Modern Architectures & Optimization

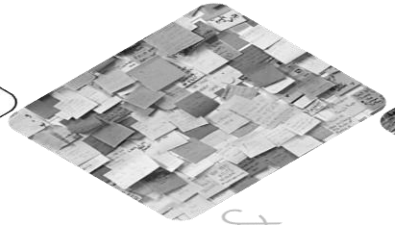
Luca Tornatore - I.N.A.F.

"**Foundation of HPC" course**

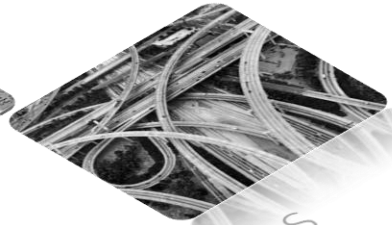
DATA SCIENCE &
SCIENTIFIC COMPUTING
2020-2021 @ Università di Trieste

Outline

First things first

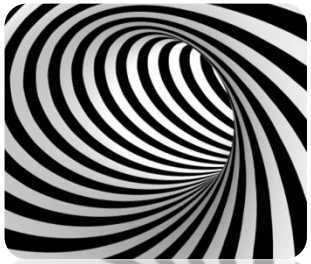


Cache & Memory



Branches

Pipelines



Loops

How to increase the CPU performance

If we define the *performance* of a CPU as the number of instructions it can deliver in, say, a second, there are basically two strategies:

1. we increase the number of times a CPU operates per second, i.e. its *clock*. That indeed happened in the past, as we have seen.
   In abstract sense, this amounts to keep its design and hurry it up.

2. we change the design of the CPU, trying to increase the number of instructions that in average it delivers *every "time" it operates*, i.e. the number of instructions per cycle that the CPU can dispatch. This second approach has no tension with the first, since both can be pursued at the same time.
   In this lecture we focus on this second way: what is the design change that makes it possible and how to exploit it from the programmer's point of view.

Increasing the "productivity"

A way to increase the ***average*** number of instructions that a CPU can deliver each time it operates – i.e. every CPU's cycle – is to overlap the execution of different operations.

This is called *instructions-level parallelism (ILP)*.

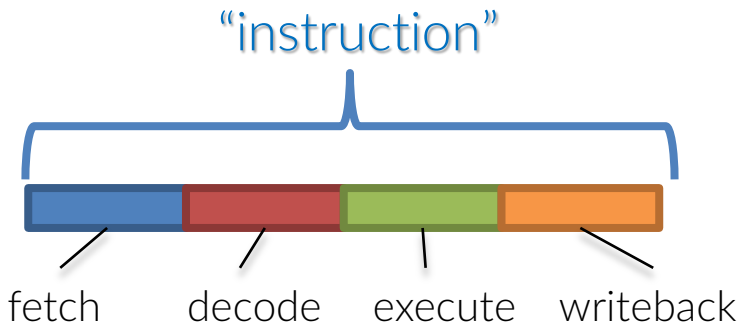
The technique covered in this lecture that implements it is called *pipelining*, which exploits the natural parallelism among the different simpler operations needed to complete a single CPU's instruction.

This technique is about **increasing the average number of instructions delivered per cpu's cycle**, ***i.e. its throughput***, not about rendering the single instructions faster.

Atomic instructions

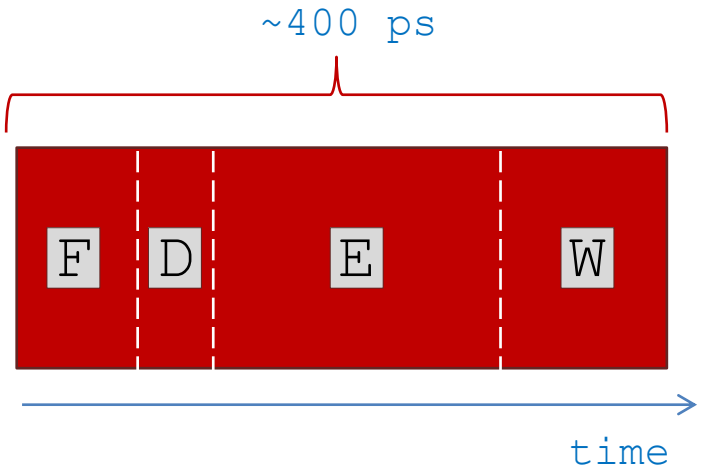
It would be obvious to think that an “instruction” is a kind of *atomic* operation that the CPU perform as a whole. Indeed that was true until the mid of the 80s.

If you think carefully about it, it is easy to understand that actually an “instruction” involves at least the following ***independent*** steps:



1. *Fetching*
   it must be recalled from memory/Icache
2. *Deconding:*
   it must be “understood and interpreted”
3. *Execution*
4. *Writeback :*
   the result must be accounted in memory ()

Atomic instructions ?

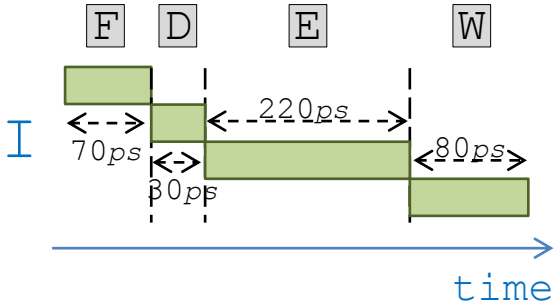


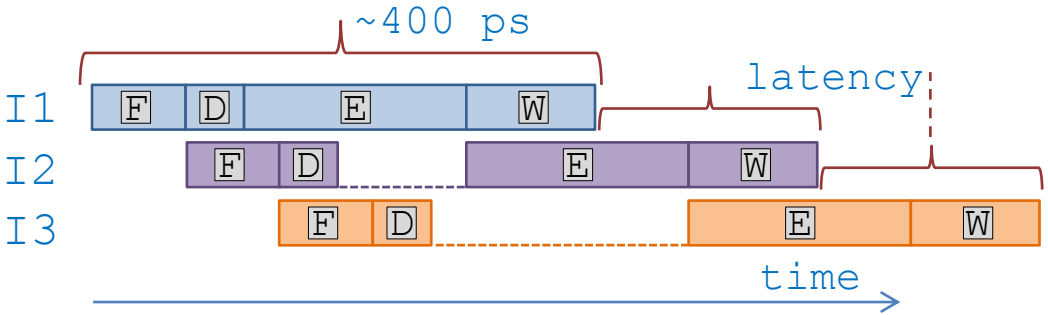
If all the four stages take ~400ps, we then would obtain a **throughput** of 2.5GIPS (giga-instructions per second).
400ps is also the total time required to get a result from an instruction, and so it is the **latency** of the instruction, or the **delay** between two subsequent instructions.

However, if we were able to "detach" the four stages, we could organize things differently, like in a car-building chain, or even at the mensa of the university.

Pipelines

Note: all the timing estimates are hypothetical for the purpose of discussing the concepts



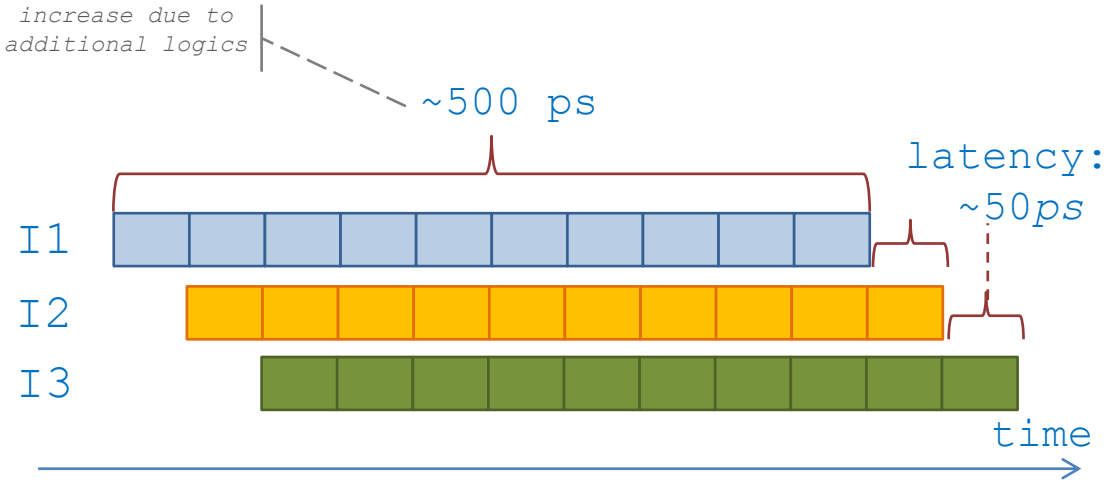


If many independent logical units exist to perform each step, they could operate subsequently on **different instructions**:

If the stage delays are not uniform, the throughput is limited by the latency `F+(D+E)-(F+D)` = `E` ~ 220*ps*, which means we have a throughput of ~**4.5GIPS** just because of logic units separation.

Pipelines

Therefore, introducing the instructions pipelining, we can increase the **throughput** of our system by a large factor.
However, the efficiency of the pipelines is limited by its longest stage: the better option would be to have all equal stages, for instance further subdividing each stage – especially the most demanding ones (*).



Now the throughput of our system has increased to 1 instruction retired every 50*ps*, i.e. 20GIPS

(*) this is called *superpipelining*: modern CPUs may have 10-20 stages per pipeline.

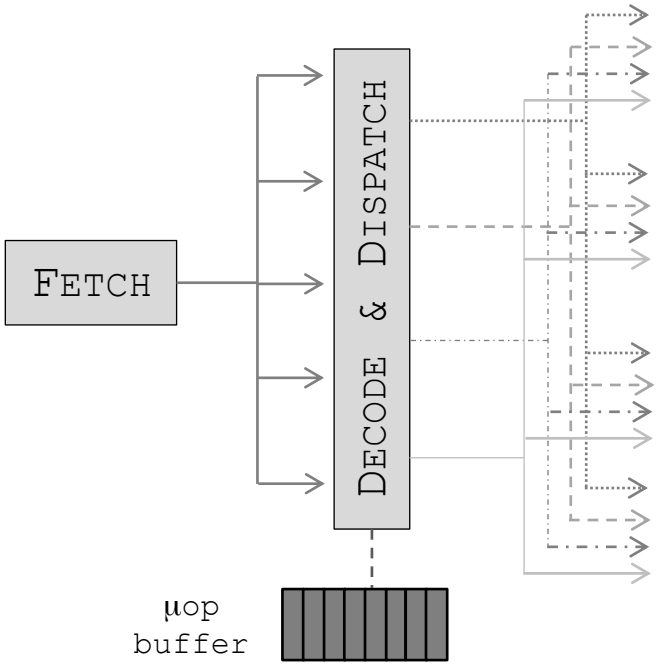
Pipelines are about throughput

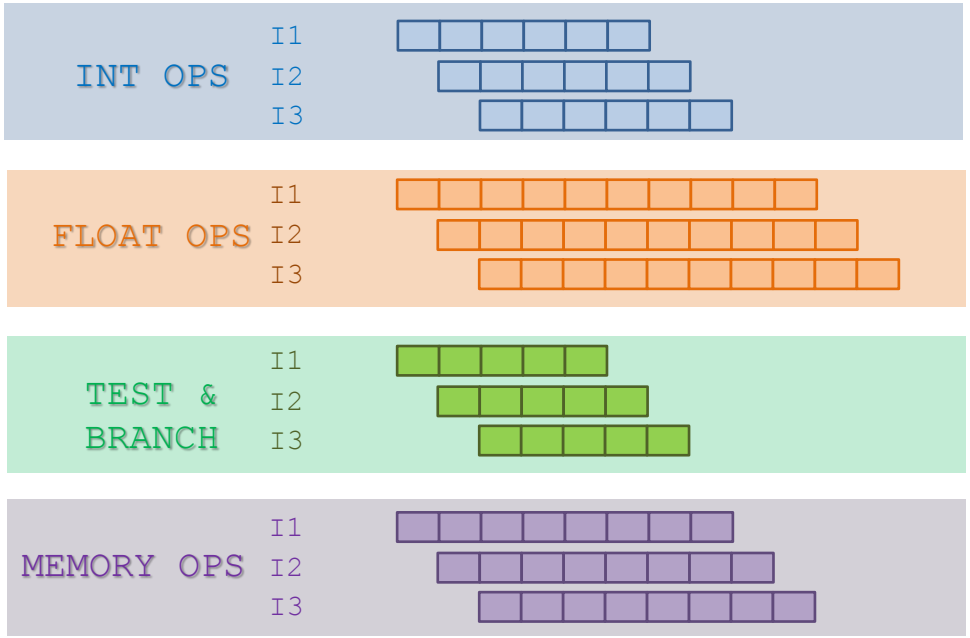
Pipelining is then about increasing the throughput of a system, and as we have seen it could be really effective.

However, there is more that can be done working on the **execution stage** that, of course, encompasses a large number of different ***functional units***, performing a **different and independent tasks**.

With an enhancement of the decode/dispatch stage, we can address multiple pipelines that can be active at the same time:

Multiple pipelines





Pipelines are about throughput

Let's have a look at an example. A given CPU has a clock of 2GHz, i.e. a single cycle lasts 0.5*ns*; let's consider its instructions have the following latencies:

ALU	3 cycles
BRANCH	5 cycles
MEMORY ACCESS	4 cycles

Let's say that in a given average code, the frequency of those instructions are 20%, 30% and 50%. Then, the average delivery (cycles-per-instructions, CPI) of this CPU-code pair is:

$$20\% \times 3cyc + 30\% \times 5cyc + 50\% \times 4cyc = 0.2 \times 3cyc + 0.3 \times 5cyc + 0.5 \times 4cyc = 4.1 \text{ CPI}$$

Note that a different code, with frequencies (80%,10%,10%; then a much more *arithmetic-intensive* one) would obtain, on the same CPU,

$$0.8 \times 3cyc + 0.1 \times 5cyc + 0.1 \times 4cyc = 3.3 \text{ CPI}$$

Pipelines are about throughput

Then let's say that we pipeline all the instructions in the previous CPU, in such a perfect way that all the stages require exactly the same time, i.e. 1 CPU cycle.
As a consequence, after the initial delay due to the latency of the first instruction, our CPU now is able to deliver 1 instruction per cycle, i.e. has a CPI = 1

Let's also define the gain in performance as

$$gain = \frac{\langle CPI_{np} \rangle}{\langle CPI_{p} \rangle}$$

where the susbcripts *np* and *p* mean *non-pipelined* and *pipelined*, respectively, and the <> brackets mean "average".
Then, the performance increase of the two previous codes on this new CPU is ×4.1 and ×3.3.

Pipelines are about throughput

However, due to the implementation some overhead is unavoidable, and it amounts to 0.1*ns*. Then, our pipelined cycle now lasts 0.6*ns* instead of 0.5*ns*.
The real performance gain, as measured in wall clock time, is actually limited by this overhead:

$$\frac{clock_{np} \times \langle CPI_{np} \rangle}{clock_p \times \langle CPI_p \rangle} = \frac{0.5ns \times \langle CPI_{np} \rangle}{0.6ns \times \langle CPI_p \rangle} = 0.83 \times \frac{\langle CPI_{np} \rangle}{\langle CPI_p \rangle}$$

The pipelining, then, although increases the throughput of the CPU renders each single instructions a little bit slower due to the overhead that comes with the pipelining itself.

Pipelines latency

The number of cycles between when an instruction's execution stage begins and when its result is available for other instructions usage, is the ***latency*** of the instruction and it grows with the pipeline deepness (the number of stages).

Typical latencies in modern processors (read the manual of yours) range from 1cyc for integer ops, to 3-6cyc for `add` and `mul` flop to ≥10-20 for `div` flop, ~50 for trigonometric or exponential functions.

Latency for memory loads can be severely troublesome, because they are highly unpredictable and they block all other operations making it difficult to fill the delay with some other ops.

The real pipelines performance

In a perfect world, as we have seen until now, where all the stage of a pipeline are perfectly equilibrated and the pipeline implementation has no overhead at all, the theoretical maximum gain from the ILP due to the pipelining is simply

$$gain = \frac{\langle CPI_{np} \rangle}{\langle CPI_p \rangle}$$

However, at least there is some overhead (the 0.1*ns* in the previous example) due to the pipeline's implementation details and to the managing of some other factors we'll see in the next slides.
In addition, there is the impact of those same factors we are going to

The real pipelines performance

There are 3 majors type of *hazards* that impact on the pipelines performance, lowering their theoretical gain:

1. *Structural hazards*
2. *Data hazards*
3. *Control hazards*

When a pipeline incurs in an hazard, it *stalls* until the hazard is solved. Normally, instructions issued before the hazard continue while all those issued after the one stalled, stall too.
A more realistic model for the improvement from pipelining is then

$$speedup = \frac{\langle CPI_{np} \rangle}{1 + \langle SCPI_p \rangle} \left(= \frac{\text{PIPELINE DEPTH}}{1 + \langle SCPI_p \rangle} \right)$$

where <*SCPI*> means “average stalled cycles per instruction” and the last equality holds in the simplest case in which all the instructions require the same # of cycles, which in turn are equal to the pipeline’s #stages.

Pipeline hazards / structural

Structural hazards

these happens when there are not enough hardware units that can cope with the request of instructions execution.
Furthermore, some instructions are deemed (or forced to be) more *uncommon*, and so privilege has been given to the more common ones.

A typical example: the multiplication/summation (the common ones) vs. the division (the uncommon one).

Pipeline hazards / structural

From the Intel's manual "Architectures Optimization"

Some instructions are deemed (or forced to be) more *uncommon*, and so privilege has been given to the more common ones.
A typical example: the multiplication/summation (the common ones) vs. the division (the uncommon one).

Instruction	Latency[1]				Throughput			
CPUID	06_4E,06_5E	06_3D/47/56	06_3C/45/46/3F	06_3A, 06_3E	06_4E,06_5E	06_3D/47/56	06_3C/45/46/3F	06_3A, 06_3E
Lock CMPXCHG8B m64	22	19	19	24	22	19	19	24
Lock CMPXCHG16B m128	32	28	28	29	32	28	28	29
DEC/INC	1	2	2	2	0.25	0.25	0.25	0.33
IMUL r64, r64	3	3	3	3	1	1	1	1
IMUL r64[11]	4, 5	3, 4	3, 4	3, 4	1	1	1	1
IMUL r32	5	4	4	4	1	1	1	1
IDIV r64 (RDX!= 0)[9]					~85-100	~85-100	~85-100	~85-100
IDIV r32[10]					~20-26	~20-26	~20-26	~19-25
LEA	1	1	1	1	0.5	0.5	0.5	0.5

In the table above you can appreciate how *much* different the latency and the throughput (i.e., in the Intel's manual language "`The number of clock cycles required to wait before the issue ports are free to accept the same instruction again`", somehow the reciprocal of the one we are using here) of different instructions (in evidence: integer multiplication and integer division) can be. That induces you to avoid as much as possible some instructions, depending on the architecture you run.

Pipelines hazards / data hazards

Data hazards

These happen, due to the out-of-order execution (i.e. the mixing of different stages from different instructions), when the result computed by the `i`*th* instruction is used by one or more following instructions, or when there is some conflict about the memory location to be used by more than on result:

Note about notation →

"**`ins`**" : instruction
"**`reg`**" : register

"**`i`**" is an ins that in the source code comes semantically before the ins "**`j`**"

1. Read After Write (RAW); when ins `j` reads the reg `r` that should hold the result from ins `i` before it could write it (wrong value is read by `j`).
2. Write After Read (WAR); when ins `i` read the reg `r` after ins `j` writes it (wrong value is read by `i`).
3. Write After Write (WAW); when ins `i` writes the reg `r` after the ins `j` writes in the same reg (the wrong value is propagated).

Pipelines hazards / data hazards

Control hazards

These happen because of conditional execution, i.e. whenever there is a branch in the control flow due to a condition (i.e. *if-then*) whose evaluation may, or may not, result in a jump in the code or in the data. We have seen several details in the lecture about, exactly, branches.

The problem is, basically, that either you can not fill adequately the pipeline because in view of a condition you do not know which branch will be taken, or, when you can do speculative execution (i.e. you have a branch prediction), your speculation results to be wrong and it is necessary to flush the pipeline entirely.

This is one of the main cause of performance loss in modern superscalar/out-of-order CPUs. The logics for the runtime branch predictor occupies a large space on processor chips but it definitely is worth it.

Pipelines hazards

A very long pipeline, then, is not much more effective than a shorter one due to the real intrinsic nature of the codes that run on the CPUs.

The hazards we have just described are actually very common (very rarely a program is a stream of totally independent instructions with no jumps) and so the full exploitation of superscalarity + superpipelining is never reached.

Basically, that's the reason why we do not have 100-stage deep pipelines.

And the reason why branching is a performance-killer, too.

Loops optimization through pipeline optimization

As we have seen, modern processor may be able to “perform more than one operations at a time”, through super-pipelining operations.

As for what concerns the programmer’s perspective, codes must be written so to make the pipelines saturation as effective as possible.

```
for (int i = 0; i < N; i++)
  S += a[i] * b[i];
```

This way S is both read and written (the S as input in iteration *i* depends on S as output of iteration *i−1*) and the FP pipeline is difficult to be exploited (iteration *i* must unavoidably wait for iteration *i−1* to end, which takes ~3-6 cycles at least)

Make things easier for the compiler and the CPU

```
for (i = 0; i < N; i += 2) {
    sum0 += a[i] * b[i];
    sum1 += a[i+1] * b[i+1]; }
```

Unrolling make it easier for the CPU to saturate the pipelines.

v. A

```
for (i = 0; i < N; i += 2)   {
    double tmp0 = a[i] * b[i];
    double tmp1 = a[i+1] * b[i+1];
    sum0 += tmp0;
    sum1 += tmp1; }
```

Separate load and multiply from addition

v. B

Loops optimization through pipeline optimization

Make things easier for the compiler and the CPU

```
for (i = 0; i < N; i += 4)   {
      sum0 += a[i] * b[i];
      sum1 += a[i+1] * b[i+1];
      sum2 += a[i+2] * b[i+2];
      sum3 += a[i+3] * b[i+3]; }
```

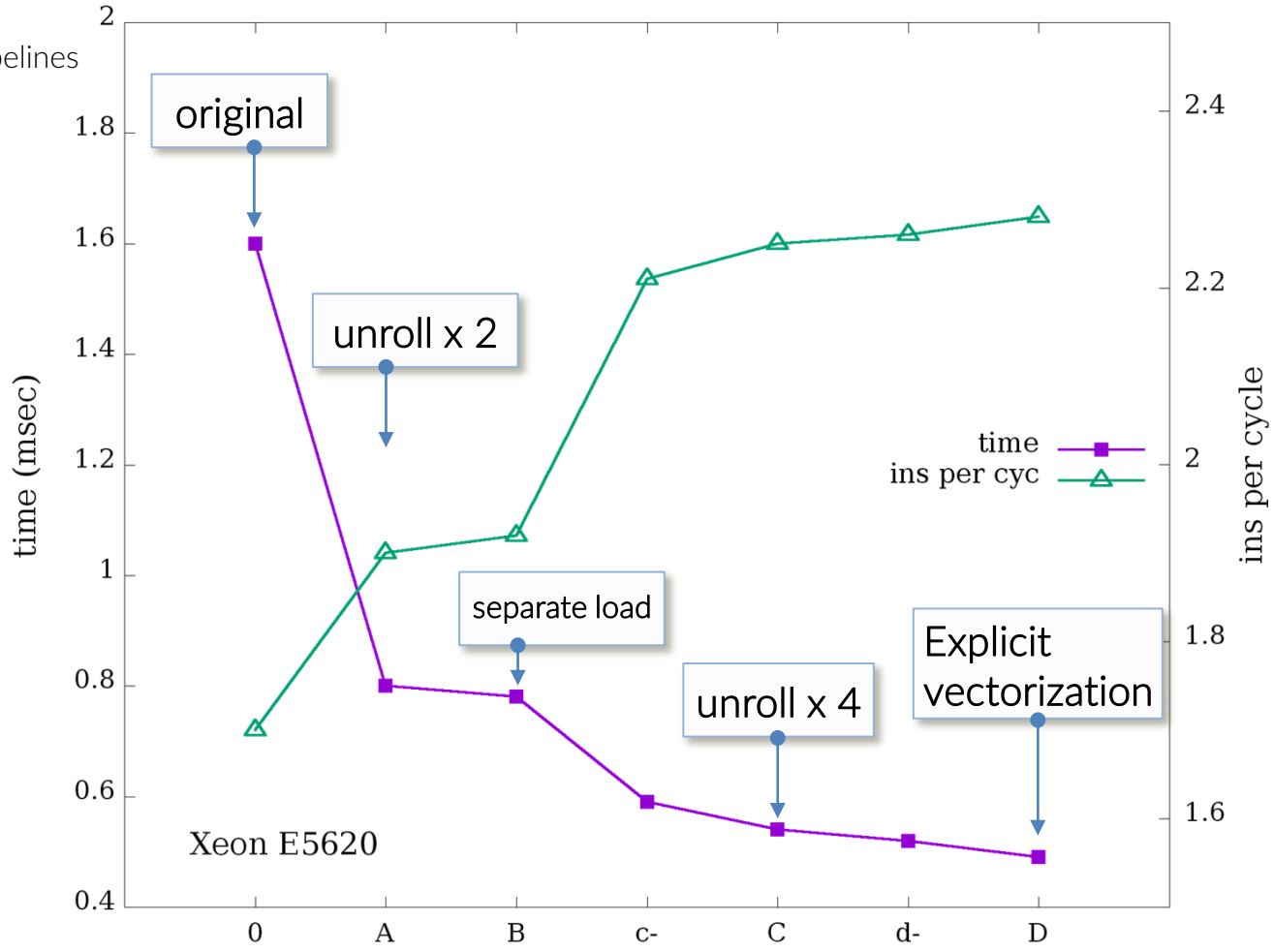
More unrolling may work even better

v. C

```
typedef double v4df __attribute__ ((vector_size (4*sizeof(double))));
v4df array1, array2;
v4df sum;
for (i = 0; i < N/4; i ++)
   sum += array1[i] * array2[i];
```

Explicit vectorization

v. D


original
unroll x 2
separate load
unroll x 4
Explicit vectorization
time
ins per cyc
Xeon E5620
time (msec)
ins per cycle
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
2.4
2.2
2
1.8
1.6
0
A
B
c-
C
d-
D

Let us now look in more detail to what happens at assembler level, so to understand in deeper details why some implementations are more effective than others.

We will use `-O0`, for the usual reason: with such simple kernels, compilers are very good (not all equally good..) in optimizing the code, and differences among improved version are more vagues.

*…that **does NOT** relieve you of knowing how to write not-that-bad code..*

Hands-on

v0

```
for (int i = 0; i < N; i++)
  S += a[i] * b[i];
```

Comparison btw 2 compilers

v0

```
.LB3365:
##       for ( int i = 0; i < N; i++ )
        cmpl    %ebx, %r14d
        jge     .LB3366
##  lineno: 185
..LN20:

        movslq  %r14d, %rax
        movq    -72(%rbp), %rcx
        vmovsd  (%rcx,%rax,8), %xmm0

        vmovsd  -32(%rbp), %xmm1

        vfmadd132sd     (%r12,%rax,8), %xmm1, %xmm0

        vmovsd  %xmm0, -32(%rbp)


        addl    $1, %r14d
        jmp     .LB3365
```

PGI 2018.10

```
.L8:
# v0.c:185:     sum += array1[i] * array2[i];
        mov     eax, DWORD PTR -136[rbp]        # tmp158, i
        cdqe
        lea     rdx, 0[0+rax*8]
        mov     rax, QWORD PTR -96[rbp] # tmp159, array1
        add     rax, rdx
        vmovsd  xmm1, QWORD PTR [rax]
        mov     eax, DWORD PTR -136[rbp]
        cdqe
        lea     rdx, 0[0+rax*8] # _23,
        mov     rax, QWORD PTR -88[rbp] # tmp161, array2
        add     rax, rdx
        vmovsd  xmm0, QWORD PTR [rax]
        vmulsd  xmm0, xmm1, xmm0
        vmovsd  xmm1, QWORD PTR -104[rbp]       # tmp163, sum
        vaddsd  xmm0, xmm1, xmm0
        vmovsd  QWORD PTR -104[rbp], xmm0       # sum, tmp162
        inc     DWORD PTR -136[rbp]     # i
# v0.c:184:       for ( int i = 0; i < N; i++ )
        mov     eax, DWORD PTR -136[rbp]        # tmp164, i
        cmp     eax, DWORD PTR -148[rbp]        # tmp164, N
        jl      .L8
```

Gcc 8.2.0

Comparison btw 2 compilers

```
.LB3365:
##       for ( int i = 0; i < N; i++ )
        cmpl    %ebx, %r14d
        jge     .LB3366
##  lineno: 185
..LN20:

        movslq  %r14d, %rax
        movq    -72(%rbp), %rcx
        vmovsd  (%rcx,%rax,8), %xmm0

        vmovsd  -32(%rbp), %xmm1

        vfmadd132sd     (%r12,%rax,8), %xmm1, %xmm0

        vmovsd  %xmm0, -32(%rbp)


        addl    $1, %r14d
        jmp     .LB3365
```

```
luca@GGG:~/code/HPC_lectures/pipeline/2018% pgi/v0_papi_O0 50000000
generating 100000000 numbers..done
sum is 1.24993e+07
time is :0.176009 (min 0.175734, std_dev 0.00054682, all 1.76643)
transfer rate was    4.23 GB/sec (  28.34% of theoretical max that is    15 GB/sec)
        IPC:    1
         [ time:  0.176sec  -  ins:     5e+08 ]
```

```
.L8:
# v0.c:185:     sum += array1[i] * array2[i];
        mov     eax, DWORD PTR -136[rbp]        # tmp158, i
        cdqe
        lea     rdx, 0[0+rax*8]
        mov     rax, QWORD PTR -96[rbp] # tmp159, array1
        add     rax, rdx
        vmovsd  xmm1, QWORD PTR [rax]
        mov     eax, DWORD PTR -136[rbp]
        cdqe
        lea     rdx, 0[0+rax*8] # _23,
        mov     rax, QWORD PTR -88[rbp] # tmp161, array2
        add     rax, rdx
        vmovsd  xmm0, QWORD PTR [rax]
        vmulsd  xmm0, xmm1, xmm0
        vmovsd  xmm1, QWORD PTR -104[rbp]       # tmp163, sum
        vaddsd  xmm0, xmm1, xmm0
        vmovsd  QWORD PTR -104[rbp], xmm0       # sum, tmp162
        inc     DWORD PTR -136[rbp]     # i
# v0.c:184:       for ( int i = 0; i < N; i++ )
        mov     eax, DWORD PTR -136[rbp]        # tmp164, i
        cmp     eax, DWORD PTR -148[rbp]        # tmp164, N
        jl      .L8
```

```
luca@GGG:~/code/HPC_lectures/pipeline/2018% ./v0_papi_O0 50000000
generating 100000000 numbers..done
sum is 1.24993e+07
time is :0.190236 (min 0.189567, std_dev 0.000538453, all 1.90762)
transfer rate was    3.92 GB/sec (  26.22% of theoretical max that is    15 GB/sec)
        IPC:  1.9
         [ time: 0.1902sec  -  ins: 1.00158e+09 ]
```

Note: no optimization has been used in both cases

Comments on the previous slides

The PGI compiler has opted for using specialised fuse multiply-addition instruction (`fmadd`) that allows for a more compact code, but an IPC of 1 (which basically means that the code occupies 1 pipeline at a time).

The gcc compiler, instead, opts for a redundant code (the 2 subsequent blocks `mov | cdqe | lea | mov | add | vmosd`) that can involve 2 pipelines, and separate multiplication and addition.

Both compiler could be induced to generate different and more efficient code playing with their options (the default behaviour can differ among different compilers).
The result is more or less equivalent from the point of view of the run-time due to the fact that the code generated by `gcc` is larger (more instructions) but it is dispatched to more pipelines (larger IPC).

NOTE: the IPC is a good metric, but it must be understood in a larger context. I.e. a larger IPC is good, but *how* good is the code depends on what instructions are being executed.

Pipelines
HANDS-ON

Turning on the optimization

Let's have a preview of how the generated code changes..

```
for (int i = 0; i < N; i++)
  S += a[i] * b[i];
```

-O0

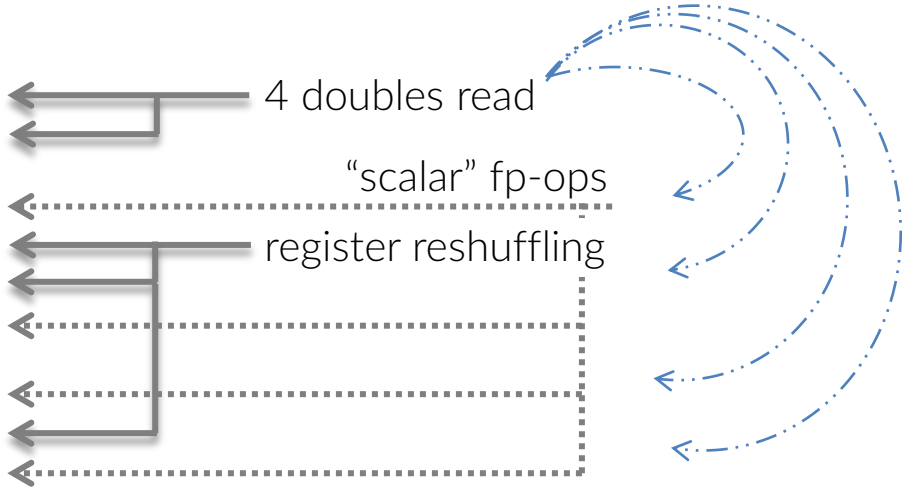
```
.L8:
# v0.c:185:     sum += array1[i] * array2[i];
        mov     eax, DWORD PTR -136[rbp]        # tmp158, i
        cdqe
        lea     rdx, 0[0+rax*8]
        mov     rax, QWORD PTR -96[rbp] # tmp159, array1
        add     rax, rdx
        vmovsd  xmm1, QWORD PTR [rax]
        mov     eax, DWORD PTR -136[rbp]
        cdqe
        lea     rdx, 0[0+rax*8] # _23,
        mov     rax, QWORD PTR -88[rbp] # tmp161, array2
        add     rax, rdx
        vmovsd  xmm0, QWORD PTR [rax]
        vmulsd  xmm0, xmm1, xmm0
        vmovsd  xmm1, QWORD PTR -104[rbp]       # tmp163,
        vaddsd  xmm0, xmm1, xmm0
        vmovsd  QWORD PTR -104[rbp], xmm0       # sum, tmp
        inc     DWORD PTR -136[rbp]     # i
# v0.c:184:       for ( int i = 0; i < N; i++ )
        mov     eax, DWORD PTR -136[rbp]        # tmp164,
        cmp     eax, DWORD PTR -148[rbp]        # tmp164,
        jl      .L8
```

-O3 -march=native

```
.L8:
# v0.c:55:      sum += array1[i] * array2[i];
        vmovupd ymm5, YMMWORD PTR 0[r13+rax]
        vmulpd  ymm0, ymm5, YMMWORD PTR [r15+rax]
        add     rax, 32 # ivtmp.18,
        vaddsd  xmm4, xmm0, xmm4
        vunpckhpd       xmm1, xmm0, xmm0
        vextractf128    xmm0, ymm0, 0x1
        vaddsd  xmm1, xmm1, xmm4
# v0.c:55:      sum += array1[i] * array2[i];
        vaddsd  xmm1, xmm0, xmm1
        vunpckhpd       xmm0, xmm0, xmm0
        vaddsd  xmm4, xmm1, xmm0
        cmp     rax, r12
        jne     .L8     #,
```

Turning on the optimization

```
.L8:
# v0.c:55:      sum += array1[i] * array2[i];
      vmovupd ymm5, YMMWORD PTR 0[r13+rax]
      vmulpd  ymm0, ymm5, YMMWORD PTR [r15+rax]
      add     rax, 32 # ivtmp.18,
      vaddsd  xmm4, xmm0, xmm4
      vunpckhpd       xmm1, xmm0, xmm0
      vextractf128    xmm0, ymm0, 0x1
      vaddsd  xmm1, xmm1, xmm4
# v0.c:55:      sum += array1[i] * array2[i];
      vaddsd  xmm1, xmm0, xmm1
      vunpckhpd       xmm0, xmm0, xmm0
      vaddsd  xmm4, xmm1, xmm0
      cmp     rax, r12
      jne     .L8     #,
```



v0

```
for (int i = 0; i < N; i++)
  S += a[i] * b[i];
```

gcc 8.2 on SkyLake
-O3 -march=native

HINT FOR IMPROVEMENT :
data are fetched with vector instructions, then processed in a mixed way

Comments on the previous slides

Looking at the assembler generated by the compiler may be useful to understand where to direct optimization efforts.

In the previous slide, the compiler (with `-O3 -march=native`) chooses to fetch data from memory with a vector instruction, the loading 4 doubles at a time (note that it uses the instruction for *unaligned memory*(*): `vmovupd`).
But however can not really exploit the ILP (Instruction-Level parallelism) due to the data dependency intrinsic in how we wrote the loop.

That's why the best first step is to exhibit the possible parallelism, for instance either by unrolling or by a different accumulation (v1 and v3 in the following slides).

(*)you should remember what alignment is from the lecture about memory allocation

V* profile – 00

```
time is :0.217358 (min 0.216092, std_dev 0.000667496)
transfer rate was   3.43 GB/sec (  22.95% of theoretical max that is    15 GB/sec)
      503,469,659      cycles:.                                        ( +- 12.99% )
      185,027,735      instructions:.          #    0.37  insn per cycle  ( +- 17.44% )
```

v0

```
time is :0.139794 (min 0.139124, std_dev 0.000733397)
transfer rate was   5.33 GB/sec (  35.68% of theoretical max that is    15 GB/sec)
      394,225,210      cycles:.                                        ( +-  9.57% )
      204,046,712      instructions:.          #    0.52  insn per cycle  ( +- 27.62% )
```

v1

```
time is :0.119491 (min 0.117614, std_dev 0.00116417)
transfer rate was   6.24 GB/sec (  41.75% of theoretical max that is    15 GB/sec)
    1,183,361,839      cycles:.                                        ( +-  6.59% )
      824,904,000      instructions:.          #    0.70  insn per cycle  ( +-  8.66% )
```

v3

```
time is: 0.0805418 (min 0.0798115, std_dev 0.00128884)
transfer rate was   9.25 GB/sec (  61.93% of theoretical max that is    15 GB/sec)
      288,211,800      cycles:.                                        ( +-  1.80% )
       92,540,296      instructions:.          #    0.32  insn per cycle  ( +-  3.80% )
```

v6

Pipelines
HANDS-ON

V* profile – O3

```
luca@GGG:~/code/HPC_lectures/pipeline/2018% ./v0_papi_O3n 50000000
generating 100000000 numbers..done
sum is 1.24993e+07
time is :0.104058 (min 0.103508, std_dev 0.000375562, all 1.04728)
transfer rate was   7.16 GB/sec (  47.94% of theoretical max that is    15 GB/sec)
        IPC: 0.65
        [ time: 0.1041sec  -  ins: 1.50004e+08 ]
luca@GGG:~/code/HPC_lectures/pipeline/2018% ./v1_papi_O3n 50000000
generating 100000000 numbers..done (  1.27)
pipeline demonstrator, step 1:
- unroll 2 times

sum is 1.24993e+07
time is :0.0755591 (min 0.0750261, std_dev 0.000291917, all 0.762576)
transfer rate was   9.86 GB/sec (  66.02% of theoretical max that is    15 GB/sec)
        IPC:  1.2
        [ time: 0.07556sec  -  ins: 1.93753e+08 ]
luca@GGG:~/code/HPC_lectures/pipeline/2018% ./v3_papi_O3n 50000000
generating 100000000 numbers..done (in   1.08sec)
pipeline demonstrator, step 3:

sum is 1.24993e+07
time is :0.0545231 (min 0.0544507, std_dev 8.46002e-05, all 0.545244)
transfer rate was   13.7 GB/sec (  91.49% of theoretical max that is    15 GB/sec)
        IPC:  1.1
        [ time: 0.0545sec  -  ins:   1.6e+08 ]%
luca@GGG:~/code/HPC_lectures/pipeline/2018% ./v3c_papi_O3n 50000000
generating 100000000 numbers..done (in   1.27sec)
pipeline demonstrator, step 3:
- unroll 2 times +
- separate mul and sum
- separate accumulations

sum is 1.24993e+07
time is :0.0527944 (min 0.0526688, std_dev 0.000107074, all 0.53314)
transfer rate was   14.1 GB/sec (  94.49% of theoretical max that is    15 GB/sec)
        IPC:  1.2
        [ time: 0.0528sec  -  ins:  1.75e+08 ]
```

V* profile – O3 + vectorisation

There's a lot of improvement in such a simple kernel, but why IPC even decreased?

Hint: the code becomes memory-bound (look at the memory transfer rate).

```
luca@GGG:~/code/HPC_lectures/pipeline/2018% ./v6_papi_O3n 50000000
generating 100000000 numbers..done (in   1.27sec)
sum is 1.24993e+07
time is: 0.0647406 (min 0.0642672, std_dev 0.00017916, all 0.654759)
transfer rate was   11.5 GB/sec (  77.05% of theoretical max that is    15 GB/sec)
        IPC:  0.8
        [ time: 0.06475sec  -  ins: 1.00003e+08 ]
luca@GGG:~/code/HPC_lectures/pipeline/2018% ./v6_intrinsics_papi_O3n 50000000
generating 100000000 numbers..done (in    1.1sec
sum is 1.24993e+07
time is: 0.0504149 (min 0.0468492, std_dev 0.00132376, all 0.509429)
transfer rate was   14.8 GB/sec (  98.95% of theoretical max that is    15 GB/sec)
        IPC: 0.67
        [ time: 0.05042sec  -  ins: 7.50003e+07 ]
luca@GGG:~/code/HPC_lectures/pipeline/2018% ./v8_papi_O3n 50000000
generating 100000000 numbers..done (in   1.11sec)
sum is 1.24993e+07
time is: 0.0499751 (min 0.0466076, std_dev 0.00124928, all 0.499767)
transfer rate was   14.9 GB/sec (  99.82% of theoretical max that is    15 GB/sec)
        IPC: 0.56
        [ time: 0.04998sec  -  ins: 6.25002e+07 ]
```

Comparison: v0 – v1

v0

```
for ( int i = 0; i < N; i++ )
  sum += array1[i] * array2[i];
```

v1

```
for ( int i = 0; i < N-1; i+=2 )
  // simply unrolling 2 times, exposes the fact that at least
  // 2 elements of the array can be processed independently
  {
    sum1 += array1[   i ] * array2[   i ];
    sum2 += array1[ i+1 ] * array2[ i+1 ];
  }
if ( N % 2 )
  sum = array1[ N-1 ] * array2[ N-1 ];
```

Comparison:

v0

```
.L8:
# v0.c:185:     sum += array1[i] * array2[i];
        mov     eax, DWORD PTR -136[rbp]        # tmp158, i
        cdqe
        lea     rdx, 0[0+rax*8]
        mov     rax, QWORD PTR -96[rbp] # tmp159, array1
        add     rax, rdx
        vmovsd  xmm1, QWORD PTR [rax]
        mov     eax, DWORD PTR -136[rbp]
        cdqe
        lea     rdx, 0[0+rax*8] # _23,
        mov     rax, QWORD PTR -88[rbp] # tmp161, array2
        add     rax, rdx
        vmovsd  xmm0, QWORD PTR [rax]
        vmulsd  xmm0, xmm1, xmm0
        vmovsd  xmm1, QWORD PTR -104[rbp]       # tmp163, sum
        vaddsd  xmm0, xmm1, xmm0
        vmovsd  QWORD PTR -104[rbp], xmm0       # sum, tmp162
        inc     DWORD PTR -136[rbp]     # i
# v0.c:184:       for ( int i = 0; i < N; i++ )
        mov     eax, DWORD PTR -136[rbp]        # tmp164, i
        cmp     eax, DWORD PTR -148[rbp]        # tmp164, N
        jl      .L8
```

v1

```
.L10:
# v1_aligned.c:91:        sum1 += array1[   i ] * array2[   i ];
        movl    -148(%rbp), %eax        # i, tmp186
        cltq
        leaq    0(,%rax,8), %rdx
        movq    -96(%rbp), %rax # array1, tmp187
        addq    %rdx, %rax
        vmovsd  (%rax), %xmm1
        movl    -148(%rbp), %eax        # i, tmp188
        cltq
        leaq    0(,%rax,8), %rdx
        movq    -88(%rbp), %rax # array2, tmp189
        addq    %rdx, %rax
        vmovsd  (%rax), %xmm0
        vmulsd  %xmm0, %xmm1, %xmm0
        vmovsd  -112(%rbp), %xmm1       # sum1, tmp191
        vaddsd  %xmm0, %xmm1, %xmm0
        vmovsd  %xmm0, -112(%rbp)       # tmp190, sum1
        movl    -148(%rbp), %eax        # i, tmp192
        cltq
        incq    %rax
        leaq    0(,%rax,8), %rdx
        movq    -96(%rbp), %rax # array1, tmp193
        addq    %rdx, %rax
        vmovsd  (%rax), %xmm1
        movl    -148(%rbp), %eax        # i, tmp194
        cltq
        incq    %rax
        leaq    0(,%rax,8), %rdx
        movq    -88(%rbp), %rax # array2, tmp195
        addq    %rdx, %rax
        vmovsd  (%rax), %xmm0
        vmulsd  %xmm0, %xmm1, %xmm0
        vmovsd  -104(%rbp), %xmm1       # sum2, tmp197
        vaddsd  %xmm0, %xmm1, %xmm0
        vmovsd  %xmm0, -104(%rbp)
        addl    $2, -148(%rbp)  #, i
.L9:
# v1_aligned.c:87:        for ( int i = 0; i < N-1; i+=2 )
        movl    -164(%rbp), %eax        # N, tmp198
        decl    %eax    # _42
        cmpl    %eax, -148(%rbp)        # i
        jl      .L10
```

Comparison:

v0

```
.L8:
# v0.c:185:      sum += array1[i] * array2[i];
        mov     eax, DWORD PTR -136[rbp]        # tmp158, i
        cdqe
        lea     rdx, 0[0+rax*8]
        mov     rax, QWORD PTR -96[rbp] # tmp159, array1
        add     rax, rdx
        vmovsd  xmm1, QWORD PTR [rax]
        mov     eax, DWORD PTR -136[rbp]
        cdqe
        lea     rdx, 0[0+rax*8] # _23,
        mov     rax, QWORD PTR -88[rbp] # tmp161, array2
        add     rax, rdx
        vmovsd  xmm0, QWORD PTR [rax]
        vmulsd  xmm0, xmm1, xmm0
        vmovsd  xmm1, QWORD PTR -104[rbp]       # tmp163, sum
        vaddsd  xmm0, xmm1, xmm0
        vmovsd  QWORD PTR -104[rbp], xmm0       # sum, tmp162
        inc     DWORD PTR -136[rbp]     # i
# v0.c:184:       for ( int i = 0; i < N; i++ )
        mov     eax, DWORD PTR -136[rbp]        # tmp164, i
        cmp     eax, DWORD PTR -148[rbp]        # tmp164, N
        jl      .L8
```

v1

```
.L10:
# v1_aligned.c:91:        sum1 += array1[   i ] * array2[   i ];
        movl    -148(%rbp), %eax        # i, tmp186
        cltq
        leaq    0(,%rax,8), %rdx
        movq    -96(%rbp), %rax # array1, tmp187
        addq    %rdx, %rax
        vmovsd  (%rax), %xmm1
        movl    -148(%rbp), %eax        # i, tmp188
        cltq
        leaq    0(,%rax,8), %rdx
        movq    -88(%rbp), %rax # array2, tmp189
        addq    %rdx, %rax
        vmovsd  (%rax), %xmm0
        vmulsd  %xmm0, %xmm1, %xmm0
        vmovsd  -112(%rbp), %xmm1       # sum1, tmp191
        vaddsd  %xmm0, %xmm1, %xmm0
        vmovsd  %xmm0, -112(%rbp)       # tmp190, sum1
        movl    -148(%rbp), %eax        # i, tmp192
        cltq
        incq    %rax
        leaq    0(,%rax,8), %rdx
        movq    -96(%rbp), %rax # array1, tmp193
        addq    %rdx, %rax
        vmovsd  (%rax), %xmm1
        movl    -148(%rbp), %eax        # i, tmp194
        cltq
        incq    %rax
        leaq    0(,%rax,8), %rdx
        movq    -88(%rbp), %rax # array2, tmp195
        addq    %rdx, %rax
        vmovsd  (%rax), %xmm0
        vmulsd  %xmm0, %xmm1, %xmm0
        vmovsd  -104(%rbp), %xmm1       # sum2, tmp197
        vaddsd  %xmm0, %xmm1, %xmm0
        vmovsd  %xmm0, -104(%rbp)
        addl    $2, -148(%rbp)  #, i
.L9:
# v1_aligned.c:87:        for ( int i = 0; i < N-1; i+=2 )
        movl    -164(%rbp), %eax        # N, tmp198
        decl    %eax    # _42
        cmpl    %eax, -148(%rbp)        # i
        jl      .L10
```

The `v1` version looks twice as longer (i.e.: more instructions to be executed), actually it really looks like the same code was copied & pasted below the original segment.

How does it come that the result is a faster execution?

Comparison: v0 – v1

The point is clearer considering the output of the PAPI instrumented code, that returns the IPC. Clearly, the 2fold unroll makes the CPU able to better exploit more than 1 logical units at a time.

```
luca@GGG:~/code/HPC_lectures/pipeline/2018% ./v0_papi_O0 50000000
generating 100000000 numbers..done
sum is 1.24993e+07
time is :0.191208 (min 0.190616, std_dev 0.00053432, all 1.91787)
transfer rate was    3.9 GB/sec (  26.09% of theoretical max that is    15 GB/sec)
        IPC:  1.9
        [ time: 0.1912sec  -  ins: 1.00158e+09 ]
luca@GGG:~/code/HPC_lectures/pipeline/2018% ./v1_papi_O0 50000000
generating 100000000 numbers..done ( 0.965)
pipeline demonstrator, step 1:
- unroll 2 times

sum is 1.24993e+07
time is :0.125749 (min 0.12396, std_dev 0.00142263, all 1.26349)
transfer rate was   5.92 GB/sec (  39.67% of theoretical max that is    15 GB/sec)
        IPC:  2.8
        [ time: 0.1258sec  -  ins: 9.76578e+08 ]
```

Refining v1

v1

```
for ( int i = 0; i < N-1; i+=2 )
  // simply unrolling 2 times, exposes the fact that at least
  // 2 elements of the array can be processed independently
  {
    sum1 += array1[   i ] * array2[   i ];
    sum2 += array1[ i+1 ] * array2[ i+1 ];
  }
if ( N % 2 )
  sum = array1[ N-1 ] * array2[ N-1 ];
```

v1b

```
    double register sum0 = 0;
    double register sum1 = 0;

    double register * restrict a1 = __builtin_assume_aligned(array1, 32);
    double register * restrict a2 = __builtin_assume_aligned(array2, 32);

    tstart = CPU_TIME;
#pragma ivdep
    for( int i = 0 ; i < N-1; i+=2 )
      {
        sum0 += *(a1++) * *(a2++);
        sum1 += *(a1++) * *(a2++);
      }
    if( N%2 )
      sum = *a1 * *a2;
```

Refining v1

```
luca@GGG:~/code/HPC_lectures/pipeline/2018% ./v1_papi_O0 50000000
generating 100000000 numbers..done ( 0.973)
pipeline demonstrator, step 1:
- unroll 2 times

sum is 1.24993e+07
time is :0.125593 (min 0.125287, std_dev 0.000476174, all 1.26231)
transfer rate was   5.93 GB/sec (  39.72% of theoretical max that is    15 GB/sec)
        IPC:  2.8
        [ time: 0.1256sec   ins: 9.76578e+08 ]
luca@GGG:~/code/HPC_lectures/pipeline/2018% ./v1b_papi_O0 50000000
generating 100000000 numbers.. done (in  0.973sec)
pipeline demonstrator, step 1b:
- unroll 2 times +
- align memory   +
- minimize ptr arithmetic

sum is 1.24993e+07
time is :0.0977905 (min 0.0972945, std_dev 0.000712555, all 0.977921)
transfer rate was   7.62 GB/sec (  51.01% of theoretical max that is    15 GB/sec)
        IPC:  2.1
        [ time: 0.0978sec   ins: 5.76578e+08 ]
```

Refining: v3

Let's go further in trying to make it clear for the CPU about the possible ILP

prefetching

v3

```
#pragma ivdep
      for ( int i = 0; i < N_4; i+=4 )
        {
          sum += array1[i] * array2[i] +
            array1[i+1] * array2[i+1] +
            array1[i+2] * array2[i+2] +
            array1[i+3] * array2[i+3];
        }

      for ( int i = N_4; i < N; i++ )
        sum += array1[i] * array2[i];
```

v3b

```
      double register a;
      double register b;
      a = array1[0] * array2[0];
#pragma ivdep
      for ( int i = 0; i < N_4; i+=4 )
        {
          DO_NOT_OPTIMIZE;
          b = array1[i+4] * array2[i+4];
          DO_NOT_OPTIMIZE;
          sum += a +
            array1[i+1] * array2[i+1] +
            array1[i+2] * array2[i+2] +
            array1[i+3] * array2[i+3];

          a = b;
        }

      for ( int i = N_4; i < N; i++ )
        sum += array1[i] * array2[i];
```

Refining: vectorisation

With the following statement you can define a vector type in `gcc`

```
typedef double v4df __attribute__ ((vector_size (4*sizeof(double))));
typedef union {
  v4df   V;
  double v[4];
}v4df_u;
```

By creating this union, you can access the single doubles within the vector

This is the actual vector, made up by 4 doubles. You can not access the single elements.

Refining: vectorisation

What is this strange stuff about ?

The loop is re-written here in its simplest form.

Now, however, the variables are vectors.

```
#ifdef _GNU_SOURCE
      v4df sum_ = {0, 0, 0, 0};
#else
      v4df sum_ = {0};
#endif
      v4df register mytmp;
      v4df register tmp  = *((v4df*)&array1[0]) * *((v4df*)&array2[0]);

      int N__4 = N/4;
      int N_4 = N__4*4;
      tstart = CPU_TIME;
#pragma ivdep
      for( int i = 1; i <= N__4; i++)
        {
          _DO_NOT_OPTIMIZE_BEGIN;
          mytmp = *((v4df*)array1 + i) * *((v4df*)array2 + i);
          _DO_NOT_OPTIMIZE_END;
          sum_ += tmp;
          tmp   = mytmp;
        }

      for ( int i = N_4; i < N; i++ )
        sum += array1[ i ] * array2[ i ];
```

With the following statements you prepare the ground to use *intrinsics*, both types and functions.

Vector type with 256bits (AVX)

You can initialize it as an array

The FMA in vector AVX flavour

```
#elif defined ( __AVX__ ) || defined ( __AVX2__ )

#define V_DSIZE 4
typedef __m256d vd;
vd _dzero_ = {0, 0, 0, 0};
#define MUL_ADD( A1, A2, R ) _mm256_fmadd_pd( (A1), (A2), (R) )
```

Intrinsics are an API, exposed by the compiler(*), that provide direct access to vector instructions and types.

Why am I defining those macros ?

(*) they are no compiler-dependent: look at https://software.intel.com/sites/landingpage/IntrinsicsGuide/ for Intel's intrinsics

Refining: vectorisation with intrinsics

As before, the loop is re-written here in its simplest form.
It also looks very similar to the previous one.
The variables are still vectors.

Note the usage of macros: the code is still valid both for AVX512 or SSEx

```
      vd sum_ __attribute__ ((aligned(64)));
      sum_ = _dzero_;

      vd * restrict A1 __attribute__ ((aligned(64)));
      vd * restrict A2 __attribute__ ((aligned(64)));
      A1  = (vd*)array1 ;
      A2  = (vd*)array2 ;

      int N__ = N / V_DSIZE;
      int N_  = N__ * V_DSIZE;

      tstart = CPU_TIME;

#pragma ivdep
      for( int i = 0; i < N__; i++)
        sum_ = MUL_ADD( A1[i], A2[i], sum_ );

      for ( int i = N_; i < N; i++ )
        sum += array1[ i ] * array2[ i ];
```

Hands-on / 2

Let's consider the polynomial evaluation:

$$a_0 + a_1 \times x + a_2 \times x^2 + \cdots + a_n \times x^n$$

Naïve implementation

```
double res = a[0];
double x_pwr = x;
for ( int i = 1; i <= n; i++ )
{
    res   += a[i] * x_pwr;
    x_pwr *= x;
}
```

Horner's implementation

```
double res = a[n];
for ( int i = n-1; i >= 0; i-- )
    res += a[i] + res * x;
```

Although Horner's implementation requires less FP operations, on a modern CPU the naïve implementation outperforms it. Why is it so ?

that's all, have fun