SIMD Programming

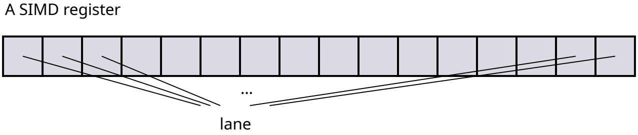
Kenjiro Taura

Contents

Contents

SIMD : basic concepts

- SIMD : single instruction multiple data
- a *SIMD register* (or a *vector register*) can hold many values (2 - 16 values or more) of a single 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



Evolving Intel instruction set

- Recent processors increasingly 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
Nehalem	SSE	4	1 add + 1 mul	8
Sandy Bridge	AVX	8	1 add + 1 mul	16
Haswell	AVX2	8	2 fmas	32
Ice Lake	AVX-512	16	2 fmas	64

- ISA : Instruction Set Architecture
- vector width : the number of single precision (SP) operands
- fma : fused multiply-add instruction
- e.g., Peak FLOPS of a machine having 2 $\times$ Intel Xeon Gold 6130 (2.10GHz, 32 cores) = 8.6 TFLOPS
- no SIMD? $\rightarrow$ can tap *at most 1/16* of SP peak performance on machines having AVX-512

Intel SIMD instructions at a glance

Some example AVX-512F (a subset of AVX-512) instructions

operation	syntax	C-like expression
multiply	`vmulps %zmm0,%zmm1,%zmm2`	`zmm2 = zmm1 * zmm0`
add	`vaddps %zmm0,%zmm1,%zmm2`	`zmm2 = zmm1 + zmm0`
fmadd	`vfmadd132ps %zmm0,%zmm1,%zmm2`	`zmm2 = zmm0*zmm2+zmm1`
load	`vmovups 256(%rax),%zmm0`	`zmm0 = *(rax+256)`
store	`vmovups %zmm0,256(%rax)`	`*(rax+256) = zmm0`

- `zmm0` ... `zmm31` are 512 bit registers; each can hold
  - 16 single-precision (`float` of C; 32 bits) or
  - 8 double-precision (`double` of C; 64 bits)

    floating point numbers
- `XXXps` stands for *packed single precision*

xmm, ymm and zmm registers

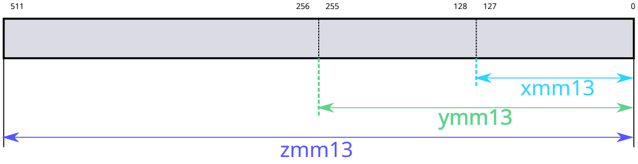
- ISA and available registers

ISA	registers
SSE	`xmm0, ...xmm15`
AVX	`{x,y}mm0, ...{x,y}mm15`
AVX-512	`{x,y,z}mm0, ...{x,y,z}mm31`

- registers and their widths *(vector widths)*

register names	register width (bits)
xmm$i$	128
ymm$i$	256
zmm$i$	512

- xmm$i$, ymm$i$ and zmm$i$ are *aliased*



Intel SIMD instructions at a glance

- look at register names (x/y/z) and the last two characters of a mnemonic (p/s and s/d) to know what an instruction operates on

	operands	vector /scalar?	ISA
`vmulss %xmm0,%xmm1,%xmm2`	1 SPs	scalar	SSE
`vmulsd %xmm0,%xmm1,%xmm2`	1 DPs	scalar	SSE
`vmulps %xmm0,%xmm1,%xmm2`	4 SPs	vector	SSE
`vmulpd %xmm0,%xmm1,%xmm2`	2 DPs	vector	SSE
`vmulps %ymm0,%ymm1,%ymm2`	8 SPs	vector	AVX
`vmulpd %ymm0,%ymm1,%ymm2`	4 DPs	vector	AVX
`vmulps %zmm0,%zmm1,%zmm2`	16 SPs	vector	AVX-512
`vmulpd %zmm0,%zmm1,%zmm2`	8 DPs	vector	AVX-512

- ...ss : scalar *single* precision
- ...sd : scalar *double* precision
- ...ps : packed *single* precision
- ...pd : packed *double* precision

Applications/limitations of SIMD

- SIMD is good at parallelizing computations doing *almost exactly* the same series of instructions on contiguous data
- ⇒ generally, main targets are simple loops whose index values can be easily identified

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

⇒

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

L is the SIMD width

Contents

Several ways to use SIMD

- auto vectorization
  - loop vectorization
  - basic block vectorization
- language extensions/directives for SIMD
  - SIMD directives for loops (OpenMP 4.0/OpenACC)
  - SIMD-enabled functions (OpenMP 4.0/OpenACC)
  - array languages (Cilk Plus)
  - specially designed languages
- vector types
  - vector types for C/C++
  - Boost.SIMD
- intrinsics
- assembly programming

Contents

Auto loop vectorization

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

```
void axpy_auto(float a, float * x, float c, long m) {
  for (long j = 0; j < m; j++) {
    x[j] = a * x[j] + c;
  }
}
```

- compile and run

```
$ clang -o simd_auto -mavx512f -mfma -O3 simd_auto.c
```

- `-mavx512f -mfma` say "should use AVX-512F and FMA instructions" (better to be explicit for the time being)
- `-O3` increases the optimization level (so the compiler should work hard to vectorize it)
- read the notebook about options of other compilers (NVIDIA and GCC)

How to know if the compiler vectorized it?

- there are 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 easily locate assembly code for the loop*

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

Contents

OpenMP SIMD constructs

- `simd` pragma
  - directive to vectorize for loops
  - syntax restrictions similar to `omp for` pragma apply
- `declare simd` pragma
  - instructs the compiler to generate vectorized versions of a function
  - with it, loops with function calls can be vectorized

simd pragma

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

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

- 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

`simd` pragma

```
void axpy_omp(float a, float * x, float c, long m) {
#pragma omp simd
  for (long j = 0; j < m; j++) {
    x[j] = a * x[j] + c;
  }
}
```

- note: there are no points in using `omp simd` here, when auto vectorization does the job
- in general, `omp simd` declares "you don't mind that the vectorized version is not the same as non-vectorized version"

simd pragma to vectorize programs explicitly

- computing an inner product:

```c
void inner_omp(float * x, float * y, long m) {
  float c = 0;
#pragma omp simd reduction(c:+)
  for (long j = 0; j < m; j++) {
    c += x[j] * y[j];
  }
}
```

- note that the above loop is unlikely to be auto-vectorized, due to dependency through `c`

`declare simd` pragma

- when given before a function definition, vectorizes a function body
- when given before a function declaration, tells the compiler a vectorized version of the function is available
- basic syntax (similar to `omp for`):

```
#pragma omp declare simd clauses
function definition or declaration
```

- clauses
  - those for `simd` pragma
  - `notinbranch`
  - `inbranch`

Reasons that a vectorization fails

- potential aliasing makes auto vectorization difficult/impossible
- complex control flows make vectorization impossible or less profitable
- non-contiguous data accesses make vectorization impossible or less profitable

  giving hints to the compiler sometimes (not always) addresses the problem

Aliasing and auto vectorization

- "auto" vectorizer succeeds only when the compiler can guarantee a vectorized version produces an *identical result* with a non-vectorized version
- vectorization of loops operating on two or more arrays is often invalid if they point to be the same array

```
for (i = 0; i < m; i++) {
  y[i] = a * x[i] + c;
}
```

what if, say, `&y[i]` $=$ `&x[i+1]`*?*

  - N.B., good compilers generate code that first checks x[i:i+L] and y[i:i+L] overlap

- if you know they don't overlap, you can make that explicit
- `restrict` keyword, introduced by C99, does just that

`restrict` keyword

- annotate parameters of pointer type with `restrict`, if you know they never point to the same data

```
void axpy_auto(float a, float * restrict x, float c,
          float * restrict y, long m) {
    for (long j = 0; j < m; j++) {
        y[j] = a * x[j] + c;
    }
}
```

- you need to specify `-std=gnu99` (C99 standard)

```
$ gcc -march=native -O3 -S a.c -std=gnu99 -fopt-info-vec-optimized
  ...
a.c:5: note: LOOP VECTORIZED.
a.c:1: note: vectorized 1 loops in function.
  ...
```

Control flows within an iteration — conditionals

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

```
void loop_if(float a, float * restrict x, float b,
             float * restrict y, long n) {
#pragma omp simd
  for (long i = 0; i < n; i++) {
    if (x[i] < 0.0) {
      y[i] = a * x[i] + b;
    }
  }
}
```

- AVX-512 supports *predicated execution (execution mask)* for that

Control flows within an iteration — nested loops

- a nested loop within an iteration causes a similar problem with conditional executions

```
void loop_loop(float a, float * restrict x, float b,
               float * restrict y, long n) {
#pragma omp simd
  for (long i = 0; i < n; i++) {
    y[i] = x[i];
    for (long j = 0; j < end; j++) {
      y[i] = a * y[i] + b;
    }
  }
}
```

- if *end* depends on i (SIMD lanes), it requires a predicated execution

Control flows within an iteration — function calls

- if an iteration has an unknown (not inlined) function call, almost no chance that the loop can be vectorized
  - the function body would have to be executed by scalar instructions anyways

```
void loop_fun(float a, float * restrict x, float b,
              float * restrict y, long n) {
#pragma omp simd
  for (long i = 0; i < n; i++) {
    f(a, x, b, y, i);
  }
}
```

- you can declare that `f` has a vectorized version with `#pragma omp declare simd` (with such a definition, of course)

```
#pragma omp declare simd uniform(a, x, b, y) linear(i:1) notinbranch
void f(float a, float * restrict x, float b, float * restrict y, long i);
```

Non-contiguous data accesses

- ordinary vector load/store instructions access a contiguous addresses

```
vmovups (a),%zmm0
```

  loads `zmm0` with the contiguous 64 bytes from address a

- → they can be used only when iterations next to each other access addresses next to each other

Non-contiguous data accesses

- that is, they cannot be used for

```
void loop_stride(float a, float * restrict x, float b,
                 float * restrict y, long n) {
#pragma omp simd
  for (long i = 0; i < n; i++) {
    y[i] = a * x[2 * i] + b;
  }
}
```

let alone

```
void loop_random(float a, float * restrict x, float b,
                 float * restrict y, long n) {
#pragma omp simd
  for (long i = 0; i < n; i++) {
    y[i] = a * x[i * i] + b;  // or x[idx[i]]
  }
}
```

- AVX-512 supports *gather* instructions for such data accesses

Non-contiguous stores

- what about store

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

- AVX-512 supports *scatter* instructions for such data accesses
- it is your responsibility to guarantee `idx[i:i+`L`]` do not point to the same element

High level vectorization: summary and takeaway

- CPUs (especially recent ones) have necessary tools
  - arithmetic → vector arithmetic instructions
  - load → vector load and gather instructions
  - store → vector store and scatter instructions
  - if and loops → predicated executions
- generally, the compiler is behind CPUs; whether the compiler is able to use them is another story
- become a friend of compiler reports and assembly (`-S`)

Contents

Vector types

- many compilers extend C by allowing you to define a type that explicitly represents a vector of values

```
typedef float floatv __attribute__((vector_size(64)));
```

- you can use familiar arithmetic expressoins on vector types

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

- Clang/NVIDIA/GCC allow you to mix scalars and vectors

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

- you can combine them with *intrinsics* I'll get to later
- for reasons I don't get into, a better definition is

```
typedef float floatv __attribute__((vector_size(64),
                                    __may_alias__,
                                    aligned(sizeof(float))));
```

An example using vector extension

- scalar code

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

- pseudo code (assume L | n (L divides n))

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

- a function or macro (V) implementing x[i:i+L]

```
/* take the address, cast it to (floatv*) and deref it */
#define V(lv) (*((floatv*)&(lv)))
```

- it is then

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

Dealing with remainder iterations

- when `L` $\nmid$ `n`, run remainders after the vectorized version

```
long 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)

Make a vector value from scalar value(s)

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

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

- a macro/function like the following makes the life better

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

with which we can write

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

… 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];
```

- but a CPU generally lacks instructions to access a lane designated by a value not known at the compile time. e.g.,

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

  it might be essentially doing the former each time you access an element, so might be very inefficient

Contents

Vector intrinsics

- processor/platform-specific functions and types
- on x86 processors, put this in your code

```
#include <x86intrin.h>
```

  and you get
  - a set of available vector types
  - a lot of functions operating on vector types
- bookmark "Intel Intrinsics Guide" (`https://software.intel.com/sites/landingpage/IntrinsicsGuide/`) when using intrinsics

Vector types + intrinsics

- vectorizing a loop is largely about converting

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

$\Rightarrow$

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

- the combination of vector types + intrinsics gives you a powerful way to *manually* vectorize code (i.e., write $S(i:i+L)$) the compiler fails to vectorize

When you want to use manual vectorization

- whenever your compiler fails, but in general
  1. a loop containing *a branch* ⇒ predicated execution + value-blending
  2. a loop accessing an array *non-contiguously* ⇒ gather + scatter
  3. a loop containing *another loop* ⇒
     - easy if all inner loops have the same trip count
     - follow the strategy for branches (tedious)

Vector intrinsics

- vector types:
  - `__m512` (512 bit vector) $\approx$ `float` $\times$ 16
  - `__m512d` (512 bit vector) $\approx$ `double` $\times$ 8
  - `__m512i` (512 bit vector) $\approx$ `long` $\times$ 8
  - there are no `int` $\times$ 16
  - similar types for 256/128 bit values (`__m256`, `__m256d`, `__m256i`, `__m128`, `__m128d` and `__m128i`
- functions operating on vector types:
  - `_mm512_xxx` (512 bit),
  - `_mm256_xxx` (256 bit),
  - `_mm_xxx` (128 bit),
  - ...
- each function almost directly maps to a single assembly instruction

Convenient intrinsics to make a vector value from scalar value(s)

- make a uniform vector

```
floatv v = _mm512_set1_ps(f); // { f, f, ..., f }
```

- make an arbitrary vector

```
floatv v = _mm512_set_ps(f0, f1, f2, ..., f15);
```

Contents

Contents

Predicated instructions

- SIMD instructions that take a vector of boolean values (*mask*) that specifies lanes for which the instruction is executed
- results on other lanes are taken from another SIMD register (or set zero)
- e.g., an ordinary SIMD add instruction (intrinsics)
  - `__m512 _mm512_add_ps(__m512` a`, __m512` b`)`
    $\equiv [\ a[i] + b[i] \mid i \in 0..L\ ]$
- predicated versions
  - `__m512 _mm512_maskz_add_ps(__mmask16` k`,` a`,` b`)`
    $\equiv [\ (k[i]\ \texttt{?}\ a[i] + b[i]\ \texttt{:}\ 0) \mid i \in 0..L\ ]$
  - `__m512 _mm512_mask_add_ps(__m512` c`,` k`,` a`,` b`)`
    $\equiv [\ (k[i]\ \texttt{?}\ a[i] + b[i]\ \texttt{:}\ c[i]) \mid i \in 0..L\ ]$

Generating a mask

- compare all values of two vectors (with $<$)
  `__mmask16` k `= _mm512_cmp_ps_mask(`a`, `b`, _CMP_LT_OS)`
  $\equiv [\ u[i] < v[i] \mid i \in 0..L\ ]$
- you get a 16 bit *mask* that can be used for predicated execution
- search intrinsics guide for symbols to compare in other ways

A template to vectorize loops containing branches

- a loop having a branch

```
for (i = 0; i < n; i++) {
  if (C(i)) {
    T(i)
  } else {
    E(i)
  } }
```

- ⇒

```
for (i = 0; i + L <= n; i += L) {
  k = C(i : i + L)
  if (any(k)) {
    T(i : i + L) predicated on k
  }
  if (any(~k)) {
    E(i : i + L) predicated on ~k
  } }
```

- note: values used after the original `if` statement are made by blending results from both branches (see next slide)

Blending values

- there are instructions specifically for blending two vectors. e.g., `__m512 _mm512_mask_blend_ps(`k`,` a`,` b`)` $\equiv$ [($k[i]$ `?` $a[i]$ `:` $b[i]$) | $i \in i..L$]
- recall that predicated instructions already have a provision for it. e.g.,
  `__m512 _mm512_mask_add_ps(__m512` c`,` k`,` a`,` b`)` $\equiv$
  `_mm512_mask_blend_ps(`k`,` $a + b$`,` c`)`

Example

- scalar version

```
for (i = 0; i < n; i++) {
  if (i % 2 == 0) {
    y[i] = x[i] + 1;
  } else {
    y[i] = x[i] * 2;
  }
}
```

- $\Rightarrow$ pseudo code (assume $L \mid n$)

```
for (i = 0; i < n; i += L) {
  __mmask16 k = (i:i+L % 2 == 0);
  t = x[i:i+L] + 1;
  y[i:i+L] = blend(~k, x[i:i+L] * 2 : t);
}
```

Example

- ⇒ actual code

```
for (i = 0; i < n; i += L) {
  __m512i z = _mm512_set1_epi64(0)
  __mmask16 k = _mm512_cmp_epi64_mask(linear(i) & 1L, z, _MM_CMPINT_EQ)
  __m512i t = V(x[i]) + 1;
  V(y[i]) = _mm512_mask_mul_ps(t, ~k, V(x[i]), 2);
}
```

- `linear(i)` is a function (not shown) to generate a vector { `i, i+1, ..., i+L-1` }
- there are C++ tricks (operator overloading) that make this code less ugly

Contents

Gather

- an instruction that can get $[\ a[i_0], a[i_1], \cdots, a[i_{L-1}]\ ]$ as a vector value
- `__m512 _mm512_i32gather_ps(__m512i` I`, void*` a`, int` s`)` takes 16 32-bit indices I and scale s. that is, $\equiv [\ f(a\texttt{[}I\texttt{[}i\texttt{]} * s\texttt{]}) \mid i \in 0..L\ ]$ where $f(p)$ gets the value at p as a `float` value ($\equiv *((\texttt{float}*)\&p)$)
- similar versions for different index/value widths
  - 64 bit indices to gather 8 double precision (64 bit) values `__m512d _mm512_i64gather_pd`
  - 64 bit indices to gather 8 single precision (32 bit) values `__m256 _mm512_i64gather_ps`
  - 32 bit indices to gather 8 double precision (64 bit) values `__m512d _mm512_i32gather_pd`
- there are predicated versions as well (`_mm512_mask_ixxgather_ps/pd`)

Scatter

- an instruction that can assignments $a[i_0] = x_0; a[i_1] = x_1; \cdots ; a[i_{L-1}] = x_{L-1};$
- similar name conventions to gather
  - 32 bit indices, to get 32 bit values `_mm512_i32scatter_ps`
  - 64 bit indices, to get 64 bit values `_mm512_i64scatter_pd`
  - 64 bit indices, to get 32 bit values: `_mm512_i64scatter_ps`
  - 32 bit indices, to get 64 bit values: `_mm512_i32scatter_pd`
- you guessed it. there are masked versions (`_mm512_mask_ixxscatter_ps/pd`)

Contents

Loops having another loop inside

- consider how to vectorize the *outer* loop

```
for (i = 0; i < m; i++) {
  for (j = 0; j < limit; j++) {
    B(i)
  }
}
```

- if the trip count of the inner loop is the same across lanes (i.e., *limit* does not depend on i), then there is no particular difficulty (the compiler nevertheless often fails to vectorize it)
- more difficult is when inner loop has different trip counts depending on i

Loops having another loop inside

- a general template of scalar code

```
for (i = 0; i < m; i++) {
  while (C(i)) {
    B(i)
  }
}
```

- $\Rightarrow$

```
for (i = 0; i < m; i += L) {
  while (any(C(i : i + L))) {
    B(i : i + L) predicated on C
  }
}
```

Vector types and intrinsics : summary

- template

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

$\rightarrow$

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

- convert every expression into its *vector* version, which contains what the original expression would have for the L consecutive iterations
- use masks to handle conditional execution and nested loops with variable trip counts
- vectorizing SpMV is challenging but possible with this approach