Vector CPUs

MJ Rutter
mjr19@cam

Easter 2023

Vector CPUs

CPU clock speeds have been fairly static for a long time. Looking at Intel CPUs, one finds:

10MHz	80286, 1982
100MHz	80486DX4, 1994
1000MHz	Pentium 4, 2000
3800MHz	Pentium 4, 2004
4300MHz	Kaby Lake, 2017

This leaves five routes to better performance from modern CPUs:

Fewer clock cycles per instruction – in most cases throughput already one
Less time waiting for memory – often, but not always, an issue
More instructions executing simultaneously – requires increasingly complex hardware
More data processed per instruction – vectors!
More cores – requires parallel programming

The Benefits

TCM does not have an Intel CPU with no vector support.

However, using OpenBLAS, which permits one to change the instructions it will use, we can test a 2.83GHz Core 2 (introduced 2008), and 3GHz Kaby Lake (introduced 2017).

On Linpack 5k the Core 2 manages 10 GFLOPS. It has a vector addition unit which can start an addition of a two element vector every clock cycle, and a vector multiplier of the same capacity, so its theoretical peak performance is 11.32 GFLOPS.

The 3GHz Kaby Lake achieves 8.5 GFLOPS if restricted to the Core 2's instruction set. One might expect it to be a little faster, but here there are presumably some very specific Core 2 optimisations which the Kaby Lake does not like.

If allowed to use a vector length of four, it improves to 21.7 GFLOPS, surprisingly more than a factor of two better. If allowed to make use of the fused multiply-add instruction, then it achieves 38.7 GFLOPS (theoretical peak 48 GFLOPS).

What is Vectorisation?

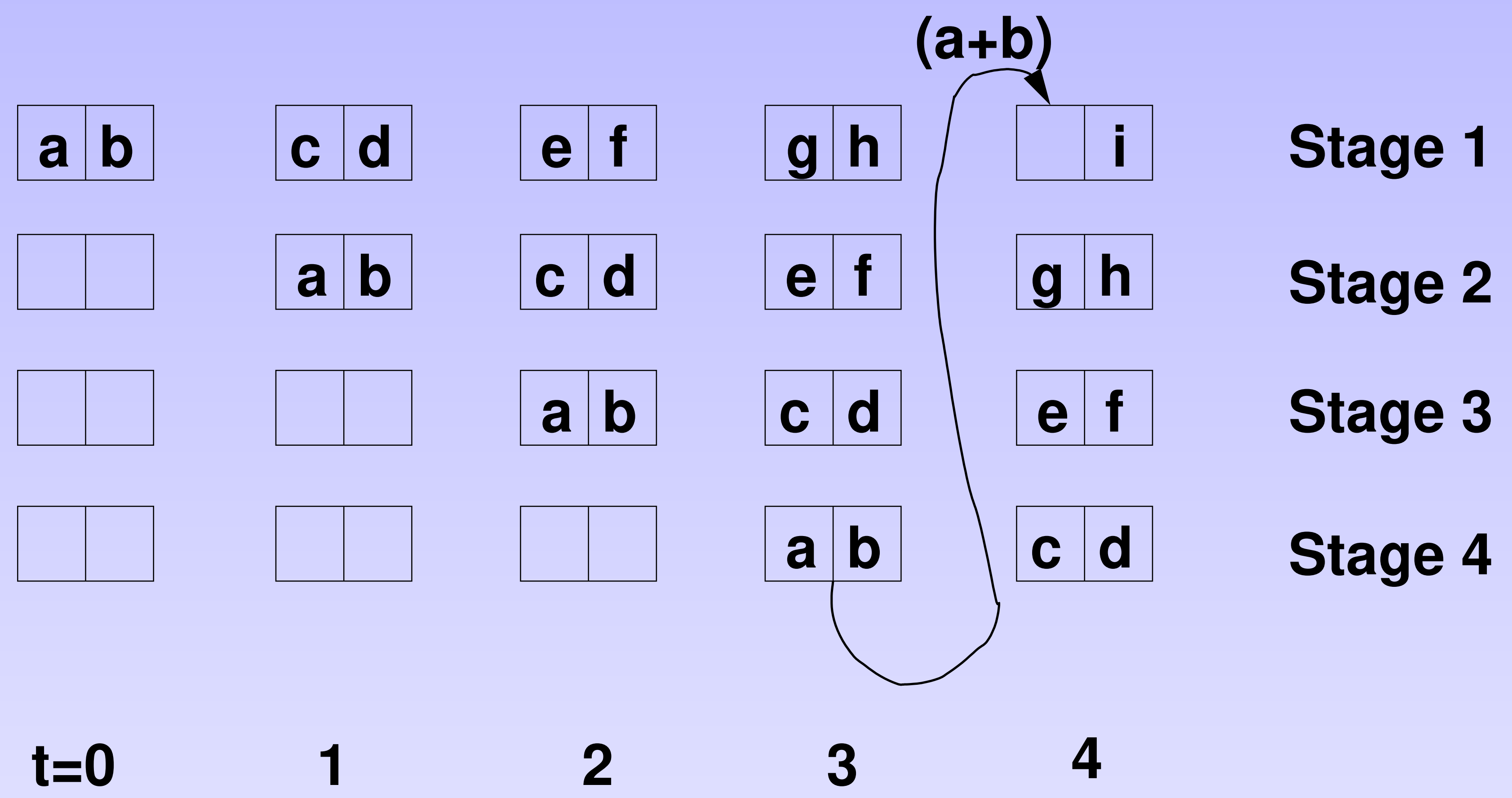
A traditional scalar CPU has functional units dedicated to particular floating-point operations (I always assume that floating point operations are all that anyone cares about. . .)

These units typically take three to six clock-cycles to produce a result once given their operands. However, they can accept new operands on every clock-cycle.

Generally there are separate units for multiplication and addition, subtraction being a trivial variation on addition, and division being so rare that worrying about how it is done is pointless.

Modern CPUs can issue operands to multiple functional units on a single clock-cycle, so on each clock cycle one addition and one multiplication can start, and the addition and multiplication started perhaps four clock-cycles earlier end.

A Scalar Pipeline



The above diagram shows data flowing through a single four-stage adder. To keep the adder fully utilised, one must always have four, data-independent additions ‘in flight’ at once.

A typical CPU will have a similar multiply unit, again with a *pipeline* of about four stages. So to keep the whole CPU ideally busy one needs an equal number of additions and multiplications, and four of each ‘in flight’ at any one time.

Data independence

```
for(i=0;i<n;i++){                do i=1,n
  sum+=a[i];                       sum=sum+a(i)
}                                enddo
```

This would appear to require four clock cycles per iteration, as the iteration `sum=sum+a[i+1]` cannot start until `sum=sum+a[i]` has completed. However, consider

```
for(i=0;i<n;i+=4){               do i=1,n,4
  s1+=a[i];                        s1=s1+a(i)
  s2+=a[i+1];                      s2=s2+a(i+1)
  s3+=a[i+2];                      s3=s3+a(i+2)
  s4+=a[i+3];                      s4=s4+a(i+3)
}                                enddo
sum=s1+s2+s3+s4;                 sum=s1+s2+s3+s4
```

The distinct partial sums have no interdependency, so one add can be issued every cycle.

Rely on the compiler to do this, as you need to know a lot about the particular processor you are using before you can tell how many partial sums to use. And worrying about *codas* for `n` not divisible by 4 is tedious.

Note that using partial sums may change the result: finite precision addition is not associative.

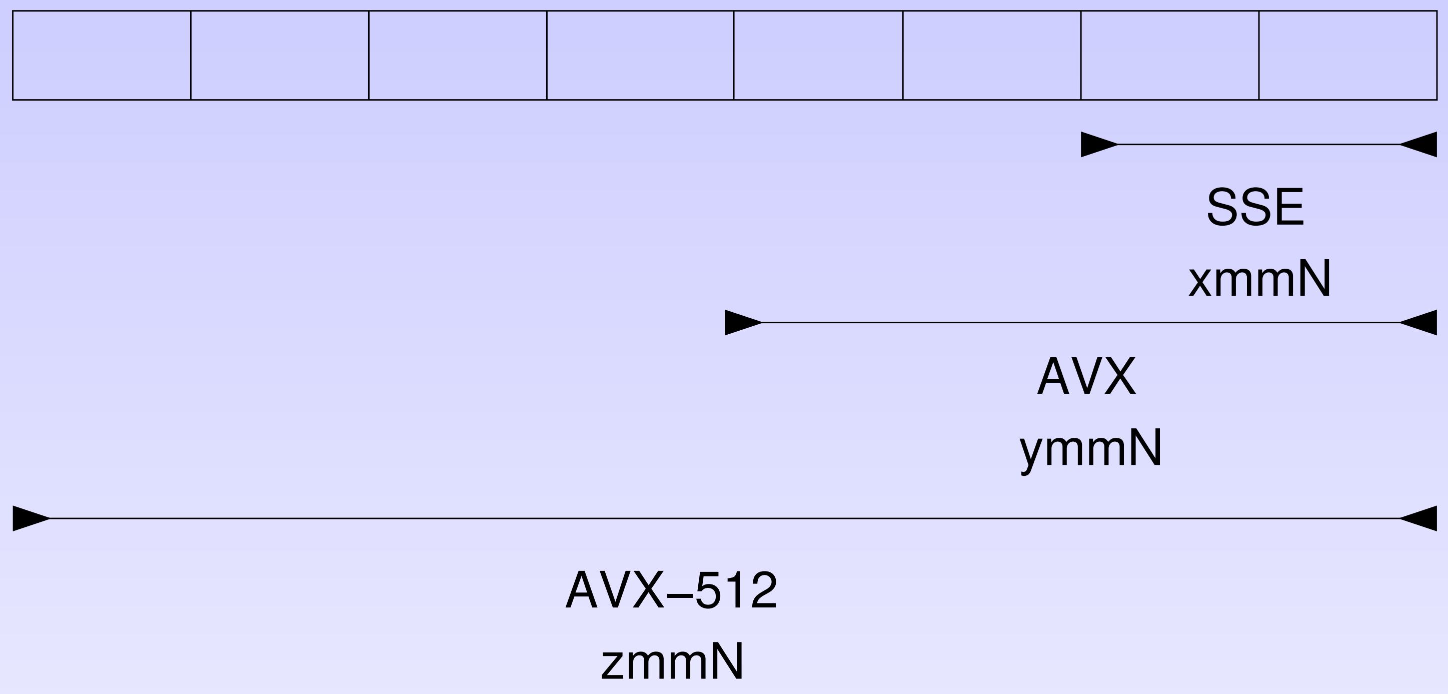
Vectorisation and Registers

The first change with vectorisation is that a set of *vector* registers exists, each of which holds multiple elements of the same datatype. In practice ‘multiple’ always means a power of two, and is generally two, four or eight. (The old vector supercomputers of the later 1980s and early 1990s were more ambitious, and tended to use vector lengths of between 32 and 128.)

Intel’s approach is to use general-purpose vector registers of a fixed length in bits. The first ones to support vectors of double-precision floating-point data were those introduced with the ‘SSE2’ instruction set with the Pentium 4 in 2000. These were 128 bits in size, and so could contain two sixty-four bit floating point doubles, or four thirty-two bit floating point singles. The same registers could also hold integers, as sixteen 8-bit integers, eight sixteen-bit integers, four 32-bit integers, or two 64-bit integers. This ability for the same registers to hold either floating-point or integer data is unusual.

More Registers

The sixteen 128 bit SSE2 registers holding two doubles each became 256 bit AVX registers holding four doubles each in 2011, and then 512 bit AVX-512 registers holding eight doubles each in selected processors from 2017 (no desktop processors yet (2019)). These registers are arranged such that AVX-512 instructions act on the full register, AVX instructions on the bottom four doubles, SSE instructions on the bottom two doubles, and there are scalar instructions too which act on just the bottom element.



(If interested in other data types, for ‘double’ read ‘pair of singles’ or ‘64 bit integer’ or ‘pair of 32 bit integers, etc.)

Vector Operations

Instructions for the expected basic arithmetic and logical operations on the vector exist, and they almost all act element-wise, e.g. for multiplication each element of the output vector is the product of the corresponding elements of the two input vectors – not what a mathematician would expect a vector product to do!

Comparison operations exist, and the result they produce has bits set to one where the comparison is true, and zero where it is false.

In 2004, Intel introduced instructions which did reductions, such as summing the elements in a vector to produce a scalar.

Vector Functional Units

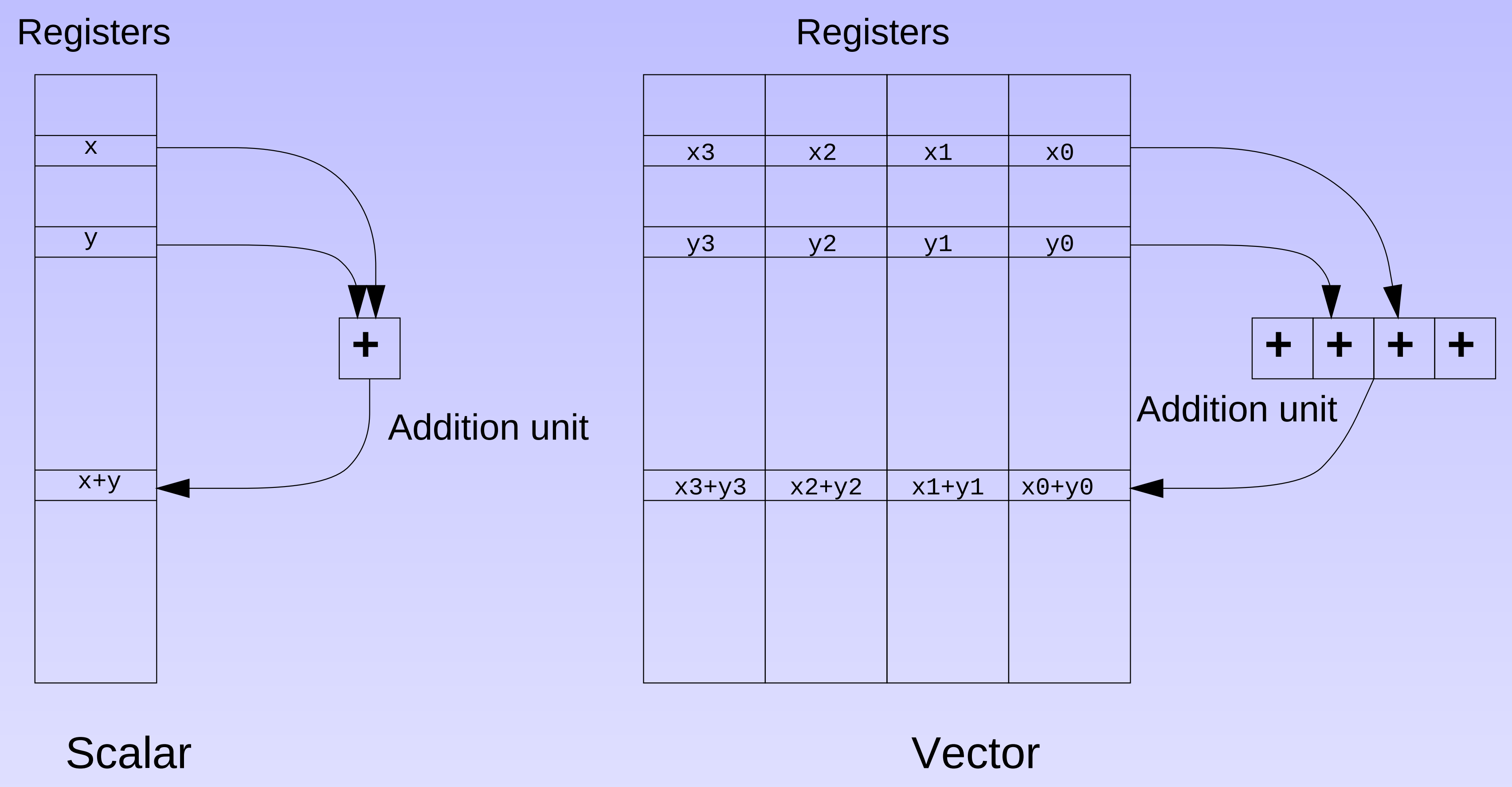
The Crays of the 1980s and early 1990s, and the Pentium 4, did not really have vector functional units. They had highly pipelined scalar units, and used the fact that an operation on a vector was an operation on a set of independent scalar values, and therefore assuming that the scalar unit could start a new operation every clockcycle, the vector register could feed in a new set of operands each cycle. Indeed, on Cray-like systems one would see instruction timings expressed in ways such as “n+8 cycles” where ‘n’ was the number of elements in the vector in use.

The recent approach from Intel has been rather different. From the Core onwards (2006), most of Intel’s CPUs have been designed with functional units which can operate on a full-width vector simultaneously. In other words, the floating point addition unit on the Core CPU can accept two 64-bit adds, or four 32-bit adds, per clock-cycle.

CPUs supporting AVX have addition units which can accept four double-precision adds each clock cycle.

Merging never occurs. An AVX unit will accept one four element instruction, one two element instruction, or one scalar instruction on each clock cycle. It will never accept two separate scalar instructions, or any other combination of multiple instructions which add up to no more than four elements. If the code contains no AVX instructions, half of each vector unit will always be idle.

In pictures



Registers and functional units quadrupled in size. Instruction fetch and issue logic unchanged. For Intel, the dedicated integer registers and functional units remain scalar, and there are dedicated vector registers and functional units, which can process integer or floating-point data.

(Intel’s old scalar floating point registers and instructions are now almost never used. Scalar operations are done in the vector registers.)

Memory costs

Assuming that the elements of a vector are stored consecutively in memory, there can be further advantages. Remember that memory likes consecutive access, and hates random access. Caches store data in lines, which are typically 64 bytes long on Intel CPUs (512 bits).

Loading a single element from memory to cache is equivalent to loading a whole line: caches generally cannot load partial lines. Loading a vector register from a cache line, assuming the internal bus in the CPU is wide enough, might take no longer than loading a scalar into a register.

However, this works best if the vector does not cross cache lines. Correct data alignment can boost performance: loads and stores associated with a register of size 2^n bytes should always be from/to addresses which are multiples of 2^n bytes.

But a vector load of any alignment rarely takes more time than two scalar loads, yet might load four (or eight) times as much data.

Intel’s Mess

Intel’s first vector instructions, SSE2, took the form of

```
    a = a OP b
```

where `a` was an SSE2 register, and `b` an SSE2 register or a memory location. With AVX this changed to

```
    c = a OP b
```

where `c` is a register, and can be distinct from `a`.

Instructions with mnemonics such as `vaddpd` act on all elements of the vector, and the number of elements is determined by the register specified (`xmm`, `ymm` or `zmm` for 2, 4 or 8).

Instructions with mnemonics such as `vaddsd` act just on the lowest element, so permitted vector lengths are 1, 2, 4 and 8.

(‘s’ = scalar/single, ‘p’ = packed, ‘d’ = double (precision)).

An aside: FMA

Intel added the fused multiply add (FMA) instruction in 2013. These instructions deal with expressions of the form a+b*c or (a+b)*c. Modern Intel CPUs can start two floating-point vector instructions each clock cycle. Each instruction can be either addition (including subtraction), multiplication, or an FMA instruction. In terms of GLFOPS, using FMA instructions doubles the potential. But not all algorithms can be expressed solely in terms of FMA operations: FFTs cannot, for instance (unless one introduces unnecessary multiplications by unity).

Assuming one has two independent floating point execution units, then one's peak FLOPS per Hz is:

No vectorisation	2	All
SSE2, SSE3	4 (vector length 2)	All
AVX	8 (vector length 4)	Sandy/Ivy Bridge
AVX2	16 (vector length 4 plus FMA)	Haswell and desktop Lakes
AVX-512	32 (vector length 8 plus FMA)	High-end Lakes

Note that FMA is not precisely equivalent to using separate adds and multiplies. FMA rounds once, separate adds and multiplies round twice. In some cases answers may differ.

Intel's Names

Intel used to have a simple naming scheme: each generation of processor had a different name under which it was marketed: 8086, 80286, 80386, 80486, Pentium, Pentium MMX, Pentium Pro, Pentium II, Pentium III, Pentium 4, Core, Core 2.

Then Intel changed to the Core i3, i5, i7, i9 scheme. The i5 and i7 were introduced in 2008 (Nehalem), the i3 in 2011 (Sandy Bridge), the i9 in 2017 (Skylake). Whilst at any given time i9>i7>i5>i3 (for some interpretation of >), the i5 and i7 in particular have been made in many different generations, and today's i3 is faster, and supports many more features, than the original i7.

So one is forced to use the names given to each generation during their development to describe the CPUs, as their marketing names are almost meaningless. Does a Core i7 support AVX or FMA? It depends which precise model number…

More on Intel's Names

The one constant about the different Core i3/i5/i7/i9 is that they do not support any form of error detection or correction in their memory. This might be fine for home use (with Windows) when software-related crashes are more likely than hardware related ones, and anyway an individual is likely to be lucky and never to own a PC with defective memory.

TCM, with over 70 desktops, takes the opposite view.

And all multisocket computers sold by Intel support memory which offers error checking and correction (ECC), and, in practice, are never sold without it.

ECC-supporting Intel processors are almost identical to the non-ECC varieties, but, since the introduction of the Core i5, they are sold under the Xeon brandname, not the Core brandname. They need different motherboards, and rarely have on-board audio.

Vectorisation Can Fail

If one's code is limited by bandwidth to main memory, then vectorisation might not help. Consider summing two arrays to make a third:

```
do i=1,n                        for(i=0;i<n;i++)
  a(i)=b(i)+c(i)                  a[i]=b[i]+c[i];
enddo
```

If this uses a single scalar addition unit, then on each clock cycle it needs to load two doubles and store one: 24 bytes per clock cycle. At 3GHz, that is 72GB/s. Assume the computer has two DDR4/3200 memory channels. That gives a theoretical bandwidth of $2 \times 64 \times 3200$MBit/s, or 51.2GB/s. One functional unit, operating in scalar mode, from just a single core, can saturate the main memory bus feeding all four cores on any of TCM's desktops. Vectorisation will probably produce a small improvement of a few percent.

If memory references are sequential, so that several scalar loads or stores get replaced by a single vector load or store, then vectorisation is likely to be at worst neutral, and almost certainly beneficial. However, if the elements which need to be loaded into a vector register (or stored from a vector register) are not sequential in memory, there is a problem.

A Problem

```
a(i,j)=0
do k=1,n
  a(i,j)=a(i,j)+b(i,k)*c(k,j)
enddo
```

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

Translates to something like

```
vtmp(1:4)=0
do k=1,n,4
  vec_b(1)=b(i,k)
  vec_b(2)=b(i,k+1)
  vec_b(3)=b(i,k+2)
  vec_b(4)=b(i,k+3)
  vec_c(1:4)=c(k:k+3,j)
  vtmp=vtmp+vec_b*vec_c
enddo
a(i,j)=sum(vtmp)
```

```
vtmp[0:3]=0;
for(k=0;k<n;k+=4){
  vec_b[0:3]=b[i][k:k+3];
  vec_c[0]=c[k][j];
  vec_c[1]=c[k+1][j];
  vec_c[2]=c[k+2][j];
  vec_c[3]=c[k+3][j];
  vtmp[0:3]+=vec_b[0:3]*vec_c[0:3];
}
a[i][j]=sum(vtmp[0:3]);
```

This will (probably) be a win as one of the loads has neatly translated into a single vector load. However, the other has split into four loads of single elements, and this part is likely to prove to be slower than the corresponding scalar code.

Why so bad?

Ideally one’s code mixes loads, stores, and arithmetic operations so that multiple instructions can be sent to different functional units on each clock cycle. Needing to execute a large block of loads before the arithmetic operations can start is not helpful.

Intel provides instructions for loading scalars into the bottom two elements of a vector only. One then needs to do some shuffling to fill the whole of an AVX vector register. (AVX-512 is better in this regard.)

Transparency

Unit stride memory operations are always faster. The basic design of SDRAM ensures this, as does the design of caches. However, without vectorisation what matters is the memory access pattern seen by the memory controller. A loop such as

```
do i=1,n,s                  for(i=0;i<n;i+=s)
  dot=dot+b(i)*c(i)           dot+=b[i]*c[i];
enddo
```

will run fast whenever `s=1`, and will run almost (or often exactly) as fast as the loop

```
do i=1,n                 for(i=0;i<n;i++)
  dot=dot+b(i)*c(i)           dot+=b[i]*c[i];
enddo
```

With vectorisation this is not the case. If `s` is known to be one, then the loop consists of trivial vector loads. If `s` is not known to be one at compile time, then the compiler will have to produce scalar loads, and either pack these into a vector, or give up and do the arithmetic with scalars. Either approach will be slower than the unit stride vectorised approach.

Ugh!

In one piece of code I have been working on, I have observed that writing

```
if (s.eq.1) then                  if (s==1){
  do i=1,n                          for(i=0;i<n;i++)
    dot=dot+b(i)*c(i)                 dot+=b[i]*c[i];
  enddo                           }
else                              else{
  do i=1,n,s                        for(i=0;i<n;i+=s)
    dot=dot+b(i)*c(i)                 dot+=b[i]*c[i];
  enddo                           }
end if
```

is, with most compilers, faster than leaving out the conditional. But this makes the code a long, ugly, and potentially buggy, mess as one tries to get round something that the compiler should be doing automatically. I really cannot recommend it.

Vectorisation and Compilers

Compilers tend to vectorise inner loops, with each vector element corresponding to a different loop iteration.

This requires that the iterations are

- independent
- free from conditionals
- free from function calls

But some simple functions should be allowed. Certainly sqrt, min and max. If you are lucky sin, cos and exp (but not with the complex data type).

Helping Compilers

In general one should keep one's code simple. Don't unroll loops by hand, or attempt to construct loops for a given vector length. This will confuse the compiler. If you cannot trust your compiler to do unrolling and vectorisation, to the correct degree, itself, then, in general, you need a new compiler, not changes to your code!

Some languages are notoriously unhelpful about loops. C is one of the worst.

```
for(A;B;C) {D;}
```

is precisely equivalent to

```
A;
while (B) {D;C;}
```

The use of `for` rather than `while` is arbitrary; it imposes no further restrictions. Any of `A`, `B`, `C` or `D` may be empty, `B` need not refer to any variable in `C`, `C` need not be a simple increment (or decrement), . . .

```
#define ever (;;)
for ever {
 ...
}
```

Better

```
do i=a,b,c
  ...
enddo
```

Whilst Fortran's loop looks like C's

```
for(i=a,i<=b;i+=c){...}
```

the inflexibility of its syntax makes life much easier for compilers. Especially so as, within the body of the loop, it is not permitted to modify the loop counter.

But Fortran still guarantees that the iterations happen sequentially.

```
a(1)=1 ; a(2)=1
do i=3,size(a)
  a(i)=a(i-2)+a(i-1)
enddo
```

correctly calculates the Fibonacci sequence.

Best

```
do concurrent (i=a:b:c)
  ...
enddo
```

This (Fortran 2008) is a loop in which each iteration must be independent, and the compiler is free to execute the iterations in any order, with any degree of overlap. It could not be used with the above Fibonacci example. Fortran 90's array syntax is similar, so

```
a(1)=1 ; a(2)=1
a(3:n)=a(1:n-2)+a(2:n-1)
```

is utterly wrong, as the compiler is free to overlap and rearrange the iterations of the implicit loop.

(Fortran 95's `for all` requires that the loop body consist solely of array assignments, and guarantees that each assignment completes (for all iterations) before the next begins. It is not very useful, even to compilers!)

OpenMP to the Rescue!

OpenMP is usually thought of as a collection of directives to perform shared memory parallelism in C, C++ or Fortran. However, in 2013 OpenMP was extended to address vectorisation, or Single Instruction Multiple Data (SIMD) programming.

```
!$OMP SIMD
do i=a,b,c
  ...
enddo
```

```
#pragma omp simd
for(i=a;i<=b;i+=c){
  ...
}
```

are mostly equivalent, and equivalent to the Fortran `do concurrent` example. There must be no `exit` statement in the loop, and, even in C/C++, the loop variable must not be modified in the loop. There are also severe restrictions on function calls within the loop.

Note that Fortran leaves it up to the compiler to decide whether `do concurrent` should be ignored, cause vectorisation, cause threaded parallelisation, or both. (There are combined simd and parallelisation OMP directives.)

Excellent

```
sum=0
!$OMP SIMD reduction(+:sum)
do i=1,n
  sum=sum+a(i)
enddo
```

```
sum=0;
#pragma omp simd reduction(+:sum)
for(i=0;i<n;i++)
   sum+=a[i];
```

In this case, unnecessary – if the compiler cannot spot what to do with that loop without hints, get a new compiler!

It shows a flexibility absent from Fortran’s `do concurrent` in F2008 and F2018, but added in F2023.
`do concurrent(i=1:n) reduce(+:sum)`

Note that, with the exception of Fortran’s complex datatype, which might fill two elements of a vector, the approach of compilers to vectorisation is that each element of the vector contains data for a different loop iteration. They do not attempt to vectorise within the loop body, nor things which are not loops with run-time known iteration counts.

OpenMP SIMD support

OpenMP 4.0 was the first version of OpenMP to support SIMD, and it was standardised in 2013. However, an OpenMP compiler may ignore SIMD constructs.

The current (2023) version of NAG's Fortran compiler does not support OpenMP 4.0 at all. Indeed, it regards some OMP 4.0 directives as errors, which is very unhelpful.

I believe that GCC from 6.1, Intel's compilers from 16.0, and LLVM from 3.9 all support the SIMD directives of OpenMP 4.0, although one may need to specify extra flags to the compiler at compile time.

Ideally the compiler will have a flag to turn on OpenMP SIMD, but not OpenMP threading.

There is more to OpenMP SIMD than the above slides show: it can collapse nested loops, specify how long a vector can be safely used (i.e. iteration n will not depend on iterations $n-7$ to $n-1$, but might depend on earlier iterations), and whether variables can be considered entirely private to an iteration or not.

Minding One's Language

```
subroutine vadd(a,b,c,n)
  real(kind(1d))::a(*),b(*),(c*)
  integer :: i,n

  do i=1,n
    c(i)=a(i)+b(i)
  enddo
end subroutine
```

```
void vadd(double *a, double *b,
          double *c, int n){
  int i;

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

}
```

The Fortran loop can be vectorised. The C (or C++) loop cannot be.

Fibonacci

```
a(1)=1                                          a[0]=1
a(2)=1                                          a[1]=1
call vadd(a(1:n-2),a(2:n-1),a(3:n),n-2)        vadd(a,a+1,a+2,n-2);
```

The C/C++ code will produce the Fibonacci sequence, $x_{n+1} = x_n + x_{n-1}$.

The Fortran code is invalid. Arguments passed to subroutines or functions must be distinct, they cannot overlap.

The Fortran compiler can vectorise, as it can assume that all loop iterations are independent. The C/C++ compiler cannot, as writes to `c` may conflict with reads from `a` or `b`. C programmers need to use omp pragmas (or similar) much more than Fortran programmers.

It can be amusing to write Fortran code like the above, and see how many compilers take advantage of their freedom to produce a `vadd` which cannot produce the Fibonacci sequence.

Less Vectorisation

```
complex (kind=kind(1d0)) :: z,z0

z=z0
do i=1,n
  z=z*z+z0
  if (abs(z).gt.4d0) exit
enddo
```

The core of a Mandelbrot Set generator. The loop has an unpredictable conditional exit and each iteration depends on the previous – not vectorisable in the manner described above.

But many compilers, if the target supports at least SSE3 (a minor, but important, addition to SSE2), will treat the complex datatype as a vector of two elements. So this may vectorise with a vector length of two.

Whilst Intel has never produced a 64-bit processor which does not support SSE3, and AMD produced just a single generation of 64-bit processors which did not (the orginal, single-core Athlon64 and Opteron, superceded in 2005 by SSE3-supporting versions), some 64-bit compilers still default to SSE2-only.

The additions of SSE3 reduced the shuffling needed to perform complex-complex multiplication if both parts of a complex number were stored in the same vector register.

A Problem

Does one compile for speed, and sacrifice compatibility with older hardware, or compile for compatibility, and lose most of the performance benefits of modern hardware?

For a major, single-architecture, machine such as a local or national supercomputer, clearly one compiles for its architecture: the binary will never be run elsewhere.

Many compilers can produce code which switches execution path at run-time depending on which processor it is executing on. In theory the speed-insensitive parts get compiled to some very general instructions, and then two (or more) versions of the speed-critical parts are produced with run-time selection.

This technique is used by many optimised maths libraries, including OpenBLAS and Intel's MKL. So if the only speed-critical part of one's code is its Lapack calls, one can get this tuning for free. If the maths library is dynamically-linked, even old binaries compiled before a new architecture existed may be able to take advantage of its features.

Running code which uses instructions unsupported by the CPU results in a crash with SIGILL, SIGnal ILLegal instruction, where 'illegal' generally means 'unknown'.

Trickery

If code compiled using Intel's compiler finds itself running on an AMD processor which supports AVX instructions, will it execute the AVX instruction path, or will it choose to execute the slower SSE2 path, and thus make the AMD processor look less good?

From the documentation of Intel's compiler:

> `-axcode` Tells the compiler to generate multiple, feature-specific auto-dispatch code paths for Intel processors if there is a performance benefit. It also generates a baseline code path. The Intel feature-specific auto-dispatch path is usually more optimized than the baseline path.
>
> The baseline code path is determined by the architecture specified by options `-m` or `-x`. The specified architecture becomes the effective minimum architecture for the baseline code path.
>
> If you specify both the `ax` and `x` options, the baseline code will only execute on Intel processors compatible with the setting specified for the `x`.
>
> If you specify both the `-ax` and `-m` options, the baseline code will execute on non-Intel processors compatible with the setting specified for the `-m` option.
>
> If you specify both the `-ax` and `-march` options, the compiler will not generate Intel-specific instructions.

Measures of Success

How does one tell how well vectorised one's code is? One quick method is to run it on different machines. Whilst there are many changes (improvements?) at each generational change, if one's code does not run about twice as fast on a Sandy/Ivy Bridge as on a Nehalem, then it is probably not taking full advantage of the move from vectors of length two to length four. Little improvment between the Bridges and later machines suggests little gain from FMA. Little improvement between Haswells and desktop Lakes, and the rare machines with dual AVX-512 units suggests little gain from the move from vectors of length four to length eight.

No vectorisation	2	All
SSE2, SSE3	4 (vector length 2)	All
AVX	8 (vector length 4)	Sandy/Ivy Bridge
AVX2	16 (vector length 4 plus FMA)	Haswell and desktop Lakes
AVX-512	32 (vector length 8 plus FMA)	High-end Lakes

In TCM the `status` command (and `rbusy`) identifies the generation of Intel CPU.

Do, however, ensure that your code has been compiled to use all the features of the CPU one is running on. For `ifort` simply recompiling with `-fast` suffices. For GCC, `-march=native`.

Assembler

One can try compiling to assembler, and then looking for evidence of vector instructions in the resulting `.s` file. Recall that scalar and two-element registers are called `xmm`, four-element `ymm` and eight element `zmm`.

```
$ gfortran-8 -mavx2 -O3 -S dot.f90
$ grep -c xmm dot.s
5
$ grep -c ymm dot.s
0
$ grep -c zmm dot.s
0
```

```
$ ifort -mavx2 -S dot.f90
$ grep -c xmm dot.s
85
$ grep -c ymm dot.s
27
$ grep -c zmm dot.s
0
```

Closer inspection shows that the gfortran code is entirely scalar, and has no unrolling.

```
function dot(x,y)
  real(kind(1d0)) :: x(:), y(:), dot
  integer :: i
  dot=0
  do i=1,size(x)
    dot=dot+x(i)*y(i)
  enddo
end function
```

```
.L5:
    vmovsd  (%rdx), %xmm1
    vmulsd  (%rax), %xmm1, %xmm1
    addl    $1, %ecx
    addq    %r8, %rdx
    addq    %rdi, %rax
    vaddsd  %xmm1, %xmm0, %xmm0
    cmpl    %ecx, %esi
    jne     .L5
```

(Only the loop body is shown in assembler, as produced by gfortran. The option `-fma` or `-march=core-avx2` did cause the `vmul` to be replaced by an FMA instruction, and the `vadd` was dropped.)

Event Counters

Another approach is to use event counters. This exposes a surprising area of Intel's CPUs. Usually one thinks of Intel's CPUs as having ridiculous levels of backwards compatibility. The latest 64 bit Intel CPU will happily run a copy of 16 bit MS DOS from the 1980s, but in one area there is very little compatibility between different generations: that of event monitoring.

Modern Intel CPUs have several counter registers which can be configured to count various events: clock cycles, instructions issued, cache misses, branch prediction failures, certain types of instructions, etc. Some CPUs have enough counters, and counters of the correct sort, to be able to monitor MFLOPS and the number of vector instructions executed (and lengths thereof). Some do not. And the precise numeric codes for the different events tend to change between generations. In TCM we have a utility called `mflops` which demonstrates the use of these counters for serial code.

Example 1

```
$ OPENBLAS_CORETYPE=core2 mflops ./linpack
Linpack result 7.4 GFLOPS

Time:                   12.872s
MFLOPS:                 6554.19
Vector length  1:          0.59%
Vector length  2:         99.41%
Vector length  4:          0.00%
Vector length  8:          0.00%
Av vector length:          1.99
```

```
$ OPENBLAS_CORETYPE=skylakex mflops ./linpack
Linpack result 50 GFLOPS

Time:                   3.32022s
MFLOPS:                  25394.4
Vector length  1:          0.65%
Vector length  2:          0.03%
Vector length  4:          3.67%
Vector length  8:         95.64%
Av vector length:          7.38
```

This is on a single 2.6GHz Xeon Gold core. The `mflops` program times the whole of the program to calculate MFLOPS, whereas Linpack itself excludes the initialisation and checking time, which are significant for small sizes (this is 5,000 x 5,000). The factor of eight in performance is $4\times$ from vector length, and $2\times$ from FMA instructions.

Example 2

```
Core2_PC$ mflops castep-18.1_ifort19_mklfft ethene
Time:                18.3387s
MFLOPS:               1931.48
Vector length  1:        4.65%
Vector length  2:       95.35%
Av vector length:        1.91

AVX512_PC$ mflops castep-18.1_ifort19_mklfft ethene
Time:                4.46153s
MFLOPS:               8384.57
Vector length  1:        3.96%
Vector length  2:       15.92%
Vector length  4:        3.16%
Vector length  8:       76.95%
Av vector length:        4.48
```

The same executable, but both the executable and library will use different code for different CPUs. The (rare) AVX-512 PC has a 40% advantage in clock speed. In the above, three quarters of FP operations were part of a vector of length eight.

Not Intel

ARM has two approaches to vectorisation.

The old, and common, Neon, uses 128 bit registers, so just two doubles. The Apple M2 uses this, as does the Raspberry Pi 4.

The new, and specialist, SVE, uses instructions which can support vectors of 128 to 2048 bits. It is designed so that the code does not need to be recompiled to support hardware with different vector lengths, but can adjust automatically at run-time. It can also use any number of elements in a vector register, not just a power of two. So far SVE has been implemented with 128, 256 and 512 bit vectors.

(Nothing new here. The Hitachi S-3600, the first vector supercomputer in Cambridge, installed c. 1996, did something similar, if quite different in detail.)