SIMD Programming

Kenjiro Taura

Contents

Contents

Remember performance of matrix-matrix multiply?

```
1  void gemm(long n, /* n = 2400 */
2            float A[n][n], float B[n][n], float C[n][n]) {
3    long i, j, k;
4    for (i = 0; i < n; i++)
5      for (j = 0; j < n; j++)
6        for (k = 0; k < n; k++)
7          C[i][j] += A[i][k] * B[k][j];
8  }
```

```
1  $ ./simple_mm
2  C[1200][1200] = 3011.114014
3   in 56.382360 sec
4   2.451831 GFLOPS
```

```
1  $ ./opt_mm
2  C[1200][1200] = 3011.108154
3   in 1.302980 sec
4   106.095263 GFLOPS
```

What is the theoretical limit?

- Intel Skylake processor
- its *single core* can execute, in *every cycle*,
    - two *fused multiply-add instructions*
    - and others (e.g., integer arithmetic, load, store, . . .) I'll cover later
- a single fused multiply-add *instruction* can multiply/add *eight* double-precision or *sixteen* single-precision operands
- Single Instruction Multiple Data (SIMD) instructions

Terminology

- flops: floating point operations
- FLOPS: Floating Point Operations Per Second
- Practically,

> Peak FLOPS of a machine
> $=$ 2
> $\times$ vector width
> $\times$ max FMA instructions per cycle (IPC)
> $\times$ cycles per second (frequency)
> $\times$ the number of cores

Peak flops/cycle of recent cores

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

Microarchitecture	ISA	throughput (per clock)	vector width (SP)	max SP flops/cycle /core
Nehalem	SSE	1 add + 1 mul	4	8
Sandy Bridge	AVX	1 add + 1 mul	8	16
Haswell	AVX2	2 fmas	8	32
Skylake	AVX-512	2 fmas	16	64
Knights Landing (Mill)	AVX-512	2 fmas	16	64

- ISA : Instruction Set Architecture
- register width : the number of *single precision* 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

The goal

- practical ways to use SIMD instructions
- basics of processors to know what kind of code can get close-to-peak performance

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
- each value in a SIMD register is called a *SIMD lane* or simply a *lane*
- SIMD instructions can operate on several (typically all) values on a SIMD register

A SIMD register

...

lane
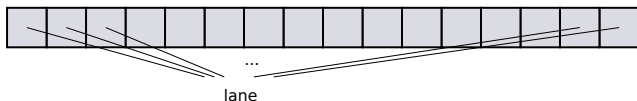
Intel SIMD instructions at a glance

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*
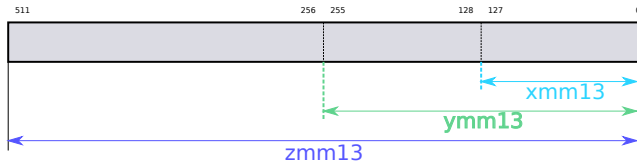
Intel SIMD instructions at a glance

- 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 : *s*calar *single* precision
- . . . sd : *s*calar *double* precision
- . . . ps : *p*acked *single* precision
- . . . pd : *p*acked *double* precision

Applications/limitations of SIMD

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

```
1  for (i = 0; i < n; i++) {
2    S(i);
3  }
```

$\Rightarrow$

```
1  for (i = 0; i < n; i += L) {
2    S(i : i + L);
3  }
4  for (; i < n; i++) {
5    S(i);
6  }
```

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
  - GCC vector extensions
  - Boost.SIMD
- intrinsics
- assembly programming

Contents

Auto loop vectorization

- write scalar loops and hope the compiler does the job

- e.g.,

```
1  void axpy_auto(float a, float * x, float c, long m) {
2    for (long j = 0; j < m; j++) {
3      x[j] = a * x[j] + c;
4    }
5  }
```

- compile and run

```
1  $ gcc -o simd_auto -march=native -O3 simd_auto.c
```

How to know if the compiler vectorized it?

- there are options useful to know whether it successfully vectorized and if not, why not

	report options
GCC	`-fopt-info-vec-{optimized,missed}`
Clang	`-R{pass,pass-missed,pass-analysis}=vectorize`
Intel	`-fopt-report-{phase,phase-missed}=vectorize`

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

```
1  asm volatile ("# xxxxxx loop begins");
2  for (i = 0; i < n; i++) {
3    ... /* hope to be vectorized */
4  }
5  asm volatile ("# xxxxxx loop ends");
```

Contents

OpenMP SIMD constructs

- `simd` pragma
  - allows an explicit vectorization of 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

```
1  void axpy_omp(float a, float * x, float c, long m) {
2  #pragma omp simd
3    for (long j = 0; j < m; j++) {
4      x[j] = a * x[j] + c;
5    }
6  }
```

- 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:

```
1  void inner_omp(float * x, float * y, long m) {
2    float c = 0;
3  #pragma omp simd reduction(c:+)
4    for (long j = 0; j < m; j++) {
5      c += x[j] * y[j];
6    }
7  }
```

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

`declare simd` pragma

- you can vectorize a function body, so that it can be called within a vectorized context

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

```
1   #pragma omp declare simd clauses
2   function definition
```

- 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

```
1  for (i = 0; i < m; i++) {
2    y[i] = a * x[i] + c;
3  }
```

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

```
1   void axpy_auto(float a, float * restrict x, float c,
2               float * restrict y, long m) {
3       for (long j = 0; j < m; j++) {
4           y[j] = a * x[j] + c;
5       }
6   }
```

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

```
1   $ gcc -march=native -O3 -S a.c -std=gnu99 -fopt-info-vec-optimized
2     ...
3   a.c:5: note: LOOP VECTORIZED.
4   a.c:1: note: vectorized 1 loops in function.
5     ...
```

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

```
1   void loop_if(float a, float * restrict x, float b,
2                float * restrict y, long n) {
3   #pragma omp simd
4     for (long i = 0; i < n; i++) {
5       if (x[i] < 0.0) {
6         y[i] = a * x[i] + b;
7       }
8     }
9   }
```

- 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

```
1  void loop_loop(float a, float * restrict x, float b,
2                 float * restrict y, long n) {
3  #pragma omp simd
4    for (long i = 0; i < n; i++) {
5      y[i] = x[i];
6      for (long j = 0; j < end; j++) {
7        y[i] = a * y[i] + b;
8      }
9    }
10 }
```

- 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

```
1  void loop_fun(float a, float * restrict x, float b,
2                float * restrict y, long n) {
3  #pragma omp simd
4    for (long i = 0; i < n; i++) {
5      f(a, x, b, y, i);
6    }
7  }
```

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

```
1  #pragma omp declare simd uniform(a, x, b, y) linear(i:1) notinbranch
2  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

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

loads zmm0 with the contiguous 64 bytes from address a

- $\rightarrow$ 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

```
1  void loop_stride(float a, float * restrict x, float b,
2                   float * restrict y, long n) {
3  #pragma omp simd
4    for (long i = 0; i < n; i++) {
5      y[i] = a * x[2 * i] + b;
6    }
7  }
```

  let alone

```
1  void loop_random(float a, float * restrict x, float b,
2                   float * restrict y, long n) {
3  #pragma omp simd
4    for (long i = 0; i < n; i++) {
5      y[i] = a * x[i * i] + b;  // or x[idx[i]]
6    }
7  }
```

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

Non-contiguous stores

- what about store

```
1  void loop_random_store(float a, float * restrict x, long * idx, float b,
2                         float * restrict y, long n) {
3  #pragma omp simd
4    for (long i = 0; i < n; i++) {
5      y[idx[i]] += a * x[i] + b;
6    }
7  }
```

- 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)

Quick experiments about the vectorization ability

- sources in `05simd` of the repository
- do not over-generalize. watch the compiler report and the output

	GCC	Clang	Clang	ICC
	5.4.0,7.3.0	3.8.0	6.0.0	18.0.1
`y[i] = a * x[i] + b`	y	y	y	y
`loop_if`		y	y	y
`loop_loop_`c	y	y	y	y
`loop_loop_`m				y
`loop_loop_`i				y
`fun`				y
`stride`	y	y		y
`random`	y	y		y
`indirect`	y	y		y
`indirect_store`	y	y		y

- `loop_loop_`$\{c,m,i\}$ refers to a version whose *end* expression of the loop is a compile-time constant (15), a loop-invariant variable (m), and a loop-dependent variable (i), respectively

Contents

GCC vector types

- GCC allows you to define a vector type

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

- You can use arithmetic on vector types

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

- recent GCCs allow you to mix scalars and vectors (Intel CC does not)

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

- You can combine them with intrinsics

axpy in GCC vector extension

- scalar code

```
1   for (long i = 0; i < n; i++) {
2     y[i] = a * x[i] + b;
3   }
```

- pseudo code; assume L | n (L divdes n)

```
1   for (long i = 0; i < n; i += L) {
2     y[i:i+L] = a * x[i:i+L] + b;
3   }
```

- with GCC vector extension

```
1   typedef float floatv
              __attribute__((vector_size(64),aligned(sizeof(float))));
2   #define V(lv) *((floatv*)&(lv))
```

assume L | n again

```
1   for (long i = 0; i < n; i += L) {
2     V(y[i]) = a * V(x[i]) + b;
3   }
```

Dealing with remainder iterations

- when L ∤ 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;
}
```

Contents

Vector intrinsics

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

```
1  #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 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
- most frequently used
    - `_mm512_fmadd_ps`, `_mm512_add_ps`, `_mm512_mul_ps`,
    - `_mm512_load_ps`, `_mm512_store_ps`,

Make a vector value from scalar value(s)

- make a uniform vector

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

- make an arbitrary vector

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

Compare and get masks

- compare all values of two vectors (with <)

```
1  floatv u, v;
2  /* k[i] = u[i] < v[i] (i = 0, ..., 15) */
3  __mmask16 k = _mm512_cmp_ps_mask(u, v, _CMP_LT_OS);
```

- you get a 16 bits *mask* that can be used for predicated execution

Predicated execution

- there are "predicated" versions for many operations. e.g.,

```
1  float a, b, c, d;
2  /* d[i] = k[i] ? (a[i] * b[i] + c[i]) : 0 ; */
3  d = _mm512_maskz_fmadd_ps(k, a, b, c);
```

- there are many variants and similar versions for other operations ($\_mm$xxx$\_mask$x$\_op\_$ps/pd)

Gather

- they take a base register + a vector of integers
- use 32 bit indices to gather 16 single precision (32 bits) values

```
1  float * a;
2  intv iv; /* int x 16 */
3  /* v[i] = a[iv[i]] for i = 0, 1, ..., 15 */
4  floatv v = _mm512_i32gather_ps((__m512i)iv, a, sizeof(float));
```

- similar versions for other combinations
    - 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 masked versions as well
  (_mm512_mask_ixxgather_ps/pd)

Scatter

- 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)

Vector types and intrinsics : summary

- template

```
1  for (i = 0; i < n; i++) {
2     S(i)
3  }
```

$\rightarrow$

```
1  for (i = 0; i < n; i += L) {
2     S(i : i + L)
3  }
```

- 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