

Session 6: Vectorisation and data layout

COMP52315: performance engineering

Lawrence Mitchell*

*`lawrence.mitchell@durham.ac.uk`

A beginning

Most of performance of chips delivered by vector instructions.
⇒ we need to use it.

We've seen that for high floating point performance we need vectorisation

What does this mean for code?

Vectorisation is typically a transformation performed on *loops*

1. Which loops can be vectorised?
2. How do we convince the compiler to vectorise them?
3. What might we expect? ⇒ roofline and other models

Simple guidelines

Necessary

1. Inner loop
2. Countable (number of iterations is known at loop entry) | \leftarrow same structure
3. Single entry/exit
4. No conditionals (but...) \leftarrow can. instructions compile to mark in much
5. No function calls (but...) \leftarrow fine if call is inlined

Best performance when

1. Simple inner loops (ideally stride-1 access)
 2. Minimize indirect addressing
 3. Align data structures to SIMD width
- $a[i+j]$ ✓
 $b[\text{idx}[i]+j]$ ✗
↓
data-dependent.

Inner loop

```
for (i = 0; i < N; i++)  
  for (j = 0; j < P; j++) // Vectorisation candidate  
  ...
```

if we want this
vectorised?
→ permute loops
for j
for i
not always possible.

Not countable

```
for (i = 0; i < N; i++)  
  for (j = 0; j < P; j++) // Not vectorisable  
    if (a[j] < 2.0) // data-dependent exit  
      break;
```

$data = [1, -2, 10, -4]$

Conditionals

✓ `for (i = 0; i < N; i++) // Vectorisable with masking`
`if (data[i] > 0)`
`sum += data[i];`

$\boxed{T \mid F \mid T \mid F} \leftarrow \text{mask}$

$sum = \begin{bmatrix} 1 \\ -2 \\ 10 \\ -4 \end{bmatrix} * \begin{bmatrix} T \\ F \\ T \\ F \end{bmatrix}$

Function calls

`for (i = 0; i < N; i++) // Not vectorisable`
`sum += some_function(data[i]);`

Can get this to vectorise if `some_function` can be inlined.

compiler can't see inside.

SIMD units, a reminder

```
double *a, *b, *c;  
...  
for (i = 0; i < N; i++)  
    c[i] = a[i] + b[i];
```

Register widths:

□ 1 operand (scalar)

□□ 2 operands (SSE)

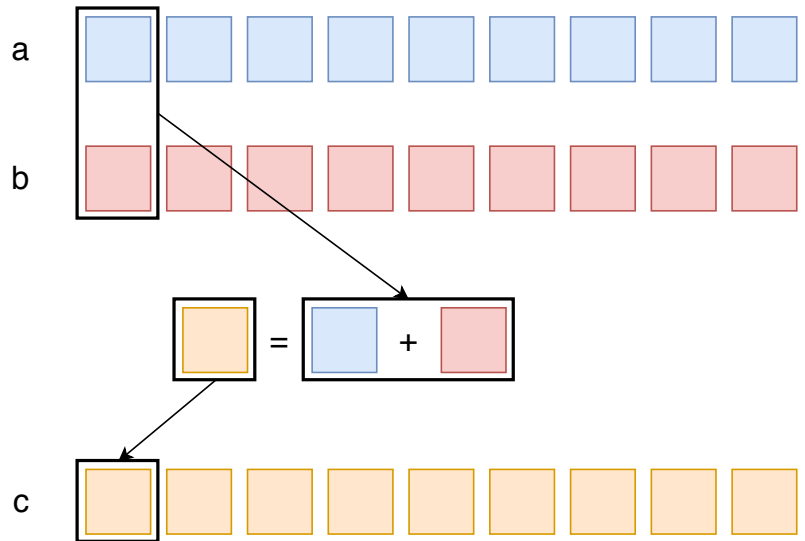
□□□□ 4 operands (AVX)

□□□□□□□□ 8 operands (AVX512)

Challenge

Best code requires SIMD loads, stores, and arithmetic.





Scalar addition, 1 output element per instruction.



SIMD units, a reminder

```
double *a, *b, *c;  
...  
for (i = 0; i < N; i++)  
    c[i] = a[i] + b[i];
```

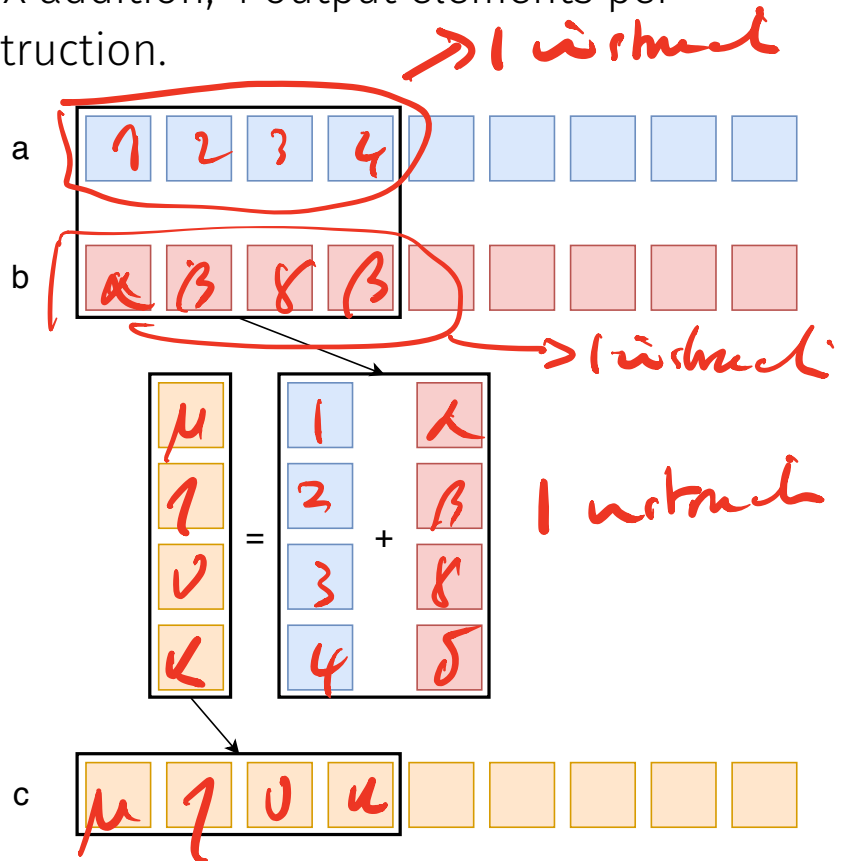
Register widths:

-  1 operand (scalar)
-  2 operands (SSE)
-  4 operands (AVX)
-  8 operands (AVX512)

Challenge

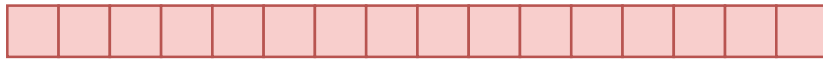
Best code requires SIMD loads, stores, and arithmetic.

AVX addition, 4 output elements per instruction.

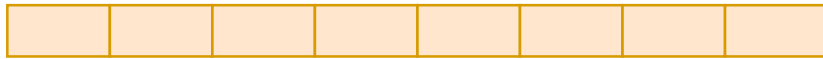


Data types in SIMD registers

Will focus on AVX (Advanced Vector eXtensions). 32-byte (256 bit) registers.



16 float16 (half precision)



8 float32 (single precision)



4 float64 (double precision)

Machine learning is really excited about half precision because “look how many numbers you can handle”.

GPU hardware has really good support for half precision.

What does the compiler do?

Unrolling

Before

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

Scalar code

↳ statements,
same ops, ←
on different
data.
⇒ candidate
for vectorisation.

After

```
// Vectorisable loop  
for (i = 0; i < n; i += 4) {  
    c[i] = a[i] + b[i];  
    c[i+1] = a[i+1] + b[i+1];  
    c[i+2] = a[i+2] + b[i+2];  
    c[i+3] = a[i+3] + b[i+3];  
}  
  
// Remainder loop  
for (i = (n/4)*4; i < n; i++)  
    c[i] = a[i] + b[i];
```

Don't do this by hand.

Carried dependencies

```
for (i = 1; i < n; i++)  
    c[i] = c[i] + c[i-1];
```

This has a *loop carried dependency*.

X
→ vectorisation not possible.

$$c[1] \leftarrow c[0] + c[1]$$

$$c[2] \leftarrow c[1] + c[2]$$

$$c[3] \leftarrow c[2] + c[3]$$

⋮

$$c[n-2] \leftarrow c[n-3] + c[n-2]$$

$$c[n-1] \leftarrow c[n-2] + c[n-1]$$

Dependencies cross
loop iterations

Can't
reorder.

Roadblocks

Pointer aliasing (C/C++ only)

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

✓ good?

each iteration looks independent.

This is allowed

```
void bar(double *data, int n) {  
    double *a = data + 2;  
    double *b = data + 1;  
    double *c = data;  
    // carried dependence in loop  
    foo(a, b, c, n - 3);  
}
```

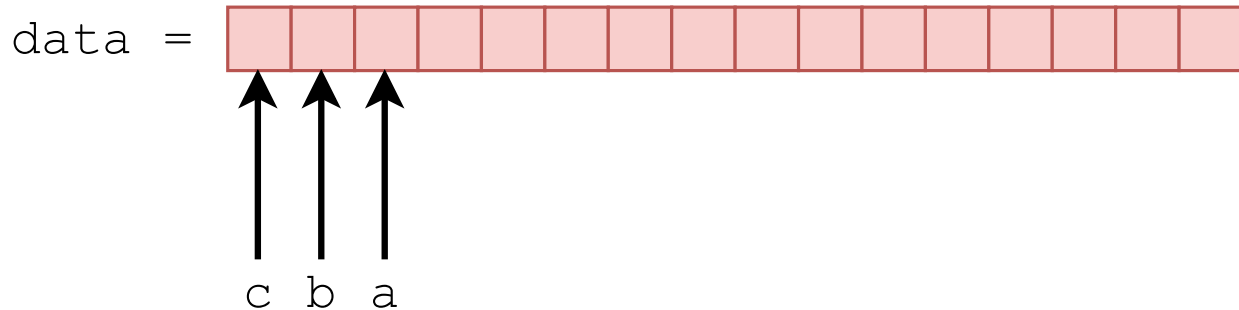
Doesn't apply to Fortran because always not allowed.

Roadblocks

This is allowed

```
void bar(double *data, int n) {  
    double *a = data + 2;  
    double *b = data + 1;  
    double *c = data;  
    foo(a, b, c, n - 3);  
}
```

*This means,
vectorisation is not
allowed.*



Pointer aliasing solution

→ think about aliasing.

Pointer aliasing (C/C++ only)

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

foo works
→ vectorise

label is one such. foo alias
→ doesn't vectorise

- Smart compiler will “multiversion” this code. Check for aliasing and dispatch to appropriate version (expensive in hot loops).
- You can guarantee that pointers won't alias (C99 or better, not C++) with `restrict` keyword (`__restrict__` in many C++ compilers)

```
void foo(double * restrict a, double * restrict b,  
         double * restrict c, int n)
```

“I promise the arrays don't alias”

Options for generating SIMD code

In decreasing order of preference

1. Compiler does it for you (but...)
2. You tell the compiler what to do (directives)
3. You choose a different (better) programming model. OpenCL, ispc
4. You use *intrinsics* (C/C++ only)
5. You write assembly code

} like these

maybe

↓
rewrite in new language.

→ Don't like this

```
#include <x86intrin.h>
void foo(...) {
    for (j = 0; j < N; j += 8) {
        t0 = _mm_loadu_ps(data + j);
        t1 = _mm_loadu_ps(data + j + 4);
        s0 = _mm_loadu_ps(s0, t0);
        s1 = _mm_loadu_ps(s1, t1);
    }
}
```

Or use
libraries that
do it for you.

Compiler options: Intel

- At `-O2` or above, intel compiler will start vectorising ✓
- Specific vector instruction sets can be selected with `-xF00`
 1. `-xSSE2` Everything post 2000ish → 128 bit
 2. `-xAVX` SandyBridge/IvyBridge (quite old) ~ 2010s
 3. `-xCORE-AVX2` Haswell/Broadwell (somewhat old) → Hamster 7.
 4. `-xCORE-AVX512` Skylake/Icelake (pretty recent) → 812 bit.
- Also prefer to set `-march=RELEVANTCPU`, e.g. `-march=broadwell`
- If you want AVX512 to really work, also need `-qopt-zmm-usage=high`
- Tell the compiler there are no pointer aliasing issues with `-fno-alias`. Don't lie here!
- See `icc -help` for more

↓
opt report compiler's
about aliasing.

Compiler options: GCC

- At `-O2` GCC vectorises if you also say `-ftree-vectorise` (or just use `-O3`).
- Specific instruction sets with
 - `-msse2`
 - `-mavx` ← *widely*
 - `-mavx2` + `-mfma` (to enable FMAs) → *haswell/broadwell*
 - `-mavx512f` ← *skylake*.
 - See `gcc --help-target` for more
- Also prefer to set `-march=RELEVANTCPU`, e.g. `-march=broadwell`
- Provide a preferred vector width (in bits) with `-mprefer-vector-width={128,256,512}`
- Tell the compiler there are no pointer aliasing issues with `-fargument-noalias`. Don't lie here!

*or vectorize?
probably.*

← widely

→ haswell/broadwell

← skylake.

How to know what is happening

- Compilers can provide feedback on what they are doing
- Intel compiler is by *far* the best here *(out intel, GCC, clang)*
- Enable output with `-qopt-report=n` ($n \in \{1, 2, 3, 4, 5\}$) with larger values producing more detail

Example

```
Begin optimization report for: foo(double *, int)
Report from: Vector optimizations [vec]
LOOP BEGIN at <source>(6,3)
remark #15344: loop was not vectorized:
vector dependence prevents vectorization

for (int i=1; i<n;i++)
  a[i] = a[i]+a[i-1];

remark #15346: vector dependence: assumed
FLOW dependence between a[i] (7:5)
and a[i-1] (7:5)
LOOP END
```

- Rpass=missed? clang.

What if things aren't vectorised

- Suppose you know that a particular loop is vectorisable, and should be, but the compiler just won't do it. *→ use intrinsics nice opt. → hard work.*
- ⇒ use OpenMP SIMD pragmas
- Turn on with `-qopenmp-simd` (Intel); `-fopenmp-simd` (GCC) *(clang)*
- Basic usage `#pragma omp simd`
- Tells the compiler "it's fine, please vectorise" ⇒ don't lie!

Example

Prerequisites

- Countable loop ✓
- Inner loop ✓

please vectorise. →

```
for (j = 0; j < n; j++)  
    #pragma omp simd  
    for (i = 0; i < n; i++)  
        b[j] += a[j*n + i];
```

Overriding the compiler cost model

not perfect.
→

- Compilers decide to do things based on some cost model
- ⇒ if the cost model is ^(often is) wrong, they may take “bad” decisions
- For vectorisation purposes we usually need to say “please vectorise this loop” or “please unroll this loop”.

Example

Vectorisation

```
#pragma omp simd
```

Extra info for
compiler.

Unrolling

```
// Intel
```

```
#pragma unroll
```

```
#pragma unroll(n)
```

```
// GCC
```

```
#pragma GCC unroll(n)
```

“unroll
n times”

clang → #pragma clang loop
unroll_count(n).

Exercise

- What do you need to do to convince Intel compiler to vectorise the GEMM micro kernel?

⇒ Exercise 9.

`https:
//teaching.wence.uk/comp52315/exercises/exercise09/`