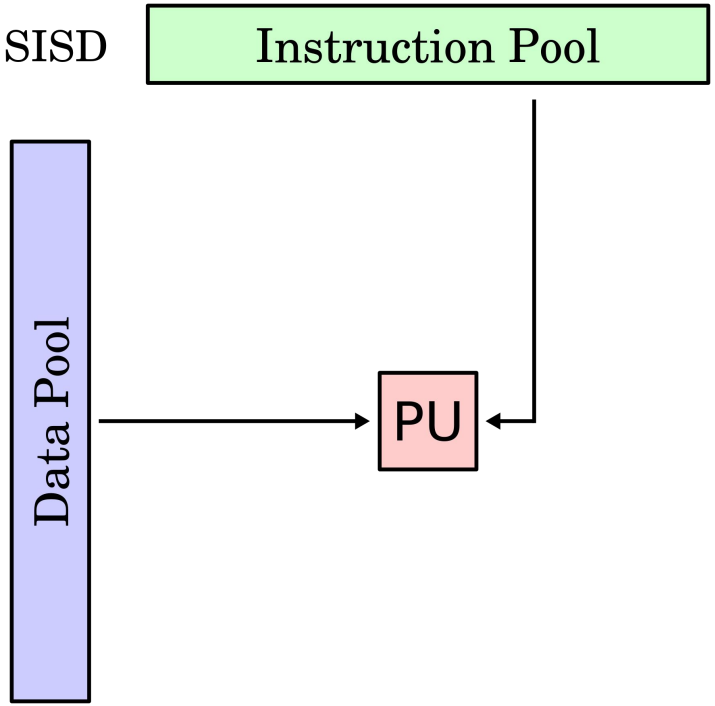
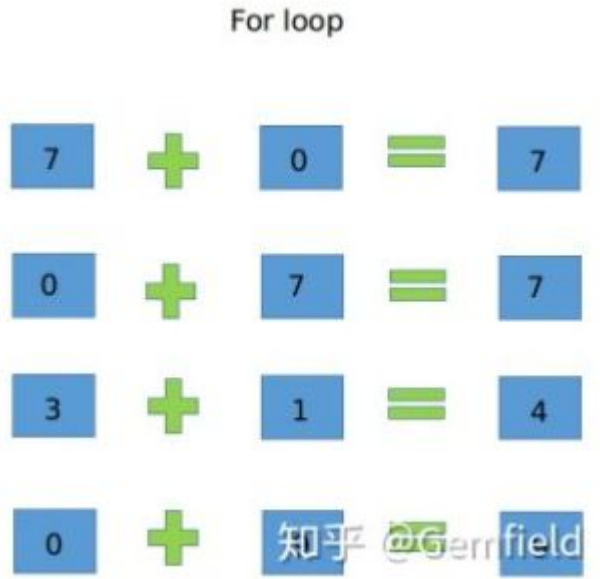
Vectorization

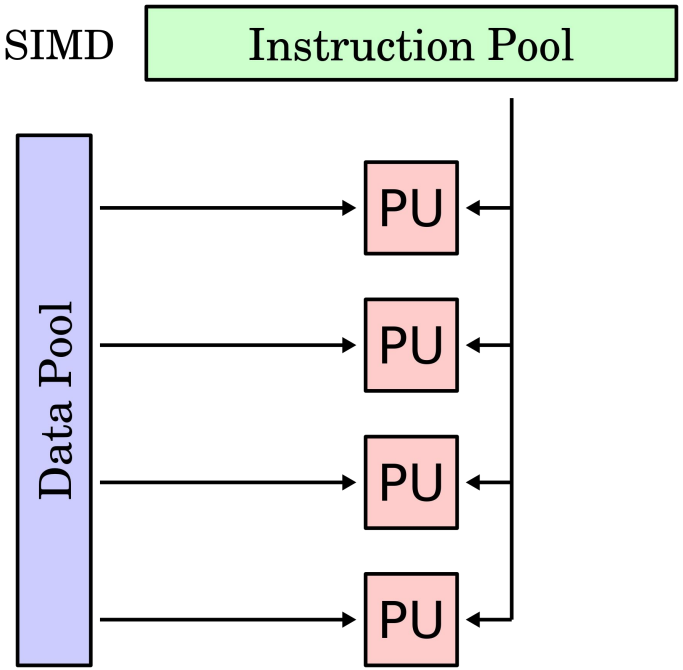
Parallel Programming Lab2-3
Oct 20 2022

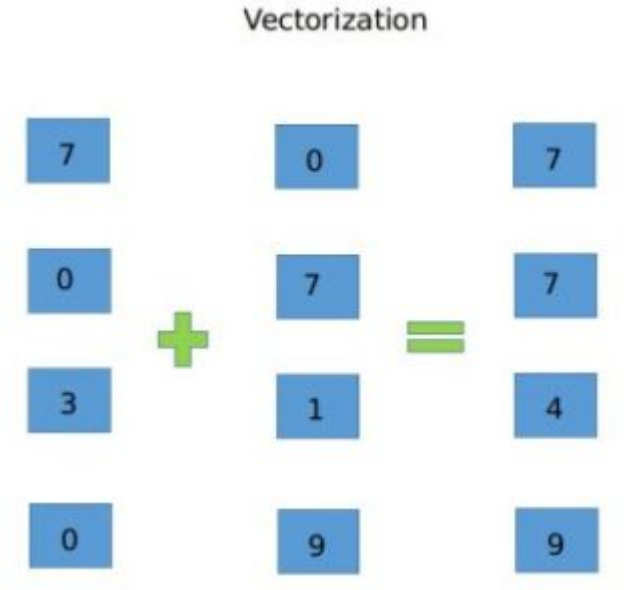
How CPU Execute an Instruction





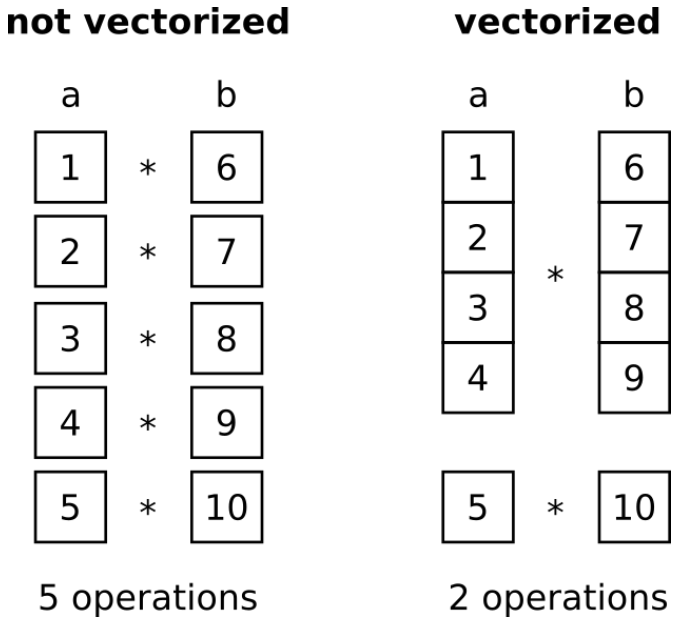
How CPU Execute an Instruction





Vector Instruction Set

- Instructions that design to operate on vectors.



When to use Vector Instruction Set?

- A part of codes that are executed many times
  - In a loop
- There are no data dependency between each iteration
  - Data dependency:
    - In iteration `i`, it needs the result of iteration `i-1` to calculate its result

When to use Vector Instruction Set?

- Vectorizable

```
void multiply_and_add(int *a, int *b, int *c, int *d, long long size) {
  for (long long i = 0; i < size; i++) {
    a[i] = b[i] * c[i] + d[i];
  }
}
```

  - Some observations
    - In each iteration, the result of a[i] is independent with each other
    - The same instructions are executed many times on different data

When to use Vector Instruction Set?

- Non-vectorizable

```
double norm2(int *a, long long size) {
  double result = 0.0;
  for (long long i = 0; i < size; i++) {
    result += a[i] * a[i];
  }
  return sqrt(result);
}

long long Fibonacci(long long ind) {
  long long *result = (long long*)malloc(sizeof(long long) * (ind + 2));
  result[1] = 1;
  result[2] = 1;
  for (int i = 3; i <= ind; i++) {
    result[i] = result[i - 1] + result[i - 2];
  }
  return result[ind];
}
```

Vector Instruction Sets

- MMX (64-bit)
- SSE (64~128-bit)
- FMA (128~256-bit)
- AVX (128~512-bit)

Vector Instruction Sets

- MMX
- SSE
  - Streaming SIMD Extensions
  - Versions: SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2
  - Calculate 128-bit data in an instruction (include integer & floating point)
    - 8x 16-bit short ⇐ 8x faster
    - 4x 32-bit integer ⇐ 4x faster
    - 4x 32-bit floating-point number ⇐ 4x faster
    - 2x 64-bit double-precision number ⇐ 2x faster

Vector Instruction Sets

- SSE
  - 16 registers (XMM00~XMM15)
    - SSE: only 32-bit floating point
    - SSE2: double, long long, int, char
  - Newer SSE only adds more kinds of instruction

Vector Instruction Sets

- FMA
  - Fused Multiply–Add
    - *a* = b * c + d
  - Versions: FMA4, FMA3
  - 128-bit and 256-bit of FMA operations
  - FMA4:
    - 4-operand instructions, only supported by AMD CPU
  - FMA3:
    - 3-operand instructions (*a* could only be *b*, *c*, or *d*)
      Ex: b = b * c + d

Vector Instruction Sets

- AVX
  - Advanced Vector eXtensions

Version	AVX, AVX2(extension of AVX)	AVX512
Width	256-bit vector instructions	512-bit vector instructions
Register	16 (YMM00~YMM15)	32 (ZMM00~ZMM31)

- More bits: More data could be calculated at a time
- More registers: More complex calculation is supported

Check Hardware Support

- Linux command: `lscpu | grep -i $instruction_set`
  - where `$instruction_set` could be mmx, sse, sse2, sse3, ssse3, sse4_1, sse4_2, avx, avx2, avx512
- Most CPUs released after 2011 support AVX instructions

```
$ lscpu | grep sse
Flags:                           fpu vme de pse tsc msr pae mce cx8 apic sep mtr
r pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall
 nx pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology
 nonstop_tsc cpuid aperfmperf pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm
2 ssse3 cx16 xtpr pdcm pcid dca sse4_1 sse4_2 popcnt aes lahf_lm epb pti ssbd ib
rs ibpb stibp tpr_shadow vnmi flexpriority ept vpid dtherm ida arat flush_l1d
```

Automatic Vectorization

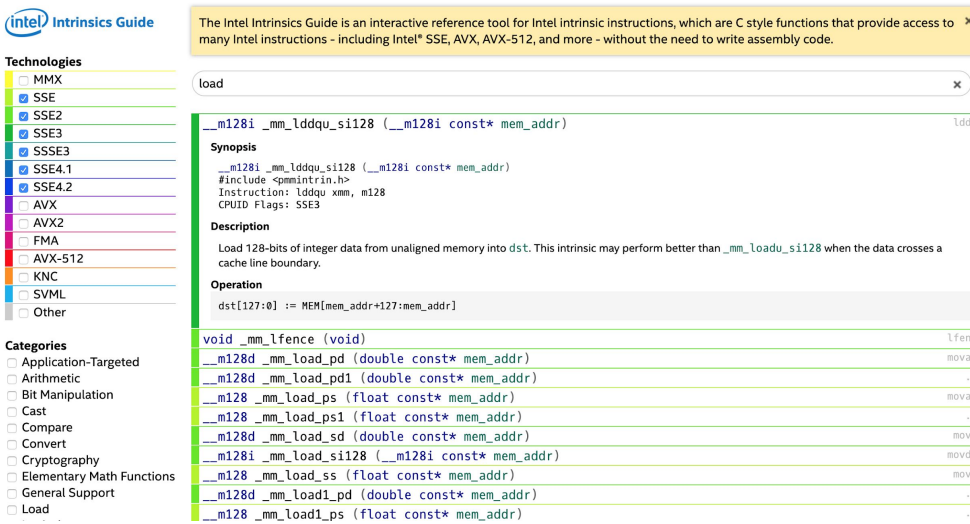
- GCC
  - Vectorization is enabled by the flag `-ftree-vectorize`
    - Enabled by default with flag `-O3`
  - Add flag `-march=native` to use instructions supported by the local CPU
  - Add compiler flag `-fopt-info-vec-all` to see vectorization log
  - Add `#pragma GCC ivdep` to code to declare that there is no data dependency in the following loop

Automatic Vectorization

```
mpicc -O3 -march=native -fopt-info-vec-all array.c -o array
array.c:8:3: optimized: loop vectorized using 16 byte vectors
array.c:8:3: optimized:  loop versioned for vectorization because of possible aliasing
array.c:7:6: note: vectorized 1 loops in function.
array.c:32:3: optimized: loop vectorized using 16 byte vectors
array.c:32:3: optimized:  loop versioned for vectorization because of possible aliasing
array.c:16:3: missed: couldn't vectorize loop
/usr/lib/gcc/x86_64-pc-linux-gnu/9.1.0/include/emmintrin.h:703:10: missed: not vectorize
d: no vectype for stmt: _40 = MEM[(const __m128i_u * {ref-all})_3];
 scalar_type: const __m128i_u
array.c:13:6: note: vectorized 1 loops in function.
```

- See more: link
- LLVM Compiler: link

Vectorize Loop with Intel Intrinsics



- Intel Intrinsics Guide: link
  a. Check the instruction set you want to use
  b. Use keyword to search
  c. Check the variable type & operation
- Procedure
  a. Load data from memory to the special registers
  b. Perform vector instructions
  c. Save data from the special registers to memory

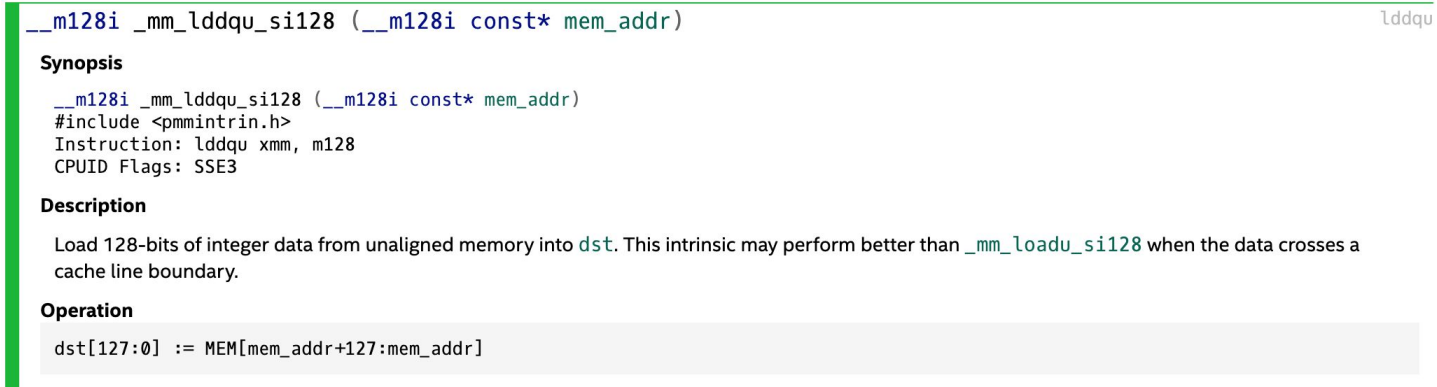
Vectorize Loop with Intel Intrinsics

- Original

```
void multiply_and_add(int *a, int *b, int *c, int *d, long long size) {
  for (long long i = 0; i < size; i++) {
    a[i] = b[i] * c[i] + d[i];
  }
}
```

Vectorize Loop with Intel Intrinsics

1. Check CPU support
   - Up to SSE4.2, no FMA (on apollo)
2. Load data from memory to the special registers
   - Use 128-bit instruction set & integer ⇒ __m128i
   - Load data (check Intel Intrinsics Guide) ⇒ _mm_lddqu_si128



```
__m128i _mm_lddqu_si128 (__m128i const* mem_addr)                    lddqu
```

Synopsis

```
__m128i _mm_lddqu_si128 (__m128i const* mem_addr)
#include <pmmintrin.h>
Instruction: lddqu xmm, m128
CPUID Flags: SSE3
```

Description

Load 128-bits of integer data from unaligned memory into `dst`. This intrinsic may perform better than `_mm_loadu_si128` when the data crosses a cache line boundary.

Operation

```
dst[127:0] := MEM[mem_addr+127:mem_addr]
```

Vectorize Loop with Intel Intrinsics

3. Perform vector instructions
   - No FMA ⇒ cannot do multiplication and addition in 1 instruction
   - Perform multiplication first, then addition

`__m128i _mm_mullo_epi32 (__m128i a, __m128i b)` pmulld

Synopsis

```
__m128i _mm_mullo_epi32 (__m128i a, __m128i b)
#include <smmintrin.h>
Instruction: pmulld xmm, xmm
CPUID Flags: SSE4.1
```

Description

Multiply the packed 32-bit integers in a and b, producing intermediate 64-bit integers, and store the low 32 bits of the intermediate integers in dst.

Operation

```
FOR j := 0 to 3
        i := j*32
        tmp[63:0] := a[i+31:i] * b[i+31:i]
        dst[i+31:i] := tmp[31:0]
ENDFOR
```

`__m128i _mm_add_epi32 (__m128i a, __m128i b)`

Synopsis

```
__m128i _mm_add_epi32 (__m128i a, __m128i b)
#include <emmintrin.h>
Instruction: paddd xmm, xmm
CPUID Flags: SSE2
```

Description

Add packed 32-bit integers in a and b, and store the results in dst.

Operation

```
FOR j := 0 to 3
        i := j*32
        dst[i+31:i] := a[i+31:i] + b[i+31:i]
ENDFOR
```

Vectorize Loop with Intel Intrinsics

4. Save data from the special registers to memory

```
void _mm_store_si128 (__m128i* mem_addr, __m128i a)
```

Synopsis

```
void _mm_store_si128 (__m128i* mem_addr, __m128i a)
#include <emmintrin.h>
Instruction: movdqa m128, xmm
CPUID Flags: SSE2
```

Description

Store 128-bits of integer data from `a` into memory. `mem_addr` must be aligned on a 16-byte boundary or a general-protection exception may be generated.

Operation

```
MEM[mem_addr+127:mem_addr] := a[127:0]
```

```
14 void multiply_and_add_sse42(int *a, int *b, int *c, int *d, long long size) {
15   long long size1 = (size >> 2) << 2;
16   long long i = 0;
17   for (; i < size1; i += 4) {
18     // 1. Declare variable to store 128-bit data with 4 integers
19     __m128i as;
20     __m128i bs = _mm_lddqu_si128((__m128i const*)&b[i]);
21     __m128i cs = _mm_lddqu_si128((__m128i const*)&c[i]);
22     __m128i ds = _mm_lddqu_si128((__m128i const*)&d[i]);
23
24     // 2. Perform SSE multiply & add instructions
25     as = _mm_mullo_epi32(bs, cs);
26     as = _mm_add_epi32(as, ds);
27
28     // 3. Store result back to memory
29     _mm_store_si128((__m128i*)&a[i], as);
30   }
31
32   // Dealing with the remaining parts
33   for (long long i = size1; i < size; i++) {
34     a[i] = b[i] * c[i] + d[i];
35   }
36 }
```

Promise the vectorized loop is aligned to 128 bits

Dealing with the remaining data

Hint

- Theoretically, use SIMD instruction can improve performance
  - With SSE instruction, it could be about 2x~8x faster
- SSE~AVX2 intrinsics is supported by gcc, but only Intel Compiler supports AVX512 intrinsics
  - Try it by yourself!!!
- Take advantage of auto-vectorization
- You can use any intrinsics in your homework