - Reduced checking in hardware
 - Regular access patterns benefit from interleaved and burst memory
 - Avoid control hazards by avoiding loops
- ▶ More general than ad-hoc media extensions (such as MMX, SSE)
 - Better match with compiler technology

Full Vector processors

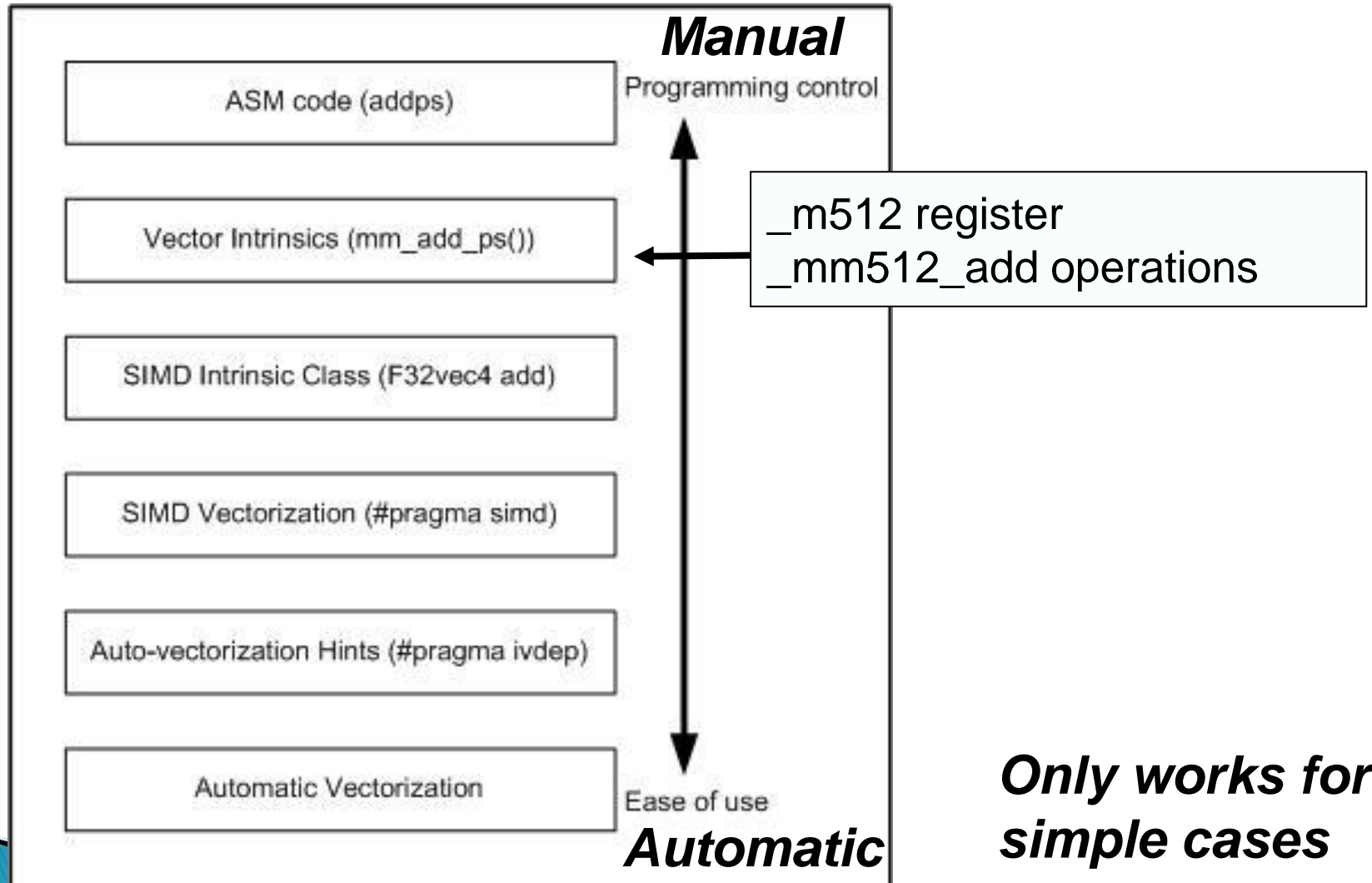
- ▶ Has long be viewed as the solution for high-performance computing
 - Why always repeating the same instructions (on different data)? => just apply the instruction immediately on all data
- ▶ 80s: special vector computers (e.g. the Cray that VUB/ULB bought)
 - But could not compete with processors of commodity devices
- ▶ However: difficult to program
- ▶ Is SIMT (GPUs) a better alternative??

Power Efficient?

- ▶ The majority of power consumption of a CPU is not from the ALU
 - Cache management, data movement, decoding, and other infrastructure
 - Adding a few more ALUs should not impact power consumption
- ▶ Indeed, 4X performance via AVX does not add 4X power consumption
 - From i7 4770K measurements:
 - Idle: 40 W
 - Under load : 117 W
 - Under AVX load : 128 W

**Automatic
vectorization?**

Vectorization



Compiler Automatic Vectorization

- ▶ In gcc, flags “-O3 -mavx -mavx2” attempts automatic vectorization
- ▶ Works pretty well for simple loops

```
int a[256], b[256], c[256];
void foo () {
    for (int i=0; i<256; i++) a[i] = b[i] * c[i];
}
```




```
.L2:
    vmovdqa xmm1, XMMWORD PTR b[rax]
    add     rax, 16
    vpmulld xmm0, xmm1, XMMWORD PTR c[ra-16]
    vmovaps XMMWORD PTR a[ra-16], xmm0
    cmp     rax, 1024
    jne     .L2
```

Generated using GCC explorer: <https://gcc.godbolt.org/>

- ▶ But not for anything complex
 - E.g., naïve bubblesort code not parallelized at all

SIMD pragma to indicate parallism



```
void dflops(double * restrict a) {
    const double c = 1.;
    const double x = 0.9;
    #pragma simd
    for (long long i = 0; i < niterations; i += 16) {
        a[0] = a[0] * x + c;
        a[1] = a[1] * x + c;
        a[2] = a[2] * x + c;
        a[3] = a[3] * x + c;
        a[4] = a[4] * x + c;
        a[5] = a[5] * x + c;
        a[6] = a[6] * x + c;
        a[7] = a[7] * x + c;

        a[8] = a[8] * x + c;
        a[9] = a[9] * x + c;
        a[10] = a[10] * x + c;
        a[11] = a[11] * x + c;
        a[12] = a[12] * x + c;
        a[13] = a[13] * x + c;
        a[14] = a[14] * x + c;
        a[15] = a[15] * x + c;
    }
}
```

Successful vectorization

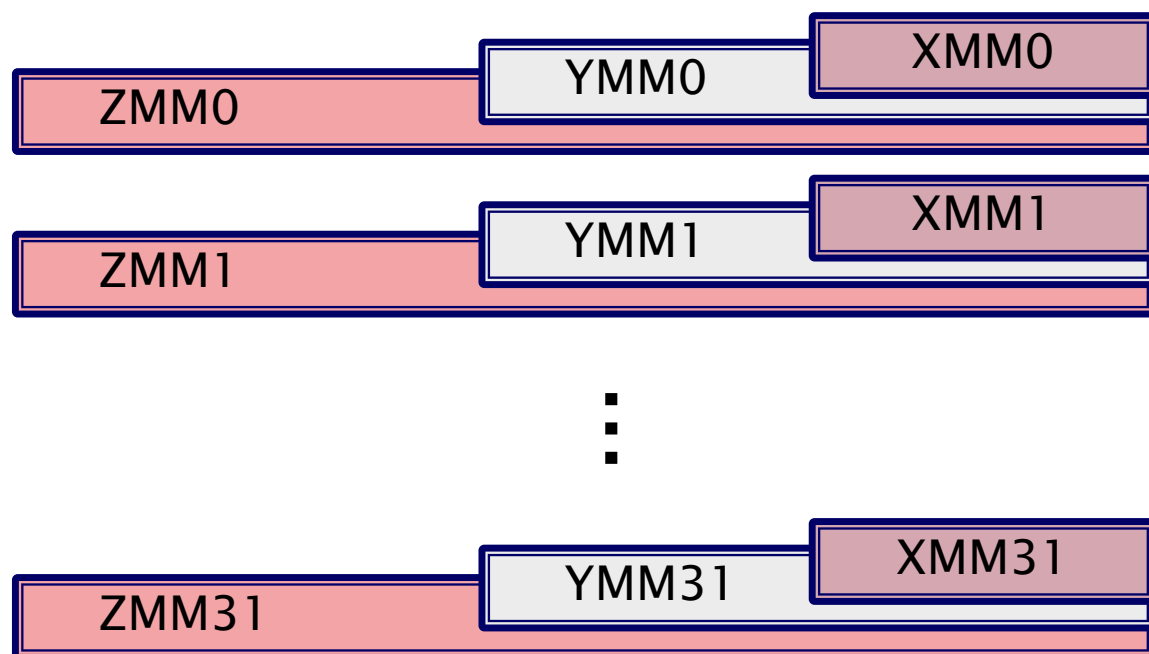
```
rdewaele@knc-2:~/Projects/adhd/simpleflops/simpleflops$ CFLAGS="-vec-re
icc -vec-report6 -mmic -std=c99 -O3 -fopenmp -funroll-loops -vec-report
test.c(84): (col. 5) remark: vectorization support: reference sa has un
test.c(84): (col. 5) remark: vectorization support: unaligned access us
test.c(83): (col. 4) remark: LOOP WAS VECTORIZED.
test.c(76): (col. 3) remark: loop was not vectorized: not inner loop.
test.c(74): (col. 2) remark: loop was not vectorized: not inner loop.
test.c(79): (col. 4) remark: SIMD LOOP WAS VECTORIZED.
test.c(13): (col. 2) remark: SIMD LOOP WAS VECTORIZED.
test.c(38): (col. 2) remark: SIMD LOOP WAS VECTORIZED.
```

x86 vectors

Intel SIMD Extensions

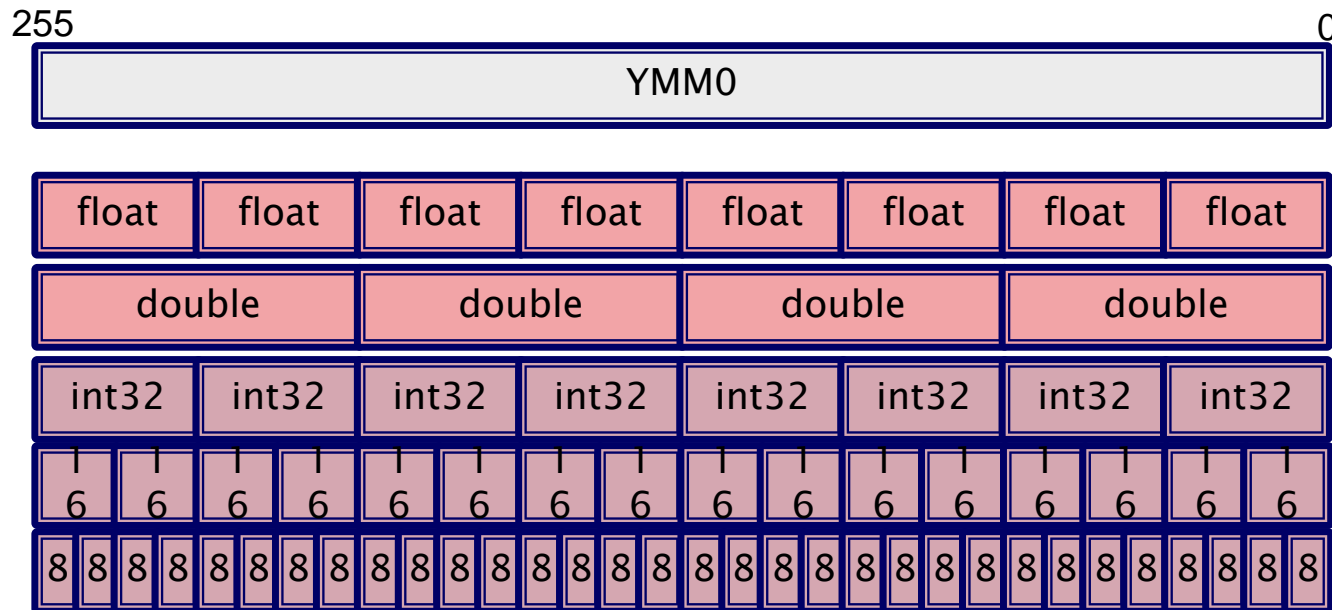
- ▶ New instructions, new registers
- ▶ Introduced in phases/groups of functionality
 - SSE – SSE4 (1999 – 2006)
 - 128 bit width operations
 - AVX, FMA, AVX2, AVX-512 (2008 – 2015)
 - 256 – 512 bit width operations
 - AVX-512 chips not available yet (as of Spring, 2019), but soon!
- ▶ F16C, and more to come?

Intel SIMD Registers (AVX-512)



- ▶ XMM0 – XMM15
 - 128-bit registers
 - SSE
- ▶ YMM0 – YMM15
 - 256-bit registers
 - AVX, AVX2
- ▶ ZMM0 – ZMM31
 - 512-bit registers
 - AVX-512

SSE/AVX Data Types



Operation on 32 8-bit values in one instruction!

Data Types in AVX/AVX2

Type	Description
__m128	128-bit vector containing 4 floats
__m128d	128-bit vector containing 2 doubles
__m128i	128-bit vector containing integers
__m256	256-bit vector containing 8 floats
__m256d	256-bit vector containing 4 doubles
__m256i	256-bit vector containing integers

__m512 variants for AVX-512

Intrinsic Naming Convention

- ▶ `_mm<width>_[function]_[type]`
 - E.g., `_mm256_fmadd_ps` :
perform `fmadd` (floating point multiply-add) on
256 bits of
packed single-precision floating point values (8 of them)

Width	Prefix
128	<code>_mm_</code>
256	<code>_mm256_</code>
512	<code>_mm512_</code>

Not all permutations exist! Check guide

Type	Postfix
Single precision	<code>_ps</code>
Double precision	<code>_pd</code>
Packed signed integer	<code>_epiNNN</code> (e.g., <code>epi256</code>)
Packed unsigned integer	<code>_epuNNN</code> (e.g., <code>epu256</code>)
Scalar integer	<code>_siNNN</code> (e.g., <code>si256</code>)

Load/Store/Initialization Operations

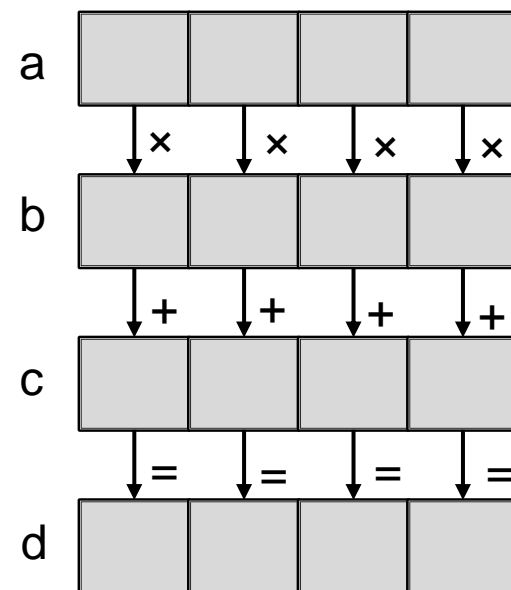
- ▶ Initialization
 - `_mm256_setzero_ps/pd/epi32/...`
 - `_mm256_set_...`
 - ...
- ▶ Load/Store : Address **MUST** be aligned by 256-bit
 - `_mm256_load_...`
 - `_mm256_store_...`
- ▶ And many more! (Masked read/write, strided reads, etc...)

e.g.,

```
__mm256d t = _mm256_load_pd(double const * mem); // loads 4 double values  
__mm256i v = _mm256_set_epi32(h,g,f,e,d,c,b,a); // loads 8 integer values
```

Vertical Vector Instructions

- ▶ Add/Subtract/Multiply
 - `_mm256_add/sub/mul/div_ps/pd/epi`
 - Mul only supported for `epi32/epu32/ps/pd`
 - Div only supported for `ps/pd`
 - Consult the guide!
- ▶ Max/Min/GreaterThan/Equals
- ▶ Sqrt, Reciprocal, Shift, etc...
- ▶ FMA (Fused Multiply-Add)
 - $(a*b)+c$, $-(a*b)-c$, $-(a*b)+c$, and other permutations!
 - Consult the guide!
- ▶ ...



```
__m256 a, b, c;
__m256 d = _mm256_fmadd_pd(a, b, c);
```

Integer Multiplication Caveat

- ▶ Integer multiplication of two N bit values require 2N bits
- ▶ E.g., `__mm256_mul_epi32` and `__mm256_mul_epu32`
 - Only use the lower 4 32-bit values
 - Result has 4 64-bit values
- ▶ E.g., `__mm256_mullo_epi32` and `__mm256_mullo_epu32`
 - Uses all 8 32-bit values
 - Result has 8 truncated 32-bit values
- ▶ And more options! Consult the guide...

Horizontal Vector Instructions

- ▶ Horizontal add/subtraction
 - Adds adjacent pairs of values
 - E.g., `__m256d _mm256_hadd_pd (__m256d a, __m256d b)`

