Parallel and Distributed Programming
SIMD

Kenjiro Taura

2026/11/09

Contents

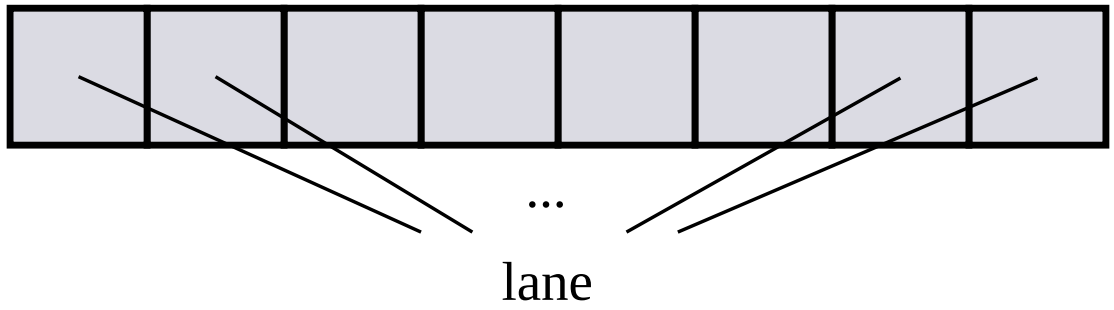
SIMD Instructions 2
SIMD programming alternatives 11
High-level approaches 13
Low-level SIMD Programming 39
Vector intrinsics 49

SIMD Instructions

SIMD : basic concepts

- **SIMD** : single instruction multiple data
- a ***SIMD register*** (or a ***vector register***) can hold several (typically 2 - 16) values of a type
- a ***SIMD instruction*** is an instruction that can apply (typically the same) operation on all or some values on a SIMD register(s)
- each value in a SIMD register is called a ***SIMD lane*** or simply a ***lane***
- they are indispensable tools for CPUs to get performance

A SIMD register



Evolving SIMD instruction sets

- recent processors rely on SIMD as an energy efficient way to boost peak FLOPS

Microarchitecture	ISA	vector width (SP)	throughput (per clock)	max SP flops/cycle /core
Intel Haswell	AVX2	8	2 FMAs	32
Intel Ice Lake	AVX-512	16	2 FMAs	64
Fujitsu A64FX	SVE	16	2 FMAs	64
Arm Neoverse V2 (Grace)	**NEON / SVE**	**4**	**4 FMAs**	**32**

- **ISA** : Instruction Set Architecture
- **vector width** : the number of single precision (SP) operands per SIMD register
- **FMA** : fused multiply-add instruction
- peak SP FLOPS of Grace (72 cores / 3.3GHz) $= 32 \times 3.3 \times 72 \approx 7.6$ TFLOPS
- no SIMD? $\rightarrow$ can tap *at most 1/4* of SP peak performance on Grace (at most 1/16 on machines having AVX-512)

Two SIMD instruction sets of Arm

- **NEON** (official name: Advanced SIMD)
  - the traditional SIMD instruction set of Arm
  - registers are 128 bits (4 SPs, 2 DPs, 4 ints, 2 longs, …)
- **SVE** (Scalable Vector Extension; and its extension SVE2)
  - the vector length (VL) is *not* fixed by the instruction set; anything from 128 to 2048 bits, depending on the processor (A64FX: 512 bits, Grace: **128 bits**)
  - the same machine code runs on processors with different VLs
  - **predicates** (masks) for almost all instructions
  - **gather / scatter** (load from / store to non-contiguous addresses)
- Grace supports both

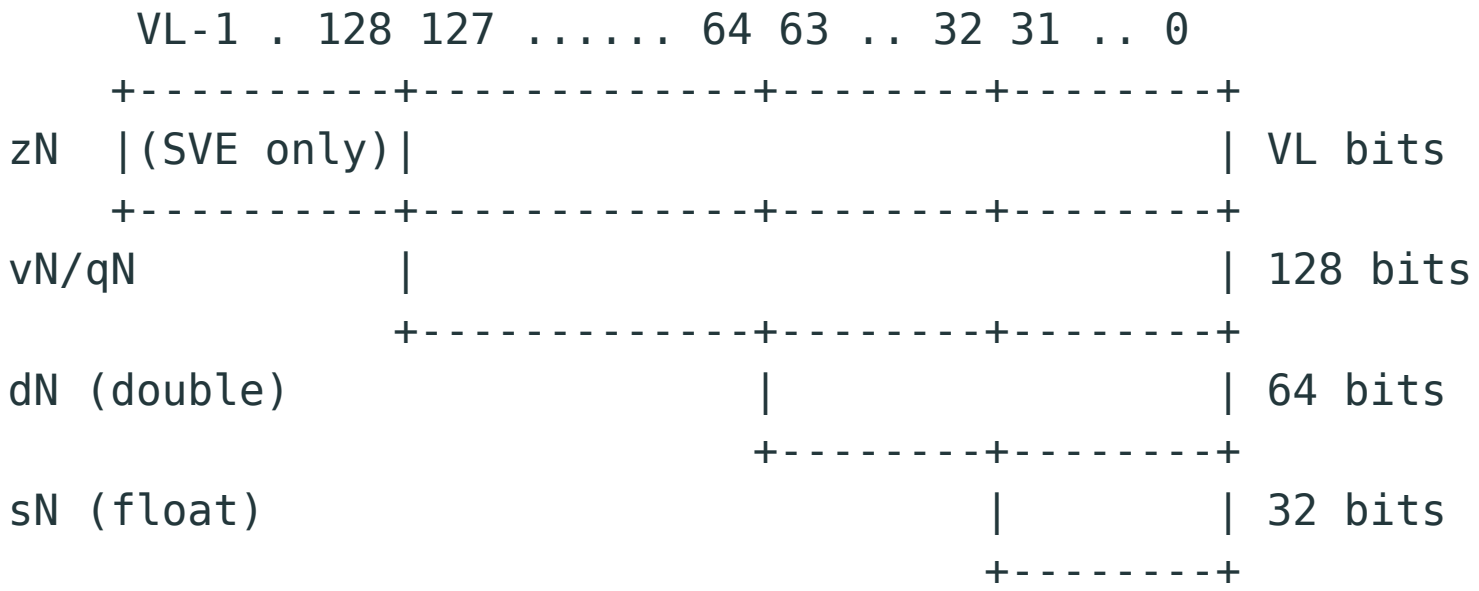
Arm SIMD instructions at a glance

operation	syntax	C-like expression
multiply	`fmul v2.4s, v1.4s, v0.4s`	`v2 = v1 * v0`
add	`fadd v2.4s, v1.4s, v0.4s`	`v2 = v1 + v0`
fma (NEON)	`fmla v2.4s, v1.4s, v0.4s`	`v2 += v1 * v0`
fma (SVE)	`fmad z2.s, p0/m, z1.s, z0.s`	`z2 = z2 * z1 + z0`
load	`ldr q0, [x0, #256]`	`v0 = *(x0+256)`
store	`str q0, [x0, #256]`	`*(x0+256) = v0`
load (SVE)	`ld1w {z0.s}, p0/z, [x0, x8, lsl #2]`	`z0 = *(x0+x8*4)`

- **`v0` ... `v31`** are 128 bit registers (NEON); each can hold
  - 4 single-precision (`float` of C; 32 bits) or
  - 2 double-precision (`double` of C; 64 bits) floating point numbers
- **`z0` ... `z31`** are VL bit registers (SVE); `p0` … `p15` are predicate registers (one bit per byte of `z` registers)
- *the 1st operand is typically the destination*

Registers and their aliases

- qi, di, si, hi, bi, vi and zi are names for (parts of) *the same register i*



- scalar FP registers si (float) and di (double) are the lowest lane of SIMD registers
- on Grace, VL = 128, so zi and vi are exactly the same bits

Telling scalar and vector instructions apart

- look at the register names and their suffixes to know what an instruction operates on

	operands	vector /scalar?	ISA
`fmul s2, s1, s0`	1 SP	scalar	
`fmul d2, d1, s0`	1 DP	scalar	
`fmul v2.4s, v1.4s, v0.4s`	4 SPs	vector	NEON
`fmul v2.2d, v1.2d, v0.2d`	2 DPs	vector	NEON
`fmul z2.s, p0/m, z2.s, z1.s`	VL/32 SPs	vector	SVE
`fmul z2.d, p0/m, z2.d, z1.d`	VL/64 DPs	vector	SVE

- NEON : vi.*count size* (`4s` = 4 × 32 bits, `2d` = 2 × 64 bits, `16b` = 16 × 8 bits)
- SVE : zi.*size* without count (it depends on VL)
- `s` = 32 bits and `d` = 64 bits; they say nothing about float or int
- see the cheat sheet for more (Grace assembly cheat sheet)

A Conceptual “Vectorization Algorithm”

- As you can see, SIMD is good for computations doing ***almost exactly*** the same thing on many data ⇒ loops are the typical target
- “Vectorization” is basically about transforming a loop

```
for (i = 0; i < n; i++) {
  S(i);
}
```

into a loop that executes “L iterations at the same time”

```
for (i = 0; i + L <= n; i += L) {
  S(i:i+L);
}
```

(we assume L divide n for simplicity)

A Simple Example

```
for (i = 0; i < n; i++) {
  x[i] = a * x[i] + b
}
```

⇒

```
for (i = 0; i < n; i += L) {
  x[i:i+L] = a * x[i:i+L] + b
}
```

expression	corresponding SIMD operation	possible?
`x[i:i+L]`	a vectorized load of contiguous elements	✅
`a * x[i] + b`	a vectorized multiply-add	✅
`x[i:i+L] = ...`	a vectorized store of contiguous elements	✅

SIMD programming alternatives

Several ways to use SIMD

- “high-level” approaches (≈ write vectorizable loops)
  - loop + **auto vectorization**
  - loop + SIMD directives
  - array expressions
- “low-level” approaches (≈ manipulate “vector of values”)
  - vector types
    - **vector types for C/C++ (GNU vector extension)**
    - `std::simd` (C++26)
  - **intrinsics**
  - assembly programming

High-level approaches

Auto vectorization of loops

- write scalar loops and hope the compiler does the job
- e.g.,

```
void axpb_auto(float a, float * x, float b, int n) {
  for (int i = 0; i < n; i++) {
    x[i] = a * x[i] + b;
  }
}
```

- compile and run normally

```
$ clang++ -O3 -mcpu=neoverse-v2 -o simd_auto simd_auto.cc
```

- **`-mcpu=neoverse-v2`** says “should use instructions of Neoverse V2”
- see notebook for details

How to know if the compiler vectorized it?

- options useful to know whether a loop is successfully vectorized and if not, why not

	report options
Clang	`-R{pass,pass-missed}=loop-vectorize`
NVIDIA	`-M{info,neginfo}=vect`
GCC	`-fopt-info-vec-{optimized,missed}`

- ***but don't hesitate to dive into assembly code***
  - make `-S` option your friend
  - a trick: ***enclose loops with inline assembler comments to locate assembly for the loop***

```
asm volatile ("# xxxxxx loop begins");
for (i = 0; i < n; i++) {
  ... /* hope to be vectorized */
}
asm volatile ("# xxxxxx loop ends");
```

What a vectorized loop looks like

```
.LBB0_6:
  ldp   q3, q4, [x11, #-32]        // load x[i:i+8]
  ldp   q5, q6, [x11]              // load x[i+8:i+16]
  mov   v7.16b, v2.16b             // v7 = b
  subs  x12, x12, #16              // 16 elements per iteration
  fmla  v7.4s, v3.4s, v0.s[0]      // v7 += x[i:i+4] * a
  mov   v3.16b, v2.16b
  fmla  v3.4s, v4.4s, v0.s[0]
  ...
  stp   q7, q3, [x11, #-32]        // store x[i:i+8]
  stp   q4, q5, [x11], #64         // store x[i+8:i+16]
  b.ne  .LBB0_6
```

- ***128-bit register names (v, z, q) are the hallmarks***

Obstacles for auto-vectorization

- the premise of auto-vectorization is it does not change the semantics of the program
  $\Rightarrow$ (potential) loop-carried dependencies prevent it

Dependencies

- vectorization changes the order in which statements are executed

```
for (i = 0; i < n; i++) {
  x[i+1] += x[i];
}
```

≡

```
x[1] += x[0]; // x[1] = x1 + x0
x[2] += x[1]; // x[2] = x2 + x1 + x0
x[3] += x[2]; // x[3] = x3 + x2 + x1 + x0
x[4] += x[3]; // x[4] = x4 + x3 + x2 + x1 + x0
  ...
```

- when vectorized with e.g., $L = 4$

```
for (i = 0; i < n; i += 4) {
  x[i+1:i+5] += x[i:i+4];
}
```

≡

```
x0,x1,x2,x3 = x[i:i+4]
x[1] += x0 // x[1] = x1 + x0
x[2] += x1 // x[2] = x2 + x1
x[3] += x2 // x[3] = x3 + x2
x[4] += x3 // x[4] = x4 + x3
```

- this changes the behavior of the program ⇒ the compiler gives up *auto* vectorization

Potential dependencies due to ambiguous aliasing

- *auto*-vectorizer must play safe when vectorization *may* change the behavior of the loop. e.g.,

```
for (i = 0; i < n; i++) {
  y[i] += x[i];
}
```

  what if, say, `&y[i] = &x[i+1]`?
- it is safe if `y` and `x` never overlaps, but the compiler is seldom able to *prove* it
- good compilers (clang included) generate ***both*** a vectorized loop and a scalar loop, and check ***at runtime*** whether `x` and `y` overlap (***loop versioning***)
- the compiler gives up when there are too many pointers to check

`restrict` keyword

- you can help the compiler by saying “they never overlap” (more precisely, “never mind even if they do”)
- annotate parameters of pointer type with **`restrict`** (C99; `__restrict__` in C++) to indicate that

```
void axpb_auto(float a, float * restrict x, float b,
               float * restrict y, int n) {
  for (int i = 0; i < n; i++) {
    y[i] = a * x[i] + b;
  }
}
```

- the versioning and runtime check disappear
- **`#pragma omp simd`** has the same effect (next)

OpenMP SIMD constructs

- **`#pragma omp simd`**
  - directive to vectorize for loops
  - the effect: say *"compiler should vectorize and I accept consequences"*
  - syntax restrictions similar to `omp for` pragma apply
- **`#pragma omp declare simd`**
  - instructs the compiler to generate vectorized versions of a function
  - with it, loops with function calls can be vectorized
- compiler-specific alternatives without OpenMP exist (`#pragma GCC ivdep`, `#pragma clang loop vectorize(assume_safety)`, `#pragma loop ivdep` of nvc), but `omp simd` is the one common to all of them

`simd` pragma

- basic syntax (similar to `omp for`):

```
#pragma omp simd [clauses]
for (i = ...; i < ...; i += ...)
    S
```

- optional clauses:
  - `aligned(var,var,...:align)`
  - `uniform(var,var,...)` says variables are loop invariant
  - `linear(var,var,...:stride)` says variables have the specified stride between consecutive iterations
  - `reduction(op:var,...)`

`simd` pragma example

```
void axpb_omp(float a, float * x, float b, int n) {
#pragma omp simd
  for (int i = 0; i < n; i++) {
    y[i] += x[i];
  }
}
```

- the effect is the compiler does not worry about potential aliasing between `x` and `y`

`simd` pragma for reductions

- computing an inner product:

```
float inner_omp(float * x, float * y, int n) {
  float c = 0;
#pragma omp simd reduction(+:c)
  for (int i = 0; i < n; i++) {
    c += x[i] * y[i];
  }
  return c;
}
```

- without the pragma `reduction(+:c)`, the compiler finds dependency via `c` so gives up vectorization

`declare simd` pragma

- given to function definition or declaration

```
#pragma omp declare simd [clauses]
int f(int x) { ... }
```

- when given before a function definition, generates a vectorized version of the function body
- when given before a function declaration, tells the compiler a vectorized version of the function is available
- optional clauses
  - those for `simd` pragma
  - `notinbranch`
  - `inbranch`
- as of this writing, gcc supports it but clang does not

Expressions/statements compilers work harder to vectorize or fail to do so

- **non-contiguous data accesses in the loop**
- **conditionals in the loop** (`if` statements)
- **loops in the loop**
- **function calls in the loop**

Non-contiguous data accesses

- ordinary vector load/store instructions access contiguous addresses

  `ldr q0, [x0]`

  loads `q0` (= `v0`) with the contiguous 16 bytes from address `x0`

- → they can be used only when consecutive iterations access consecutive addresses

✅ contiguous

```
#pragma omp simd
for (int i = 0; i < n; i++) {
  y[i] = a * x[i] + b;
}
```

❌ non-contiguous

```
#pragma omp simd
for (int i = 0; i < n; i++) {
  y[i] = a * x[2 * i] + b;
}
```

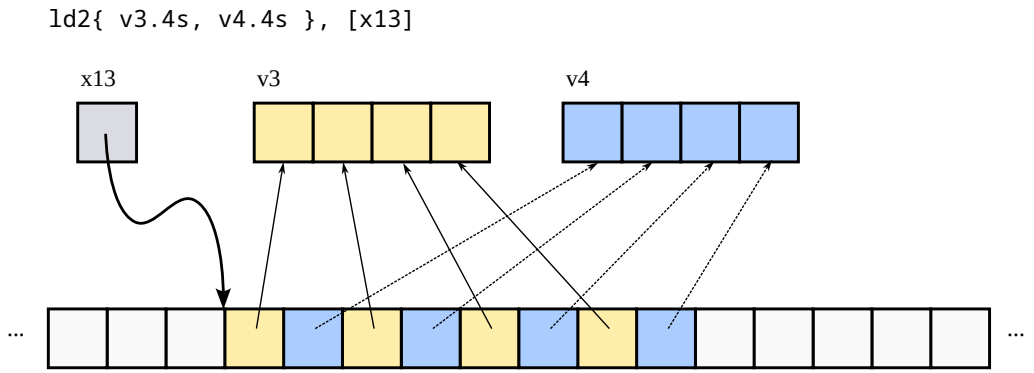
- two strategies for non-contiguous access patterns
  - load contiguous data and shuffle them on registers
  - load directly from non-contiguous addresses (**gather**)

Small strided accesses → de-interleaving loads

```
#pragma omp simd
for (int i = 0; i < n; i++) {
  y[i] = a * x[2 * i] + b;
}
```

- vectorized with NEON **`ld2`** (stride-2 access)

```
ld2   { v3.4s, v4.4s }, [x13]     // [x13+0, 8,16,24] -> v3
                                  // [x13+4,12,20,28] -> v4
```



- `ld3` / `ld4` exist for strides 3 / 4

Irregular accesses → gather

- when index is completely unpredictable, like

```
#pragma omp simd
  for (int i = 0; i < n; i++) {
    y[i] = a * x[idx[i]] + b;
  }
}
```

- SVE has ***gather*** instructions

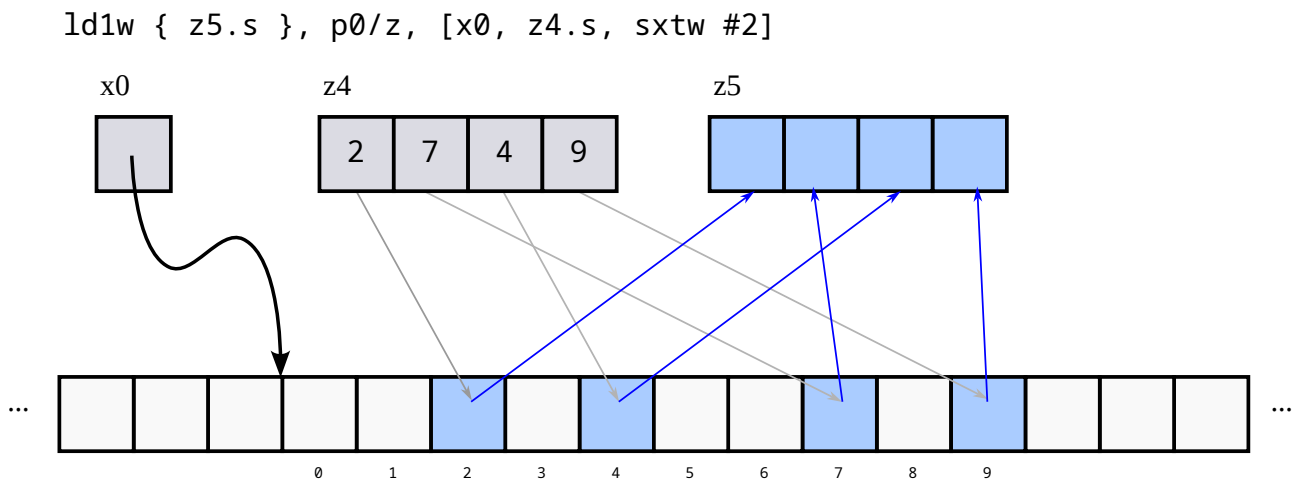
Gather instruction

- e.g.,

```
ld1w { z5.s }, p0/z, [x0, z4.s, sxtw #2] ≡

z5.s[0] = [x0 + (z4.s[0] << 2)] if p0[0] else 0
z5.s[1] = [x0 + (z4.s[1] << 2)] if p0[1] else 0
z5.s[2] = [x0 + (z4.s[2] << 2)] if p0[2] else 0
z5.s[3] = [x0 + (z4.s[3] << 2)] if p0[3] else 0
```

- `x0` : base address, `z4` : L indices

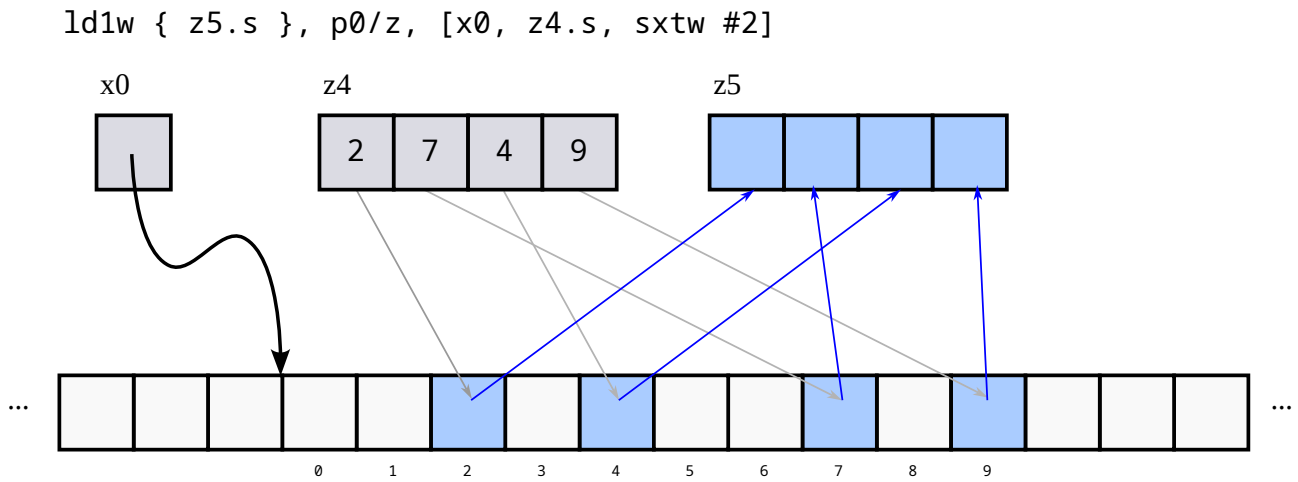


Non-contiguous stores → scatter

- a similar instruction for non-contiguous stores

```
st1w { z5.s }, p0, [x0, z4.s, sxtw #2] ≡

[x0 + z4.s[0] << 2] = z5.s[0] if p0.s[0]
[x0 + z4.s[1] << 2] = z5.s[1] if p0.s[1]
[x0 + z4.s[2] << 2] = z5.s[2] if p0.s[2]
[x0 + z4.s[3] << 2] = z5.s[3] if p0.s[3]
```



Conditionals → predicates

- a conditional execution (e.g., if statement) within an iteration requires a statement to be executed only for a part of SIMD lanes

```
#pragma omp simd
for (int i = 0; i < n; i++) {
  if (x[i] > 0.0) {
    x[i] = a * x[i] + b;
  }
}
```

- → ***predicated*** instructions

Predicates

- a predicate register (`p0` … `p15`) holds one boolean per lane
- an instruction “predicated” by `p` operates only on lanes where `p` is true; for other lanes,

notation	lanes where the predicate is false
`p0/z`	set to zero (zeroing)
`p0/m`	keep the old value of the destination (merging)
`p0` (store)	not written to memory

- template for vectorizing loops containing if-statements

```
for (...) {
  if (C)
    T
  else
    E
}
```

→

```
p = C[i:i+L]
T[i:i+L] on  p[i:i+L]
E[i:i+L] on !p[i:i+L]
```

Predicates

```
#pragma omp simd
for (int i = 0; i < n; i++) {
  if (x[i] > 0.0) {
    x[i] = a * x[i] + b;
  }
}
```

→

```
for (int i = 0; i < n; i += L) {
  p = x[i:i+L] > 0.0; // L-bit mask
  x[i:i+L] = a * x[i:i+L] + b; on p[i:i+L]
}
```

↓

```
ldr    z4, [x10]                  // z4 = x[i:i+4]
fcmgt  p1.s, p0/z, z4.s, #0.0     // p1 = (x[i:i+4] > 0.0)
fmad   z4.s, p0/m, z3.s, z2.s     // z4 = z4 * a + b (for ALL lanes)
st1w   { z4.s }, p1, [x10]        // store only lanes where p1 is true
```

Loops inside the loop

- there should be no fundamental difficulty in vectorizing this

```
#pragma omp simd
for (int i = 0; i < n; i++) {
  for (int k = 0; k < m; k++) {
    x[i] = a * x[i] + b;
  }
}
```

→

```
for (int i = 0; i < n; i += L) {
  for (int k = 0; k < m; k++) {
    x[i:i+L] = a * x[i:i+L] + b;
  }
}
```

- clang/gcc both fail to vectorize this (they appear to only vectorize ***innermost*** loops)

Loops inside the loop

- trick: you can make it the innermost loop yourself

```
for (int i = 0; i < n; i += L) {
  for (int k = 0; k < m; k++) {
    x[i:i+L] = a * x[i:i+L] + b;
  }
}
```

→

```
for (int i = 0; i < n; i += L) {
  for (int k = 0; k < m; k++) {
#pragma omp simd
    for (int ii = i; ii < i + L; ii++) {
      x[ii] = a * x[ii] + b;
    }
  }
}
```

Function calls

- what if an iteration has an unknown function call (`f` below)?

```
#pragma omp simd
for (int i = 0; i < n; i++) {
  x[i] = f(a, x[i], b);
}
```

- vectorization is impossible/ineffective unless `f` has been vectorized elsewhere
- `#pragma omp declare simd` allows you to declare exactly that

```
#pragma omp declare simd uniform(a, b) notinbranch
float f(float a, float x, float b);
```

- the loop then calls the vectorized version
- implementation status:
  - clang does not support it (gcc does)

High level vectorization: summary and takeaway

- CPUs (especially recent ones) have necessary tools
  - arithmetic → vector arithmetic instructions
  - load → vector load, de-interleaving load and gather instructions
  - store → vector store and scatter instructions
  - if → predicated executions
- compilers (e.g., clang on Grace) now use many of them
  - runtime alias checks, predicates for `if`, `ld2` for strides, gathers for indirect accesses
- still weak at
  - loops and function calls in the loop
- whether the compiler uses them is still something you cannot predict
- become a friend of compiler reports and assembly (`-S`)

Low-level SIMD Programming

Low-level SIMD Programming

- in low-level SIMD programming, programs directly manipulate ***vector of values***
- useful/necessary when you want to be absolutely sure it is vectorized
- the first step is to define ***a type representing them***

Defining vector types

- the following defines `floatv` type, which is 16 bytes = 4 floats

```
typedef float floatv __attribute__((vector_size(16)));
```

Where vector types are convenient

1. you can use familiar arithmetic expressions on vector types and you are almost certain it uses SIMD instruction

```
floatv x, y, z;
z += x * y;
```

2. you can mix scalars and vectors (scalars are implicitly expanded/broadcast to all lanes)

```
float a, b;
floatv x, y;
y = a * x + b;
```

3. dereferencing a pointer to a vector type becomes vectorized load/store

```
floatv * p = ...;
  ... *p ...
  ... p[i] ...
  *p = ...
  p[i] = ...
```

Where vector types are convenient

4. comparison (e.g., `a < b`) becomes a predicate and ternary condition expression (k `?` T `:` E) blends values from T and E

   ```
   floatv a, b;
   floatv m = a < b ? a : b;
   ```

5. the size actually does not have to match the native vector size; with `vector_size(64)`, the compiler emits four NEON instructions for each operation (convenient!)

Note:

- for reasons I don’t get into, a safer definition is

```
typedef float floatv __attribute__((vector_size(16),
                                   __may_alias__,
                                   aligned(sizeof(float))));
```

- this allows us to get elements from ordinary array of `float` with pointers to `floatv`

An example vectorizing loop with vector extension

(assuming `L` divides `n` as always)

1. scalar code

```
for (int i = 0; i < n; i++) {
  y[i] = a * x[i] + b;
}
```

→ 2. pseudo code

```
for (int i = 0; i < n; i += L) {
  y[i:i+L] = a * x[i:i+L] + b;
}
```

→ 3. working code

```
for (int i = 0; i < n; i += L) {
  V(y[i]) = a * V(x[i]) + b;
}
```

where `V(p)` is a C++ function or a macro implementing `p[0:L]`

```
floatv& V(float& lv) { return *((floatv*)(&lv)); }  // C++
#define V(lv) (*((floatv*)&(lv)))                   // C
```

Dealing with remainder iterations

- when `L` does not divide `n`, run remainders after the vectorized version

```
int i;
for (i = 0; i + L <= n; i += L) {
  V(y[i]) = a * V(x[i]) + b;
}
for (     ; i < n; i++) {
  y[i] = a * x[i] + b;
}
```

- manually doing this is tedious …
- make n a multiple of L when the problem allows it (otherwise do the tedious work, or use SVE predicates; later)

Make a vector value from scalar value(s)

- you typically make a vector value from an array of scalars

```
float * a = ...;
floatv v = V(a[i]);
```

- or list the values

```
floatv v = { f0, f1, f2, f3 };
```

… and vice versa

- you typically store a vector value to an array of scalars

```
float * a = ...;
floatv v = ...;
V(a[i]) = v;
```

  and get individual scalars from the array
- you can access a particular lane of a vector directly, as if a vector is a C array. e.g.,

```
floatv v;
float s = v[3];
```

- it is slow or rejected when index is not a compile-time constant. e.g.,

```
floatv v; int i = ...;
float s = v[i];
```

Vector intrinsics

Vector intrinsics

- what are *intrinsics*? → processor/platform-specific low-level types and functions
- on Arm processors, put these in your code

```
#include <arm_neon.h>   // NEON
#include <arm_sve.h>    // SVE
```

  and you get
  - a set of available vector types
  - a lot of functions operating on vector types
- bookmark “Arm Intrinsics” (link) when using intrinsics; it covers both NEON and SVE

NEON intrinsics

- vector types : `<element type><bits>x<count>_t`
  - `float32x4_t` ≈ `float` × 4
  - `float64x2_t` ≈ `double` × 2
  - `int32x4_t`, `int64x2_t`, `uint32x4_t`, … (integers)
- functions : `v` + operation + `q` (128 bit) + `_` + element type
  - `float32x4_t vfmaq_f32(float32x4_t a, float32x4_t b, float32x4_t c)` ≡ `a + b * c`
  - `float32x4_t vdupq_n_f32(float f)` ≡ `{ f, f, f, f }`
  - `float32x4_t vld1q_f32(const float * p)`, `vst1q_f32(float * p, float32x4_t v)`
- *each function almost directly maps to a single assembly instruction*

SVE vector types

- vector types are *sizeless* (the size is unknown at compile time)
  - `svfloat32_t`, `svfloat64_t`, `svint32_t`, `svint64_t`, ...
  - `svbool_t` : a predicate
  - no `sizeof`, no arrays of them, no structure members, no global variables

SVE intrinsic functions

- functions : `sv` + operation + `_` + element type (+ `_m`/`_z`/`_x`)
  - most of them are ***predicated***
  - `svfloat32_t svmad_f32_x(svbool_t p, svfloat32_t a, svfloat32_t b, svfloat32_t c)`
    ≡ `a * b + c` on lanes p is true, and don't care about other lanes (`_x`); `_m`: merge; `_z`: zero
  - `svbool_t svwhilelt_b32(int i, int n)` ≡ `{ i < n, i + 1 < n, ... }`
  - `svfloat32_t svld1_f32(svbool_t p, const float * a)`
  - `svst1_f32(svbool_t p, float * a, svfloat32_t v)`
  - `uint64_t svcntw()` : the number of 32 bit elements in a vector (4 on Grace)
- arithmetic operators work on sizeless types too (`a * x + b`)

When to use intrinsics?

- when you have to :-)
- use it when you know
  - something should be possible at machine code level (e.g., you know vectorized `sqrt` instruction exists),
  - but neither compiler nor vector type do the job