Imperial College London

332
Advanced Computer Architecture
Chapter 7

Data-Level Parallelism
Architectures and Programs

February 2016
Luigi Nardi

These lecture notes are partly based on:

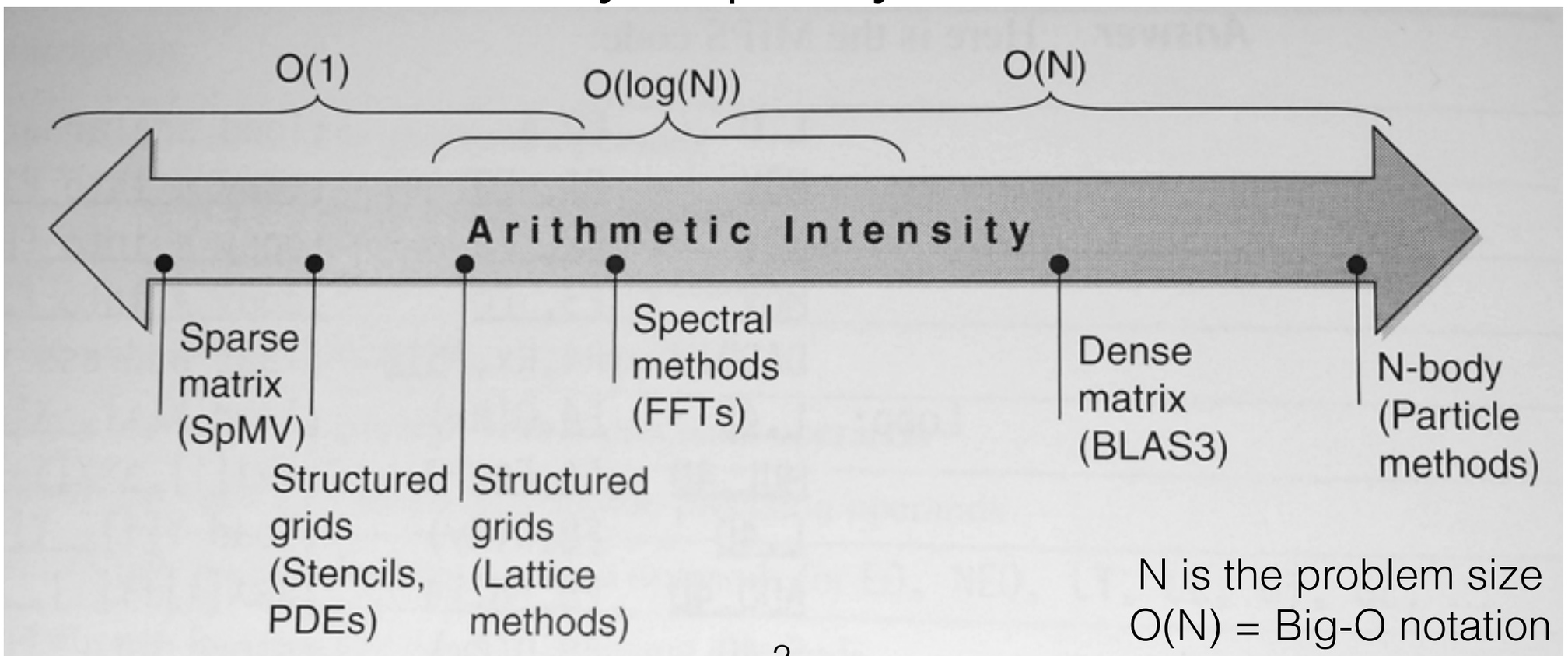
- **on the last year's lecture slides from Fabio Luporini (CO332/2014-2015)**
- **the course text, Hennessy and Patterson's Computer Architecture (5th ed.)**
- **the lecture slides from James Reinders (Intel) at ATPESC 2014**

Arithmetic Intensity

Processor	Type	Peak GFLOP/s	Peak GB/s	Ops/Byte	Ops/Word
E5-2690 v3 SP	CPU	416	68	~6	~24
E5-2690 v3 DP	CPU	208	68	~3	~24
K40 SP	GPU	4,290	288	~15	~60
K40 DP	GPU	1,430	288	~5	~40

Without enough Ops/Word codes are likely to be bound by operand delivery

Arithmetic intensity: Ops/Byte of DRAM traffic

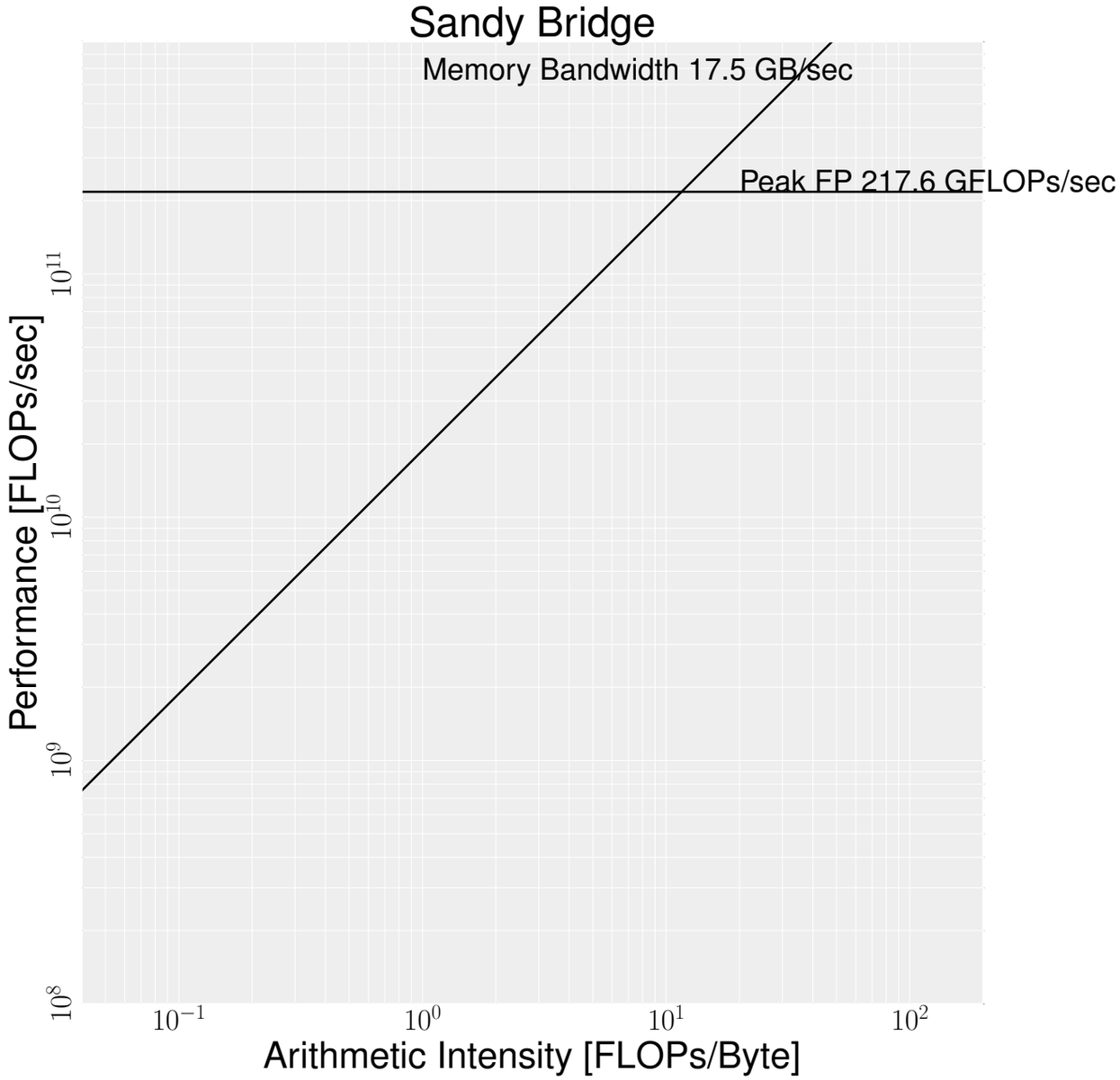


N is the problem size
O(N) = Big-O notation

Hennessy and Patterson's Computer Architecture (5th ed.)

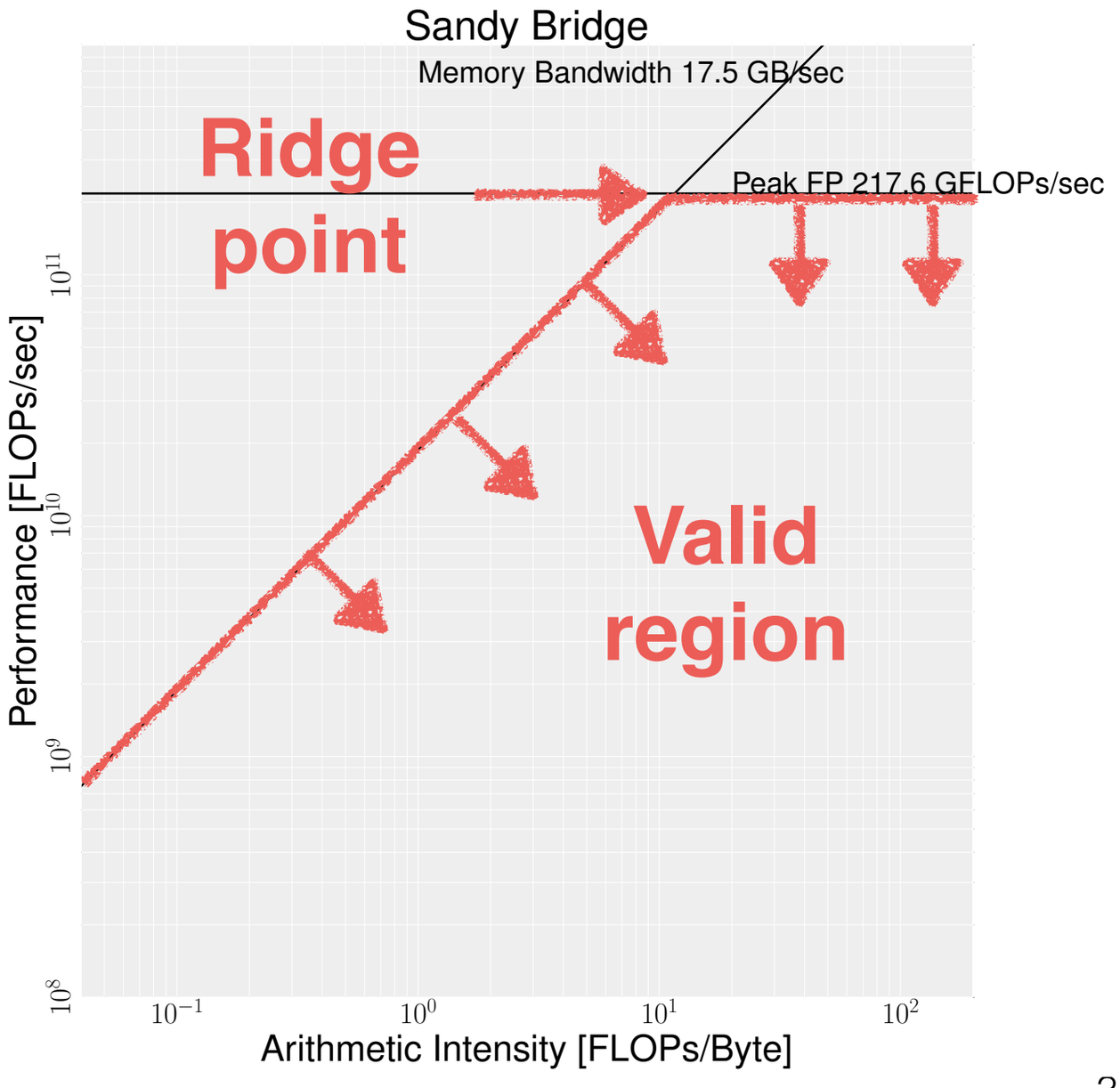
Roofline Model: Visual Performance Model

- Bound and bottleneck analysis (like Amdahl's law)
- Relates processor performance to off-chip memory traffic (bandwidth often the bottleneck)



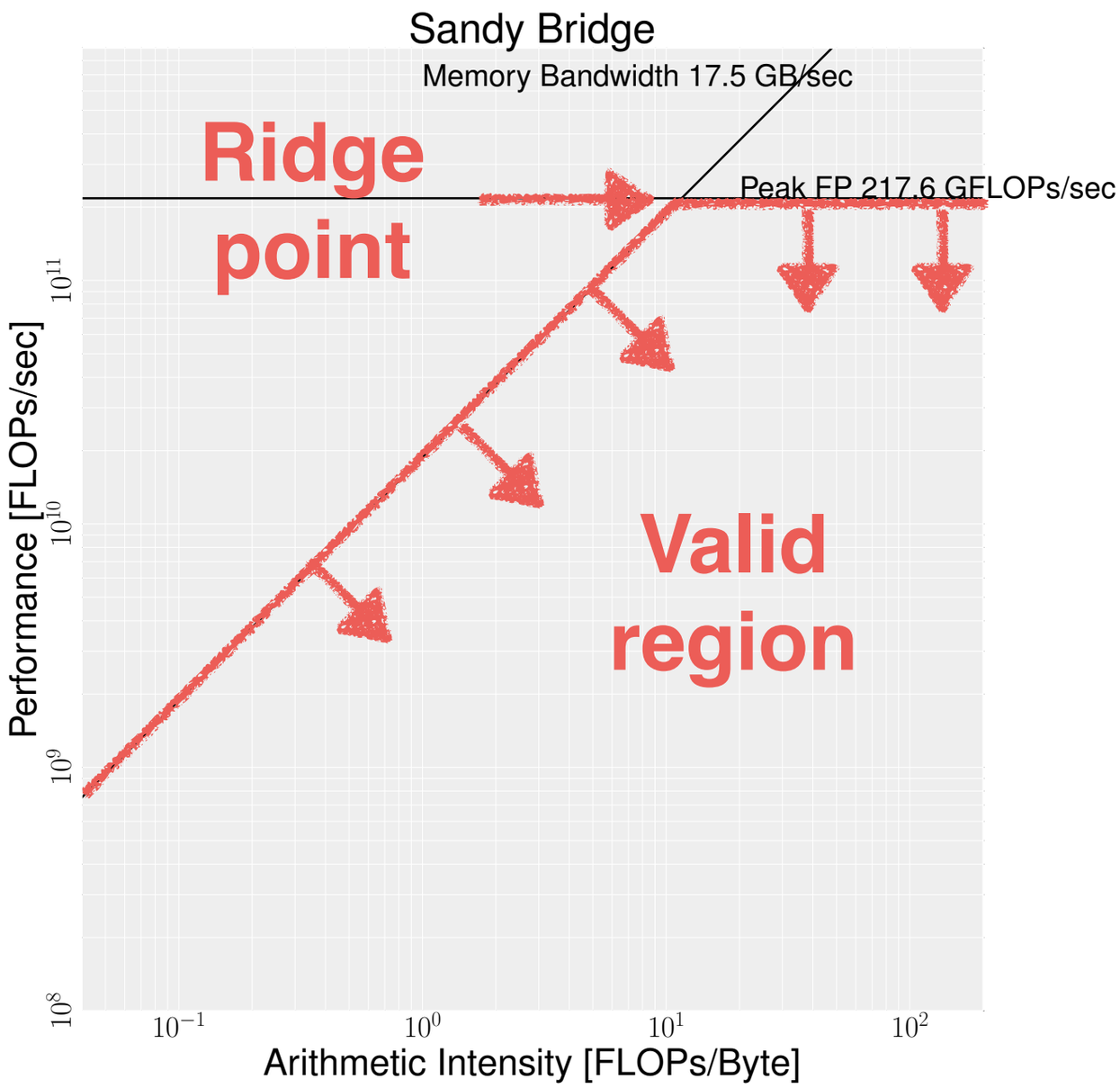
Roofline Model: Visual Performance Model

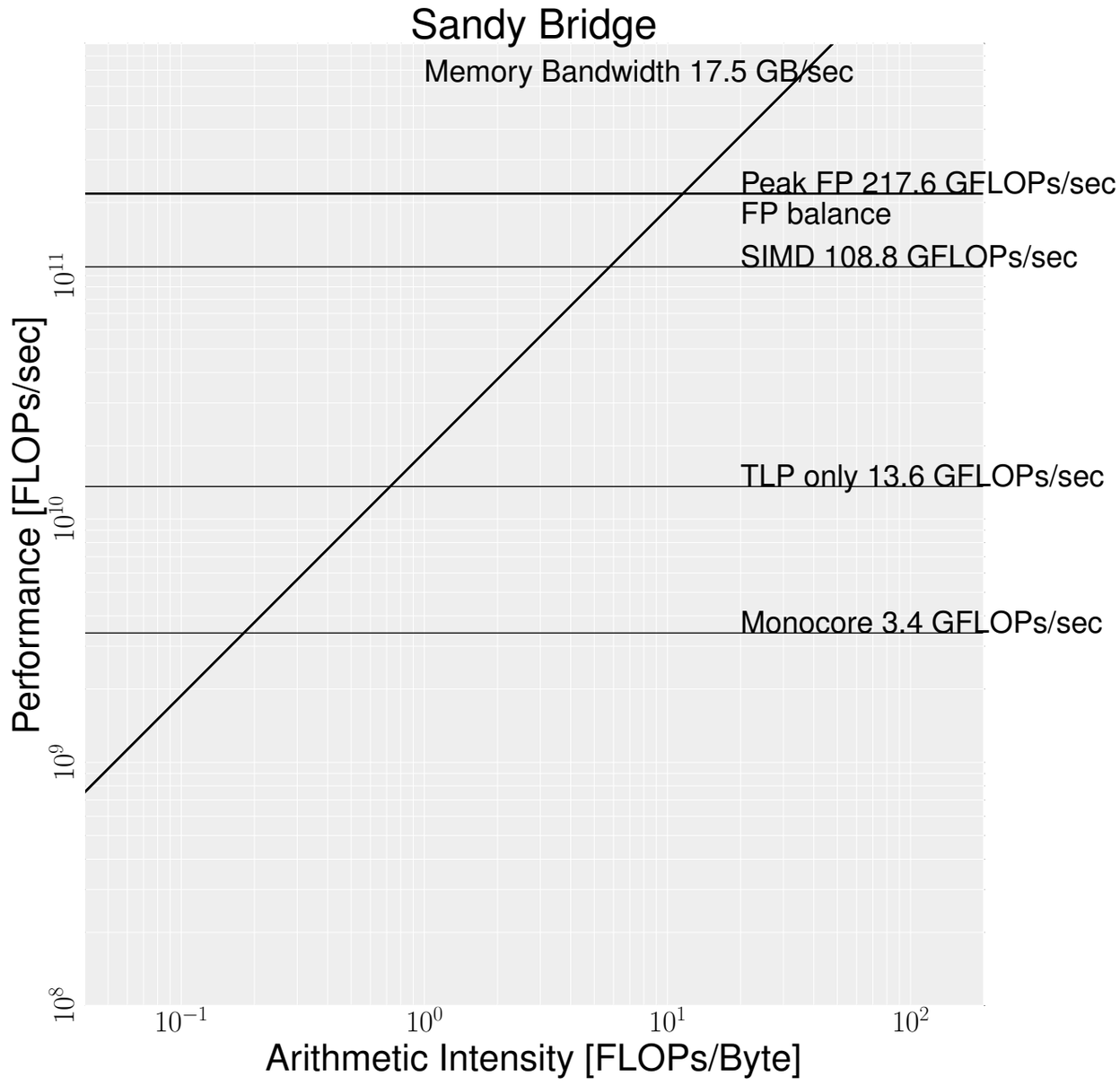
- Bound and bottleneck analysis (like Amdahl's law)
- Relates processor performance to off-chip memory traffic (bandwidth often the bottleneck)



Roofline Model: Visual Performance Model

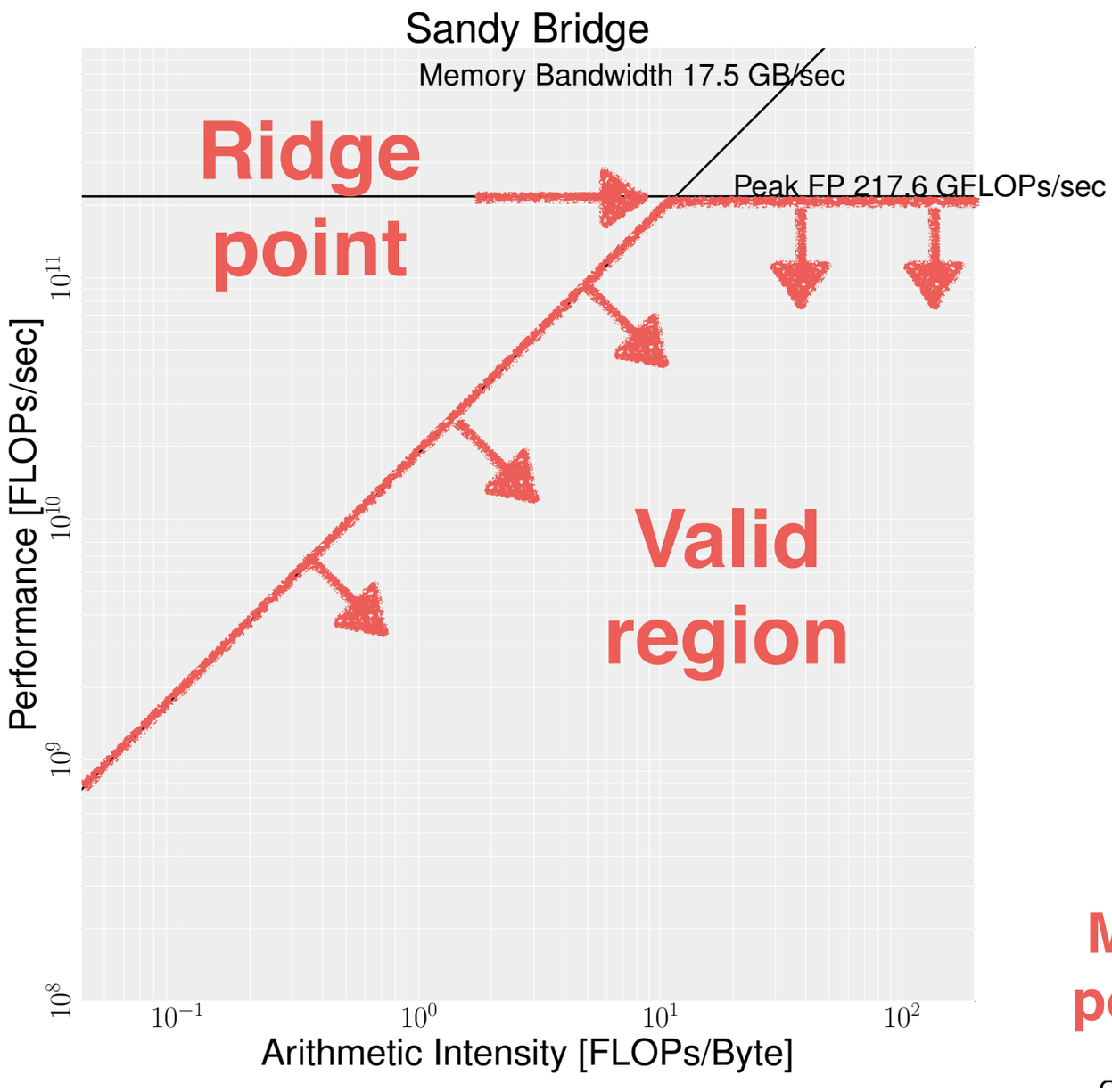
- Bound and bottleneck analysis (like Amdahl's law)
- Relates processor performance to off-chip memory traffic (bandwidth often the bottleneck)

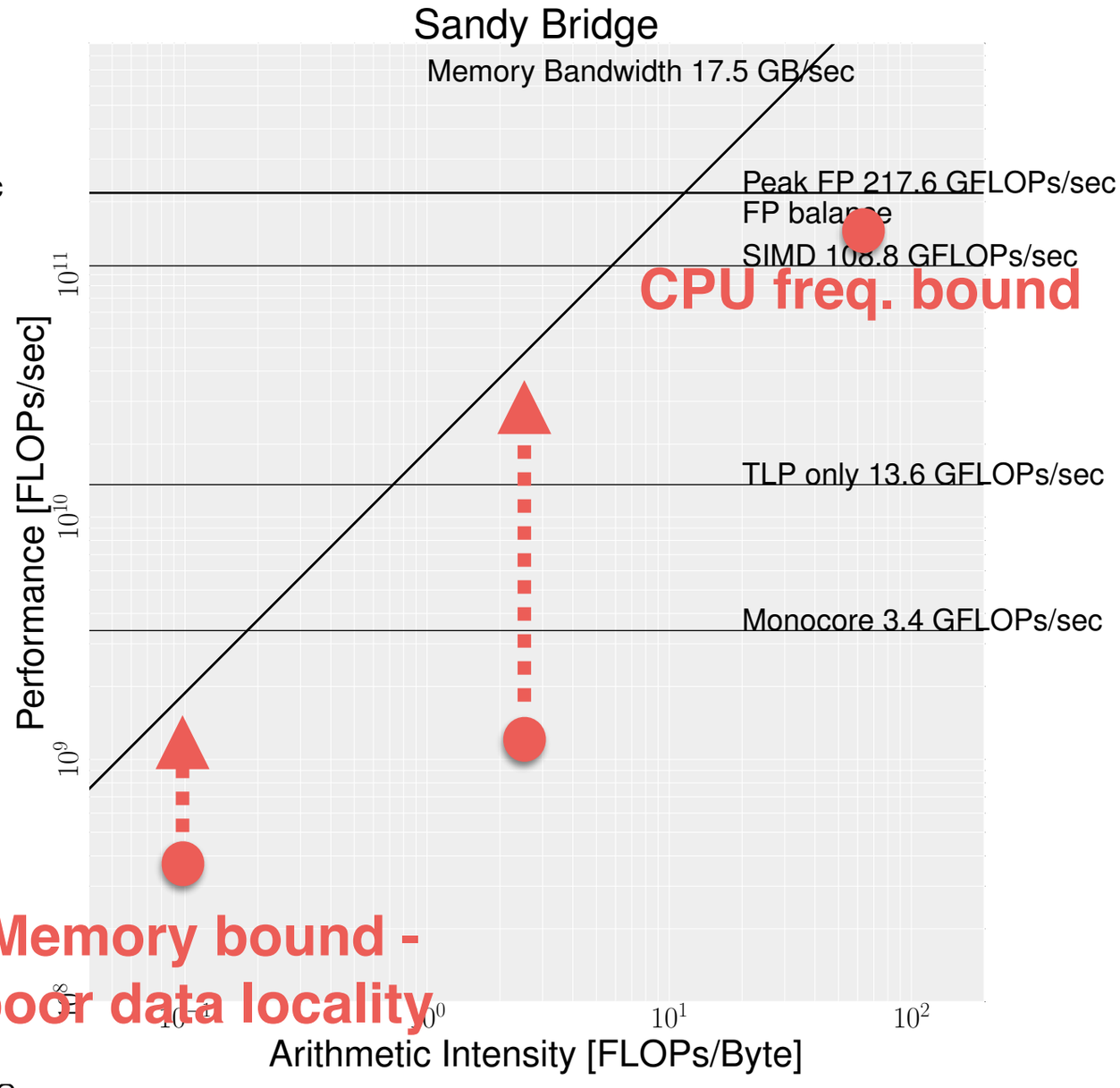




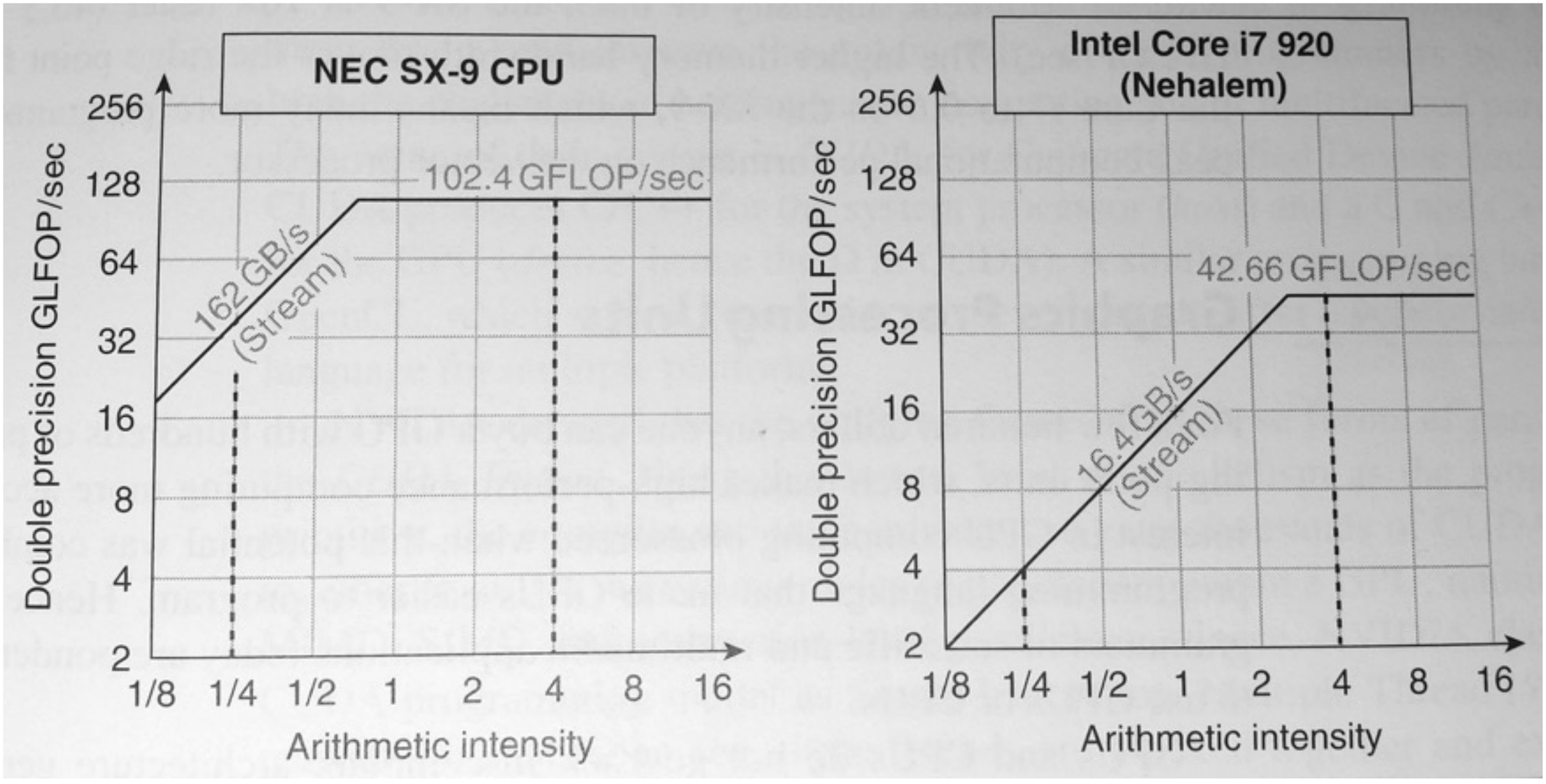
Roofline Model: Visual Performance Model

- Bound and bottleneck analysis (like Amdahl's law)
- Relates processor performance to off-chip memory traffic (bandwidth often the bottleneck)





Roofline Model: Visual Performance Model



Note that the ridge point offers insight into the computer's overall performance

Data-Level Parallelism

"A Single Instruction Multiple Data (SIMD) program
is a sequential ordering of data parallel instructions"

"… also called vector instructions"

citation source: M. Vanneschi (Prof. at University of Pisa)

Vector addition

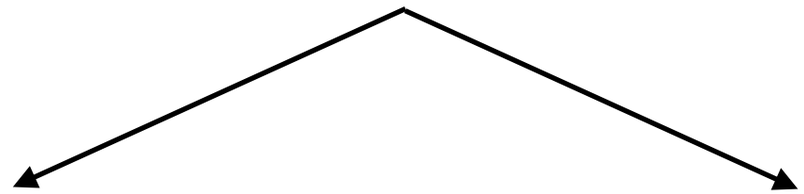
[3.2	1.1	2.2	4.5	9.0	6.7	3.4	4.2]	+
[1.2	3.1	2.5	3.6	0.5	3.2	2.2	3.5]	=
[4.4	4.2	4.7	8.1	9.5	9.9	5.6	7.7]	

Vector multiplication

[3.2	1.1	2.2	4.5	9.0	6.7	3.4	4.2]	x
[1.2	3.1	2.5	3.6	0.5	3.2	2.2	3.5]	=
[3.84	3.41	5.5	16.2	4.5	21.44	7.48	14.7]	

Data-Level Parallelism

$$c_0 = a_0 + b_0$$
$$c_1 = a_1 + b_1$$
$$c_2 = a_2 + b_2$$
$$c_3 = a_3 + b_3$$



"Scalar"

```
ADD Rc0, Ra0, Rb0
ADD Rc1, Ra1, Rb1
ADD Rc2, Ra2, Rb2
ADD Rc3, Ra3, Rb3
```

"SIMD/Vectorised"

```
VADD Rc, Ra, Rb
```

Single Instruction Multiple Data (SIMD)

- Vector processors
  Predates the other two by more than 30 years.

https://en.wikipedia.org/wiki/Vector_processor

- ISA extensions for multimedia
  e.g: Intel Pentium III, …, Haswell, AMD Jaguar, ARM Neon

http://www.filipekberg.se/2013/09/25/perfect-developer-laptop/

- Graphics Processing Units (GPUs), aka vectorisation SIMT
  (Single Instructions Multiple Threads)

http://www.nvidia.com/object/tesla-workstations.html

Example

```
void add (float *c, float *a, float *b)
{
    for (int i=0; i <= N; i++)
        c[i]=a[i]+b[i];
}
```

Sequential

```
Loop:
1.   LOAD a[i] -> Ra
2.   LOAD b[i] -> Rb
3.   ADD Ra, Rb -> Rc
4.   STORE Rc -> c[i]
5.   ADD i+1->i
6.   BNE i, N, Loop
```

Vector

```
Loop:
1.   LOADv4 a[i:i+3] -> Rva
2.   LOADv4 b[i:i+3] -> Rvb
3.   ADDv4 Rva, Rvb -> Rvc
4.   STOREv4 Rvc -> c[i:i+3]
5.   ADD i+4->i
6.   BNE i, N, Loop
```

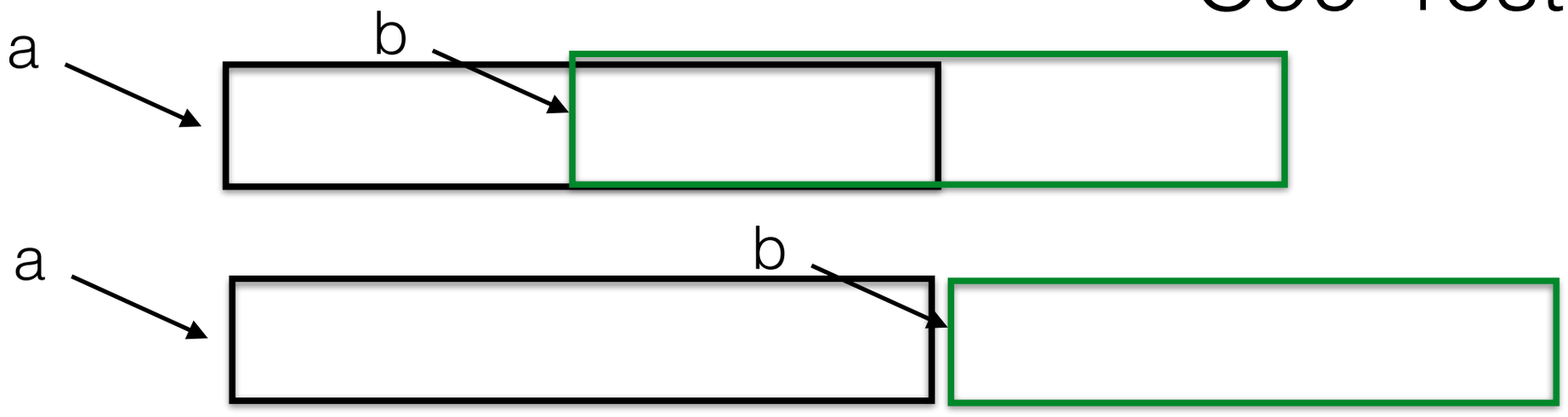
Problem:

is not legal to automatically vectorise this loop in C/C++ (without more information)

So, using a compiler switch for auto-vectorisation won't help

Choice 2: give compiler hints

C99 "restrict" keyword



```
void add (float *restrict c,
          float *restrict a,
          float *restrict b)
{
    for (int i=0; i <= N; i++)
        c[i]=a[i]+b[i];
}
```

During each execution of a function body in which a restricted pointer P is declared, if some object that is accessible through P is modified, then all accesses to that object in that block must occur through P, otherwise the behaviour is undefined

The compiler is free to ignore all aliasing implications of uses of restrict

Source: http://en.cppreference.com/w/c/language/restrict

Choice 3: ignore vector dependencies

ivdep pragma

```
void add (float *c, float *a, float *b)
{
 #pragma ivdep
     for (int i=0; i <= N; i++)
          c[i]=a[i]+b[i];
}
```

IVDEP (Ignore Vector DEPendencies) compiler hint.
Tells compiler “Assume there are no loop-carried dependencies”

Choice 4: code explicitly for vectors

OpenMP 4.0 pragmas

Option 1:

```
void add (float *c, float *a, float *b)
{
 #pragma omp simd
    for (int i=0; i <= N; i++)
       c[i]=a[i]+b[i];
}
```

Indicates that the loop can be transformed into a SIMD loop
(i.e. the loop can be executed concurrently using SIMD instructions)

Option 2:

```
#pragma omp declare simd
void add (float *c, float *a, float *b)
{
       *c=*a+*b;
}
```

"declare simd" can be applied to a function to enable SIMD instructions at the function level from a SIMD loop

Source: http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf

Choice 5: SIMD intrinsics

Lengths are hardcoded

```
void add (float *c, float *a, float *b)
{
    __m128* pSrc1 = (__m128*) a;
    __m128* pSrc2 = (__m128*) b;
    __m128* pDest = (__m128*) c;
    for (int i=0; i <= N/4; i++)
        *pDest++ = _mm_add_ps(*pSrc1++, *pSrc2++);
}
```

Choice 6: Vector Data Types with Overloading

OpenCL/CUDA vector data types: lengths are hardcoded

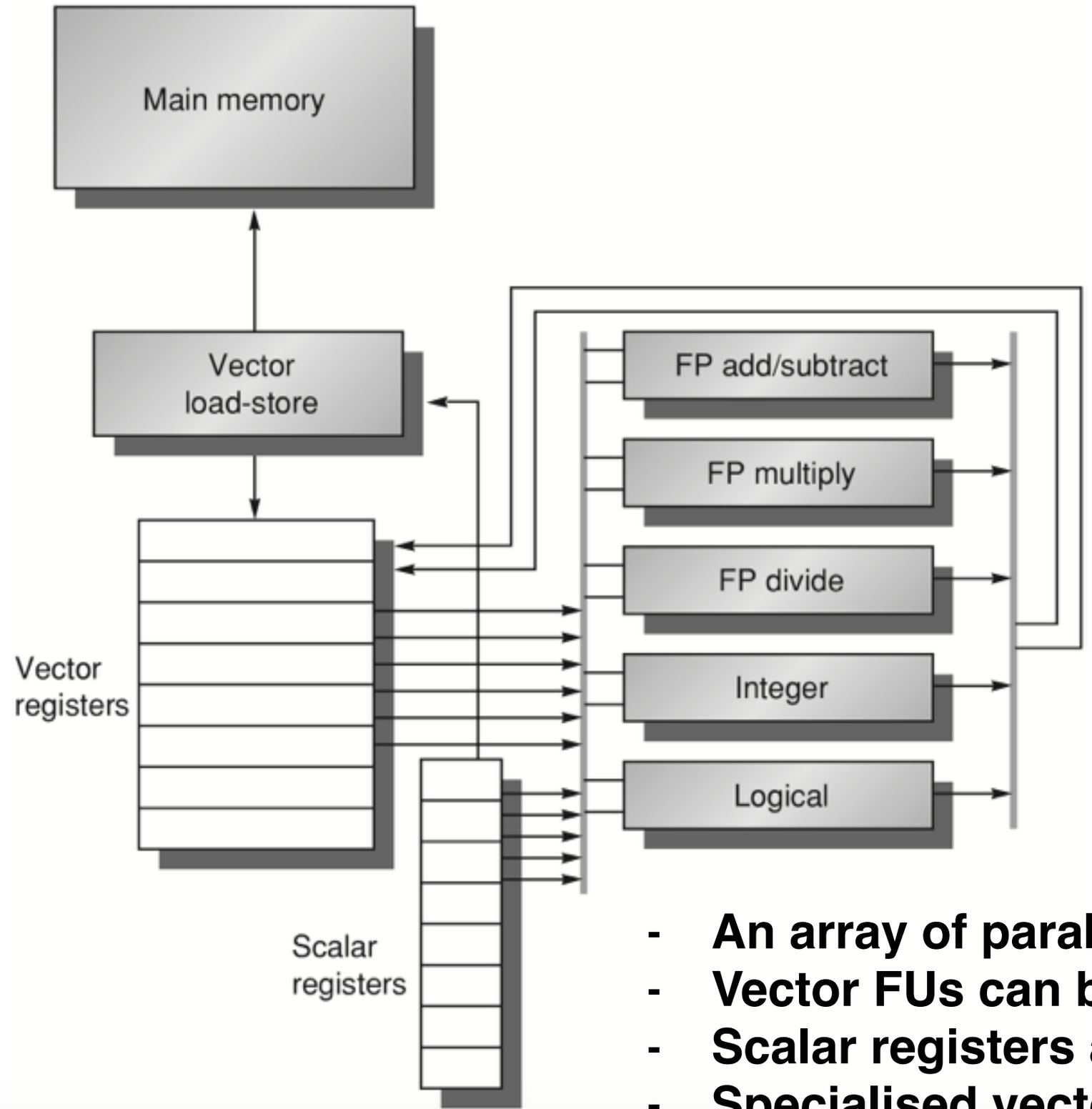
```
__kernel void add (__global float *c,
                   __global float *a,
                   __global float *b)
{
     /* We have reduced the global work size (n) by a factor of 4
        compared to the non vectorised OpenCL version.
        Therefore, i will now be in the range [0, (n / 4) - 1].
     */
     int id = get_global_id(0);
     /* Load 4 integers into 'a4' and 'b4'.
        The offset calculation is implicit from the size of the vector load.
        For vloadN(i, p), the address of the first data loaded would be
        p + i * N.
        Load from the address: a + i * 4 and b + i * 4.
     */
     float4 a4 = vload4(i, a);
     float4 b4 = vload4(i, b);
     /* Do the vector addition. Store the result at the address: c + i * 4.
     */
     vstore4(a4 + b4, i, c);
}
```

Source: http://malideveloper.arm.com/downloads/deved/tutorial/SDK/opencl/hello_world_vector_tutorial.html

Summary Vectorisation Solutions

1. Indirectly through high-level libraries/code generators
2. Auto-vectorisation (use “-O3 -mavx” and hope it vectorises):
   - sequential languages and practices gets in the way
   - use -ftree-vectorizer-verbose to report on the vectorisation
3. Give your compiler hints and hope it vectorises:
   - C99 "restrict" (implied in FORTRAN since 1956)
   - #pragma ivdep
4. Code explicitly:
   - In assembly language
   - SIMD instruction intrinsics
   - OpenMP 4.0 #pragma omp simd
   - Kernel functions:
     - OpenMP 4.0: #pragma omp declare simd
     - OpenCL or CUDA: more later

Vector Architecture



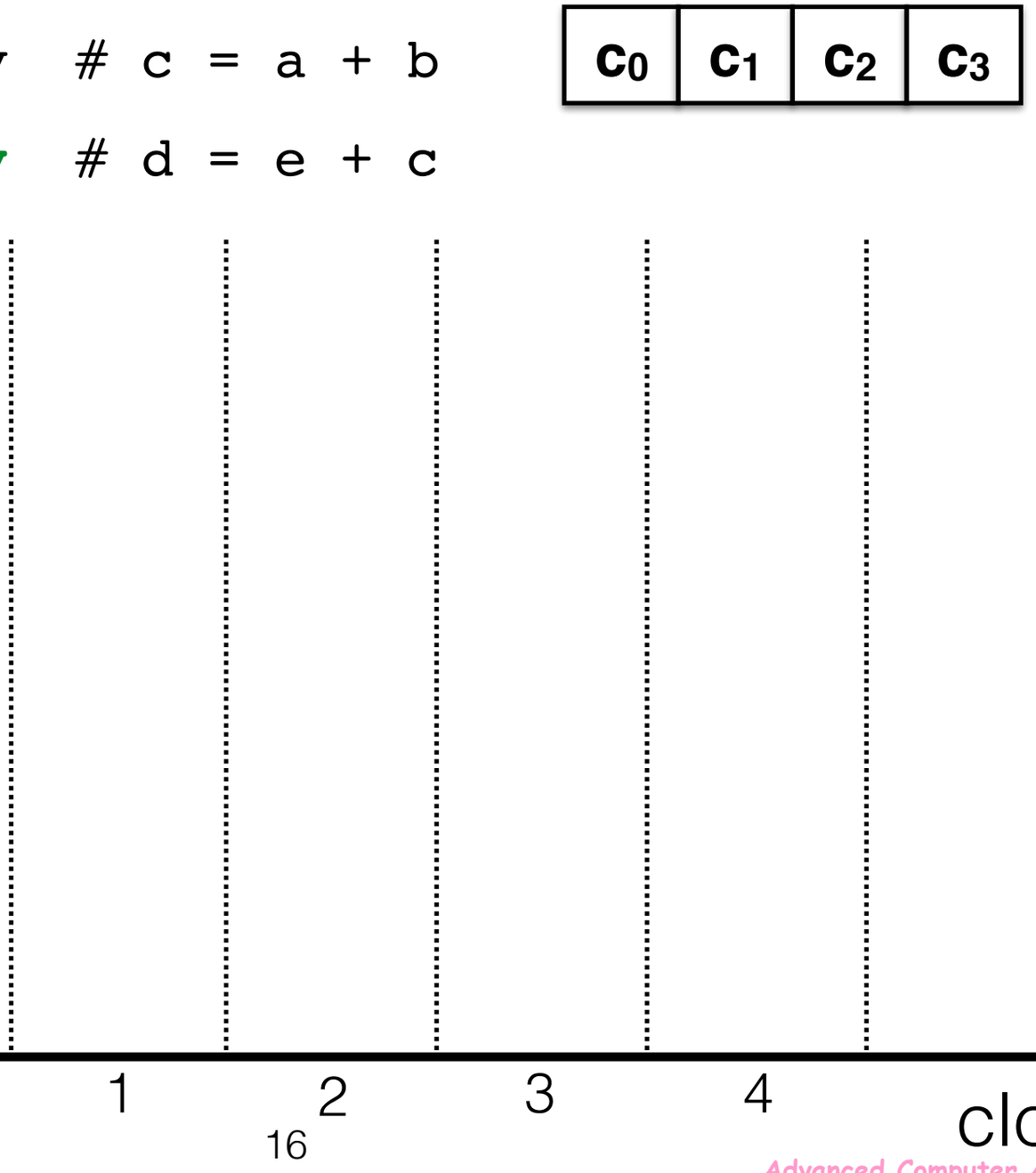
- **An array of parallel FUs**
- **Vector FUs can be pipelined**
- **Scalar registers and FUs needed**
- **Specialised vector memory system**

Source: Hennessy and Patterson's Computer Architecture (5th ed.)

Execution model and Vector chaining

Vector chaining is pipeline forwarding applied to SIMD architectures

```
VADD   cv, av, bv   # c = a + b
VMUL   dv, ev, cv   # d = e + c
```

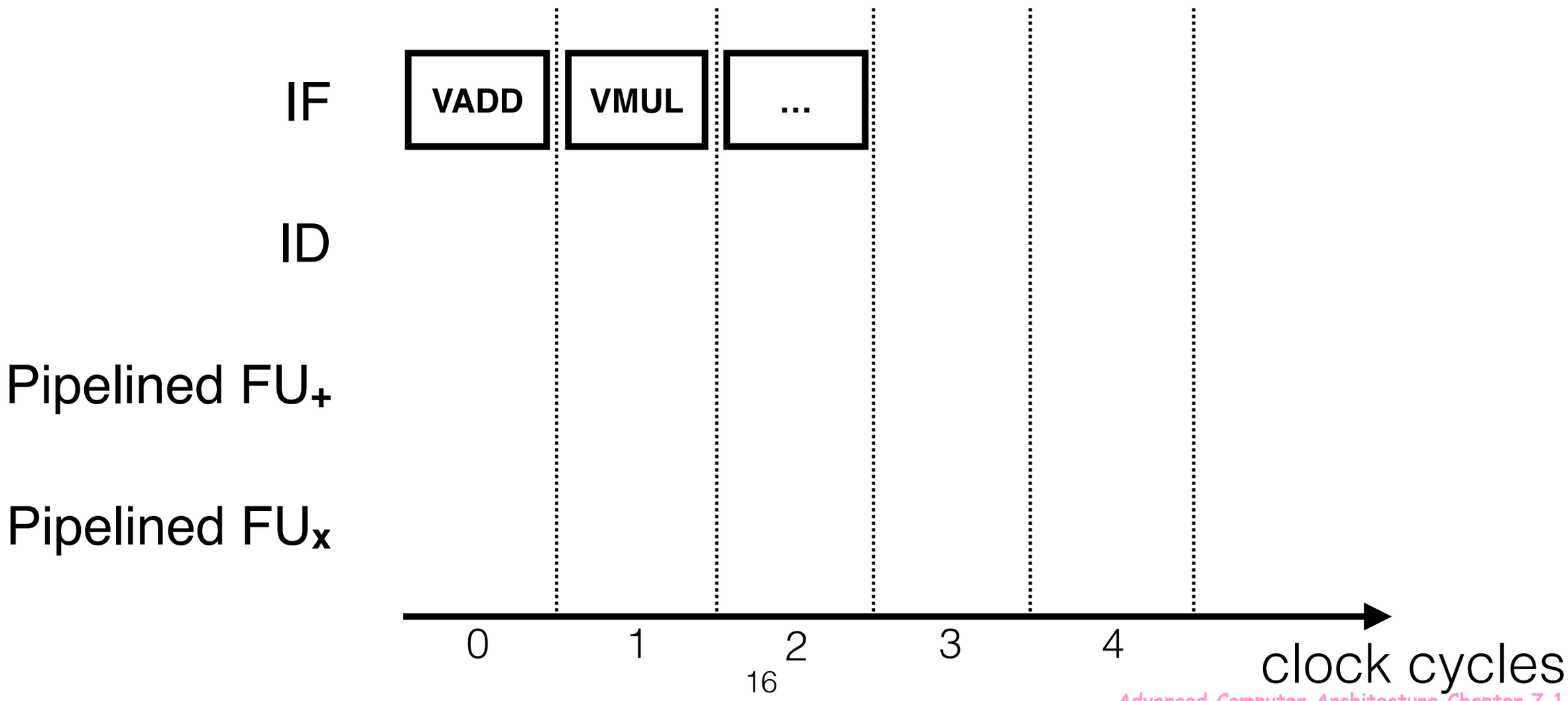


Execution model and Vector chaining

Vector chaining is pipeline forwarding applied to SIMD architectures

```
VADD   cv, av, bv   # c = a + b
VMUL   dv, ev, cv   # d = e + c
```

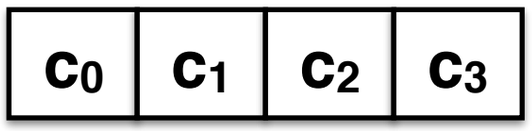
c_0	c_1	c_2	c_3

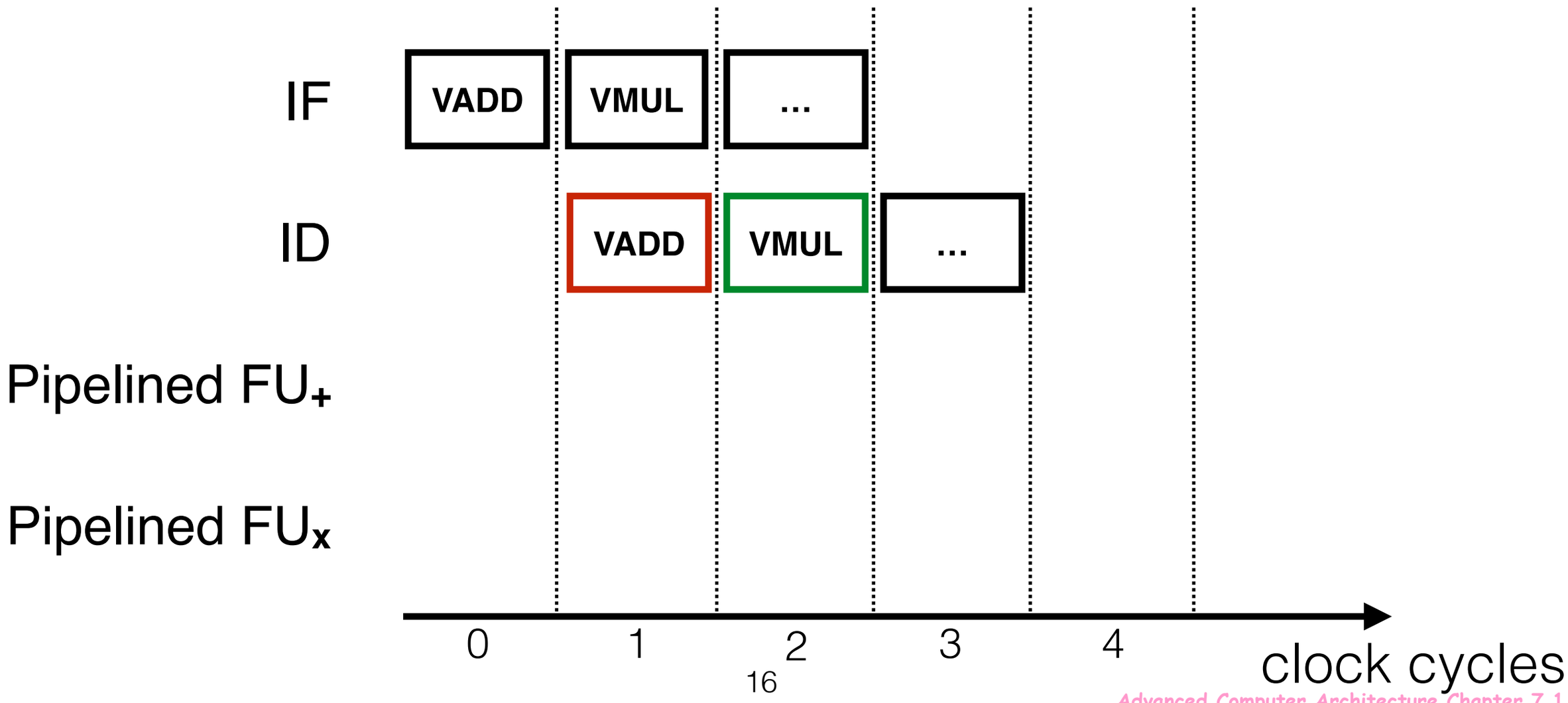


Execution model and Vector chaining

Vector chaining is pipeline forwarding applied to SIMD architectures

```
VADD   cv, av, bv  # c = a + b
VMUL   dv, ev, cv  # d = e + c
```

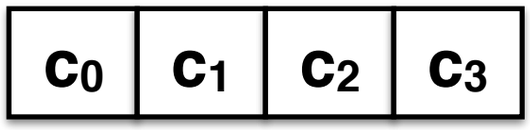


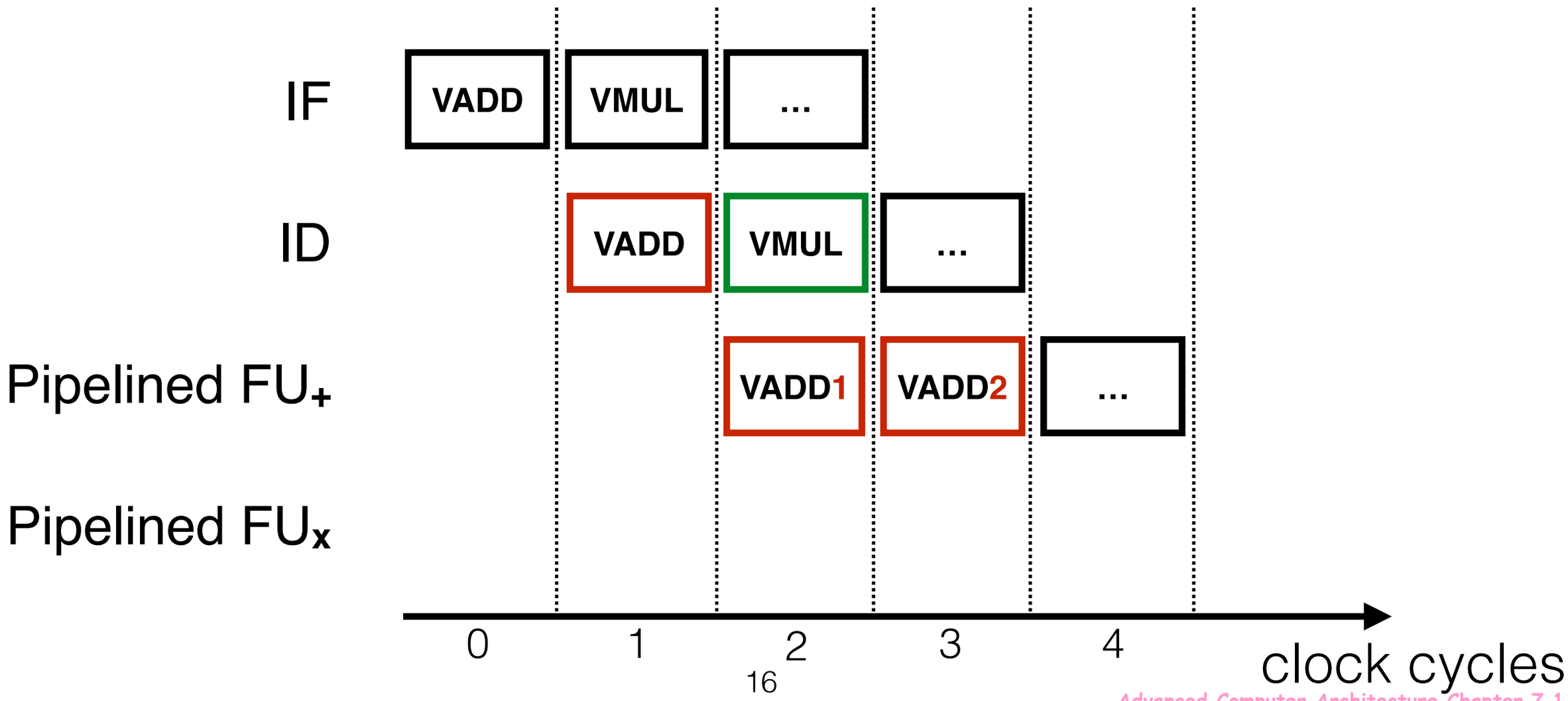


Execution model and Vector chaining

Vector chaining is pipeline forwarding applied to SIMD architectures

```
VADD   cv, av, bv  # c = a + b
VMUL   dv, ev, cv  # d = e + c
```





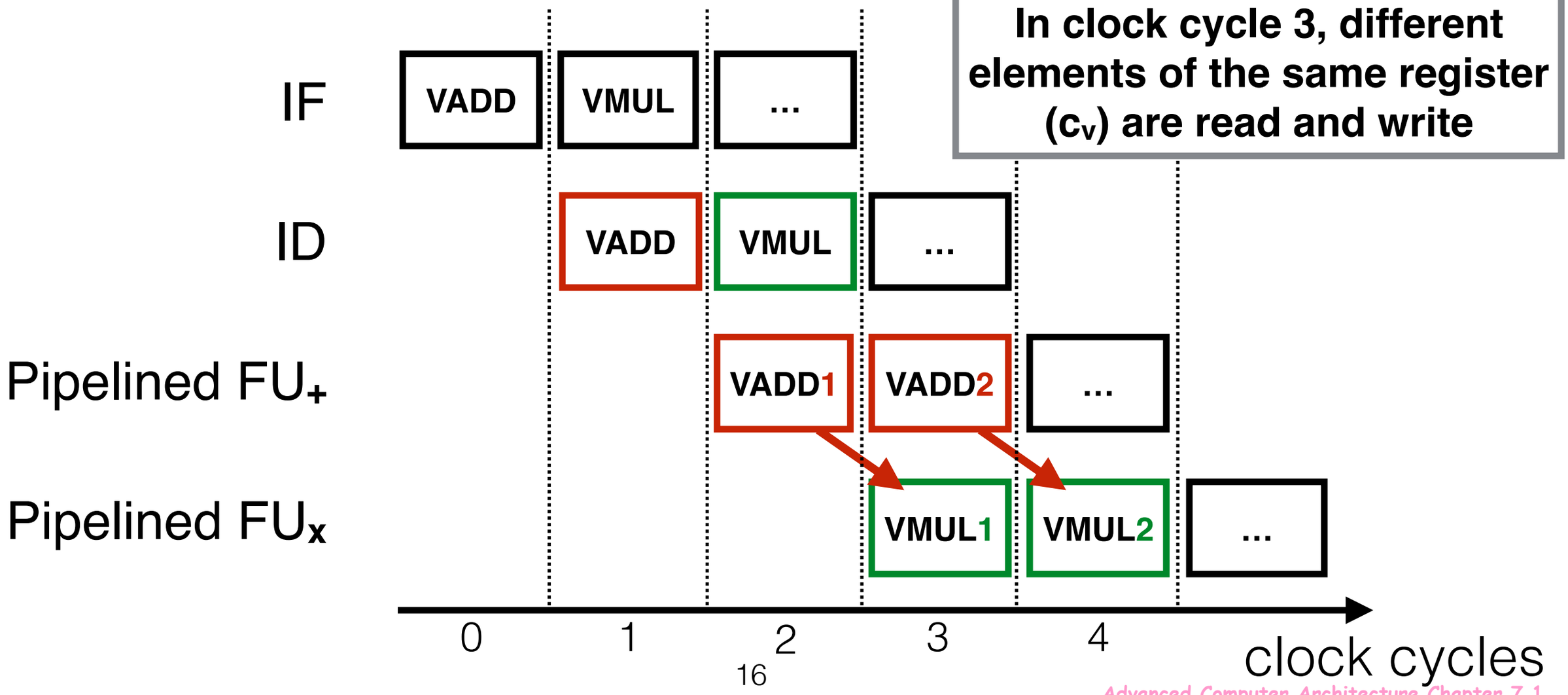
Execution model and Vector chaining

Vector chaining is pipeline forwarding applied to SIMD architectures

```
VADD   cv, av, bv   # c = a + b
VMUL   dv, ev, cv   # d = e + c
```

c_0	c_1	c_2	c_3

In clock cycle 3, different elements of the same register (c_v) are read and write



History:
Intel x86 ISA extended with SIMD

		width	Int.	SP	DP
1997	MMX	64	✔		
1999	SSE	128	✔	✔(x4)	
2001	SSE2	128	✔	✔	✔(x2)
2004	SSE3	128	✔	✔	✔
2006	SSSE 3	128	✔	✔	✔
2006	SSE 4.1	128	✔	✔	✔
2008	SSE 4.2	128	✔	✔	✔
2011	AVX	256	✔	✔(x8)	✔(x4)
2013	AVX2	256	✔	✔	✔
future	AVX-512	512	✔	✔(x16)	✔(x8)

ATPESC 2014, James Reinders: http://extremecomputingtraining.anl.gov/files/2014/08/20140804-1030-1115-ATPESC-Argonne-Reinders.2.pdf

History:

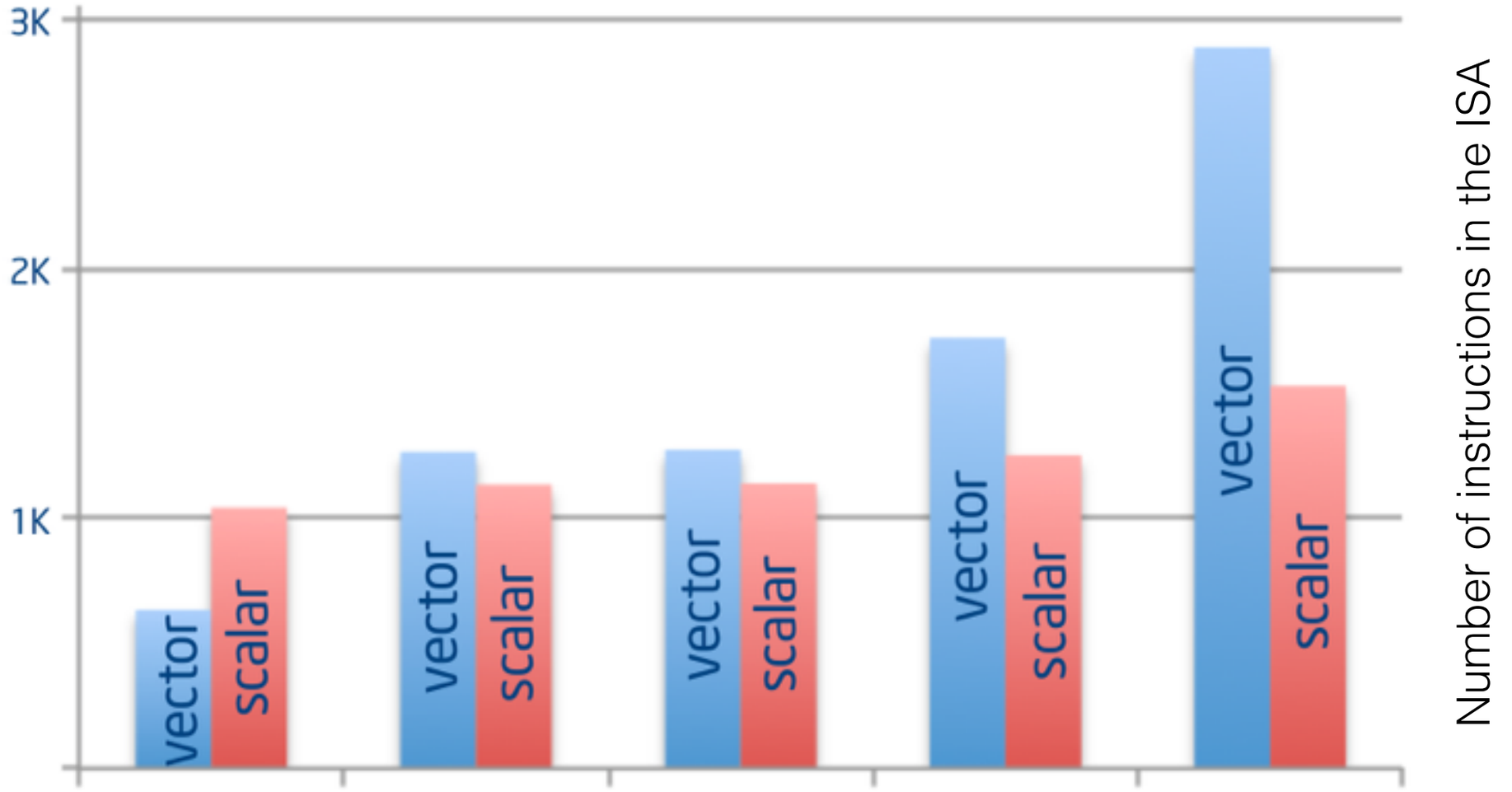
Intel x86 ISA ex

		width
1997	MMX	64
1999	SSE	128
2001	SSE2	128
2004	SSE3	128
2006	SSSE 3	128
2006	SSE 4.1	128
2008	SSE 4.2	128
2011	AVX	256
2013	AVX2	256
future	AVX-512	512

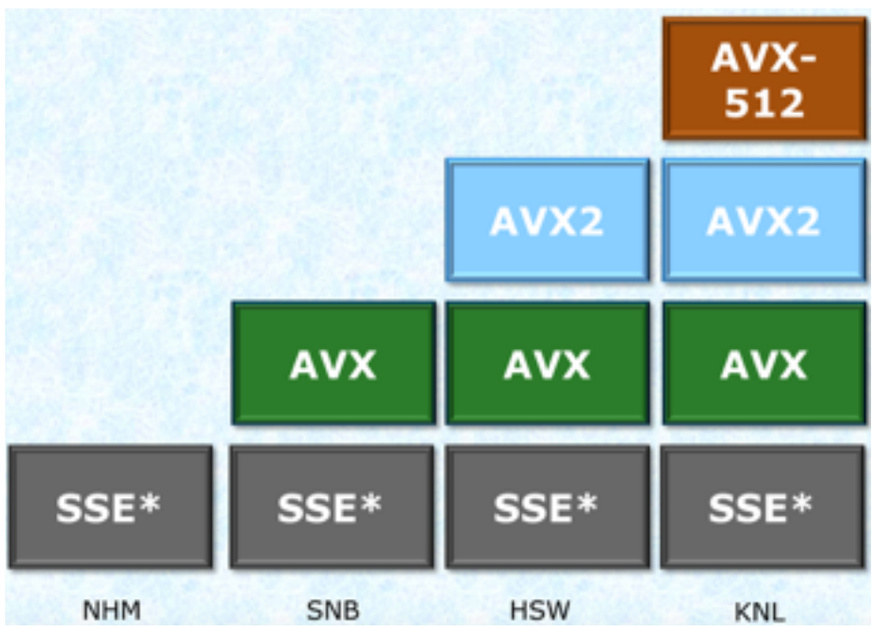
- **Wider registers**
  (from 32 to 512 bits)
- **More registers**
- **Richer instruction set**
  (predication, FMAs, gather, scatter, …)
- **Easier exploitation**
  (better compiler support, high-level functions, libraries…)

ATPESC 2014, James Reinders: http://extremecomputingtraining.anl.gov/files/2014/0 …rgonne-Reinders.2.pdf

Growth in vector instructions on Intel



Backwards compatibility accumulation



ATPESC 2014, James Reinders:
http://extremecomputingtraining.anl.gov/files/2014/08/20140804-1030-1115-ATPESC-Argonne-Reinders.2.pdf

Elena Demikhovsky (Intel): http://llvm.org/devmtg/2013-11/slides/Demikhovsky-Poster.pdf

Issues inherent in the computational model

Example 1

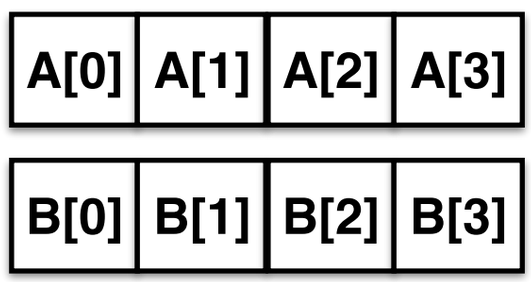
```
double A[N], B[N], C[N]
for i = 0 to N, i++
   C[i] = sqrt(A[i] + B[i])
```

SIMD version

```
loop: VLOAD  a_v, A[i:v]
      VLOAD  b_v, B[i:v]
      VADD   c_v, b_v, a_v
      VSQRT  c_v, c_v
      VSTORE C[i:v], c_v
      INCR i
      IF i<N/v: loop
```

Notation:

- **:v** indicates that the assembly operation is over v elements
- subscript **v** indicates that the register is actually a vector register, hosting **v** elements



E.g. **v**=4

Simple issues: bad array size

```
loop: VLOAD a_v, A[i:v]
      VLOAD b_v, B[i:v]
      VADD c_v, b_v, a_v
      VSQRT c_v, c_v
      VSTORE C[i:v], c_v
      INCR i
      IF i<N/v: loop
```

<u>Issue 1</u>: N might not be a multiple of the vector length v

or

N is known only at runtime

Simple issues: bad array size

```
loop: VLOAD a_v, A[i:v]
      VLOAD b_v, B[i:v]
      VADD c_v, b_v, a_v
      VSQRT c_v, c_v
      VSTORE C[i:v], c_v
      INCR i
      IF i<N/v: loop
      IF N%v==0: exit
peel: LOAD a, A[v*i + 0]
      …
      …
exit: …
```

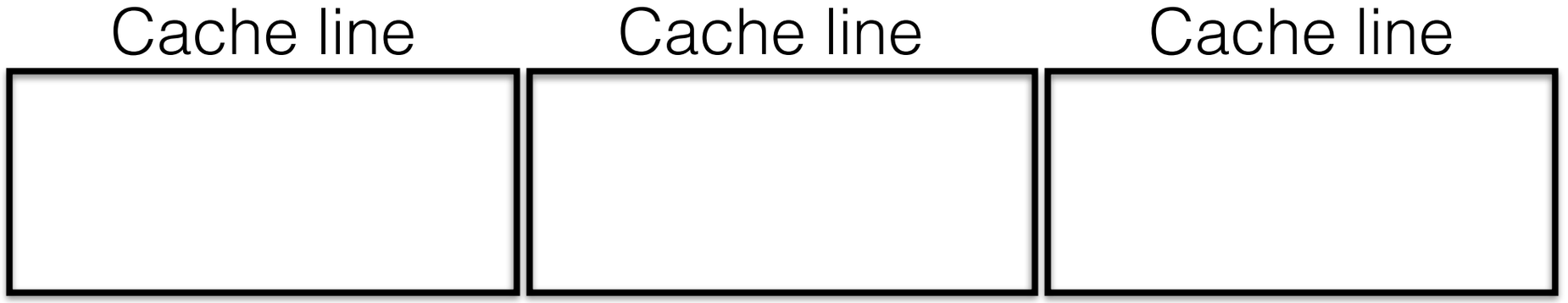
<u>Issue 1</u>: N could not be a multiple of the vector length v

or

N is known only at runtime

Medium issues: data alignment

```
loop: VLOAD a_v, A[i:v]
      VLOAD b_v, B[i:v]
      …
      VSTORE C[i:v], c_v
```



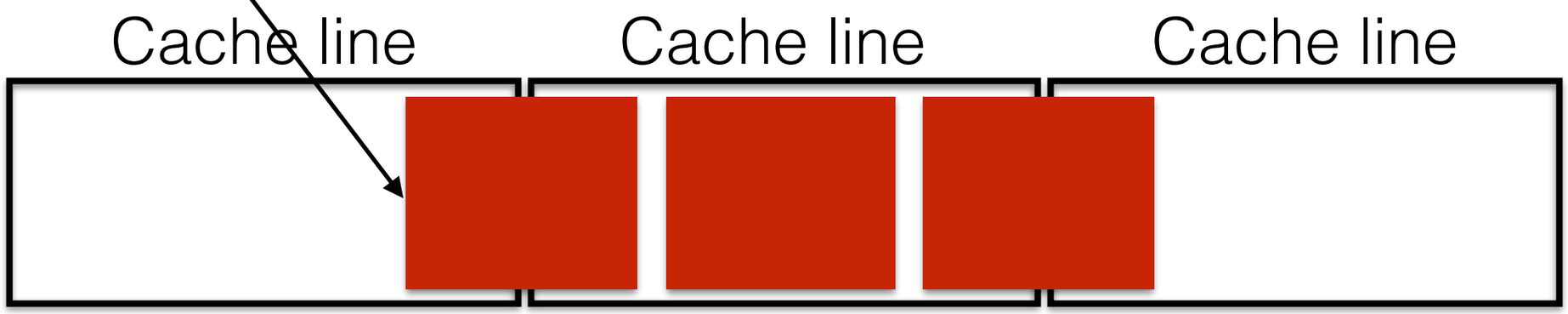
E.g.: AVX on Sandy Bridge: Cache line: 64B, vector length: 32B, double: 8B

Issue 2: Memory accesses should be aligned to page and cache boundaries

Medium issues: data alignment

```
loop: VLOAD a_v, A[i:v]
      VLOAD b_v, B[i:v]
      …
      VSTORE C[i:v], c_v
```

Base address of array A



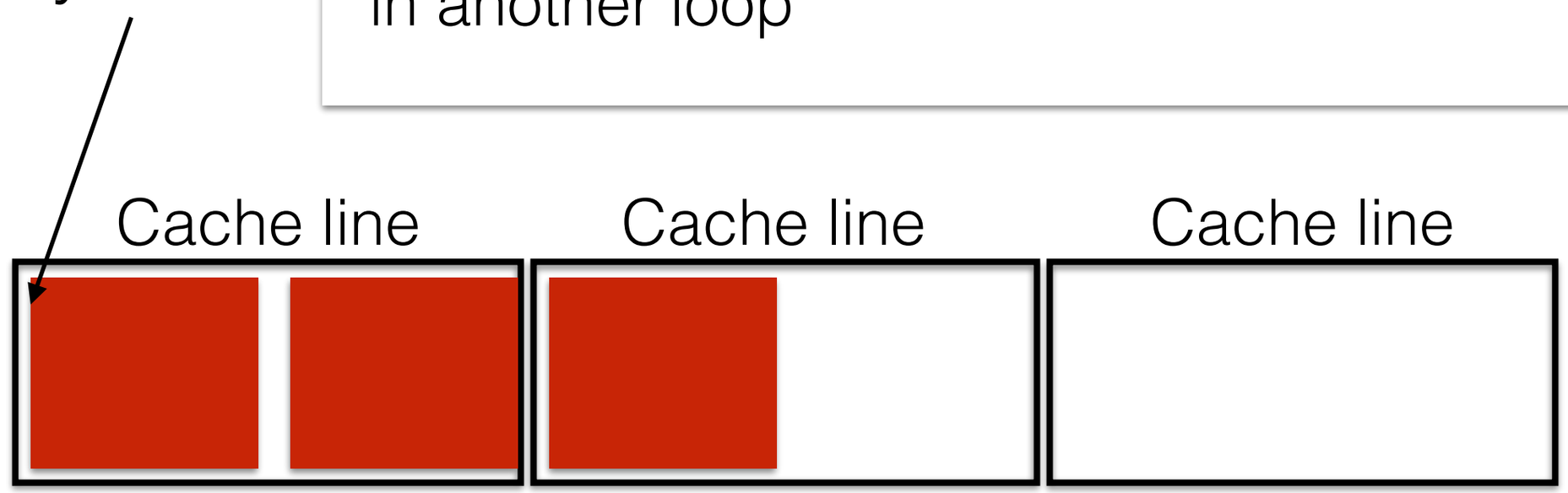
E.g.: AVX on Sandy Bridge: Cache line: 64B, vector length: 32B, double: 8B

<u>Issue 2</u>: Memory accesses should be aligned to page and cache boundaries

Medium issues: data alignment

Solution: change the allocation point of A

- Use of special mallocs or special array qualifiers
- **Global transformation**: might affect alignment in another loop



E.g.: AVX on Sandy Bridge: Cache line: 64B, vector length: 32B, double: 8B

<u>Issue 2</u>: Memory accesses should be aligned to page and cache boundaries (tricky with stencils)

Advanced issues: bad access patterns

Example 2

```
double A[N], B[N], C[N], D[N]
for i = 0 to N, i++
    C[i] = A[2*i] + B[D[i]]
```

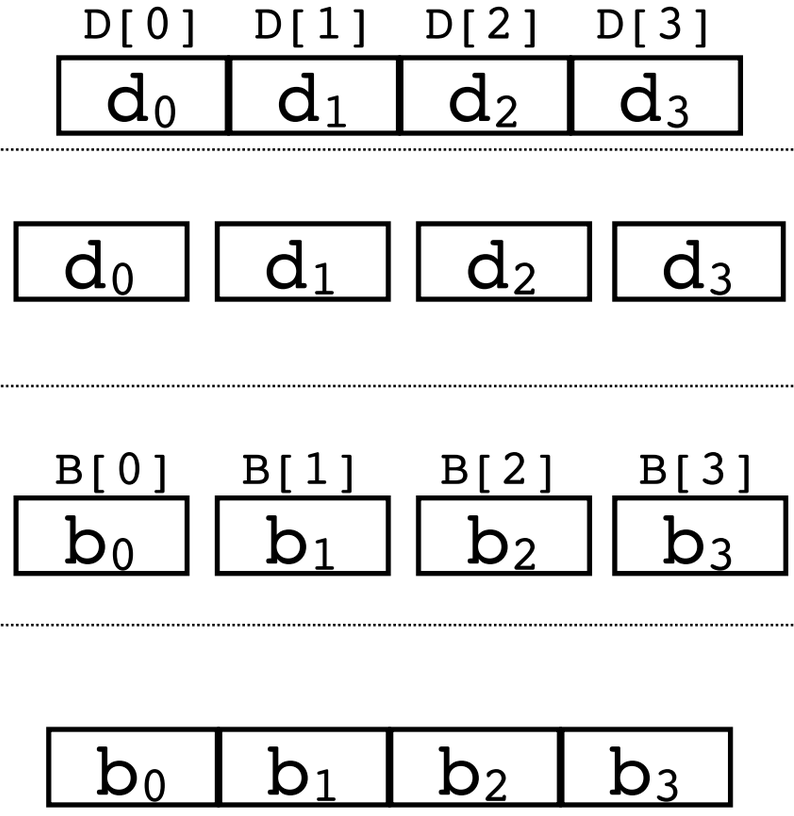
SIMD version

```
loop: VLOAD a_v, A[i], stride=2
      VGATHER b_v, B, D[i:v]
      VADD c_v, b_v, a_v
      VSTORE C[i:v], c_v
incr: INCR i
      IF i<N/v: loop
```

Advanced issues: bad access patterns

```
B[D[i]]  ===>  VGATHER b_v, B, D[i:v]
```

```
VGATHER_v=4: VLOAD d_v, D[i:v]
             UNPACK_0 d_0, d_v
             UNPACK_1 d_1, d_v
             UNPACK_2 d_2, d_v
             UNPACK_3 d_3, d_v
             LOAD b_0, B[d_0]
             LOAD b_1, B[d_1]
             LOAD b_2, B[d_2]
             LOAD b_3, B[d_3]
             PACK_0 b_v, b_0
             PACK_1 b_v, b_1
             PACK_2 b_v, b_2
             PACK_3 b_v, b_3
```



<u>Issue 3</u>: regardless of the ISA the (micro-)interpretation of these instructions is expensive

Advanced issues: branch divergence

Example 3

```
double A[N], B[N], C[N]
for i = 0 to N, i++
    if f(C[i]) > 0
        C[i] = A[i] + B[i]
```

SIMD version

```
loop: VLOAD a_v, A[i:v]
      VLOAD b_v, B[i:v]
      VLOAD c_v, C[i:v]
      IF f(c_v) <= 0: incr
      VADD c_v, b_v, a_v
      VSTORE C[i:v], c_v
incr: INCR i
      IF i<N/v: loop
```

Advanced issues: branch divergence

```
loop: VLOAD av, A[i:v]
      VLOAD bv, B[i:v]
      VLOAD cv, C[i:v]
      IF f(cv) <= 0: incr
      VADD cv, bv, av
      VSTORE C[i:v], cv
incr: INCR i
      IF i<N/v: loop
```

<u>Issue 4</u>: Need architectural support to handle branches

Solution: **Predication through masking**

Add a new boolean vector register (the vector mask register)

- Operates on elements whose corresponding bit in the mask is 1
- Requires ISA extension to set the mask register

Advanced issues: branch divergence

```
for i = 0 to 63, i++
    if A[i] > 0
        B[i] = A[i]*4
```

```
loop: VLOAD     a_v, A[i:v]
VCMP_P    R_mask, a_v, R_0
VMUL_P    b_v{R_mask}, a_v, R_4
VSTORE_P  B[i:v]{R_mask}, b_v
VRESET_P  R_mask
INCR      R_i
CMP       R_i < 64/v: loop
```

 source: http://www.inf.ed.ac.uk/teaching/courses/pa/Notes/lecture11-vector.pdf

Interesting examples in real programs

```
double s, A[64];

for i = 0 to N-1, i++
   A[i] = A[i] + s
```

Statically unknown loop size

```
for i = 0 to 63, i+=k
   A[i] = A[i] + s
```

k-strided memory accesses
(*k* can be known or not)

```
for i = 0 to 63, i++
   A[i] += B[C[i]] * s
```

Irregular and statically unknown memory access pattern

```
for i = 0 to 63, i++
   if A[i] > 0
      s += A[i]
```

Divergence

Interesting examples in real programs

```
double s, A[64], B[64];

for i = 0 to 63, i++
    s += A[i]
```

Global reductions

```
for i = 0 to 62, i++
    A[i+1] = A[i] * s
```

Loop-carried dependency

```
for i = 0 to 63, i++
    tmp = A[i] * s
    B[i] = tmp * tmp
```

Read-after-write dependency

```
for i = 0 to 63, i++
    tmp = foo(A[i])
    B[i] = A[i] + tmp
```

Function calls in the loop body

Common pitfalls of compiler's autovectorisation

```
for i = 0 to N, i++
    if (A[i] > 0 &&
        B[i] < ths)
        s += A[i]
    else
      if (A[i] < M)
          s -= A[i]
```

Complex, possibly nested branches

```
for i = 0 to 63, i++
   A[B[i]] += C[i] * s
```

Gather/scatter access pattern, even with ISA support

```
for i = 0 to 63, i++
   A[i] += FOO(A[i], b, c)
```

Non-trivial function calls

Pros of SIMD Architectures

- **Increase arithmetic operations execution (multiple FUs)**

- **Reduced pressure on instruction fetch and issue**
  - ***Fewer instructions are necessary to specify the same amount of work***
  - ***Much simpler hardware for checking dependences***

- **Generally more power efficient than MIMD architectures**
  - ***Multiple Instructions Multiple Data (MIMD)***
  - ***MIMD fetches one instruction per data operation***

- **Programmer continues to think sequentially**
  - ***Not so easy though, unfortunately***

source: http://www.inf.ed.ac.uk/teaching/courses/pa/Notes/lecture11-vector.pdf

Cons of SIMD Architectures

- **Still requires integer and FP scalar units for the non-vector operations (Turing tax - space on chip)**
- **Compiler or programmer has to vectorise programs**
- **Not suitable for many classes of applications**
- **May require a specialised high-bandwidth memory system**
  - **Usually built around heavily banked memory with data interleaving**
- **In some cases, ISA explosion**

source: http://www.inf.ed.ac.uk/teaching/courses/pa/Notes/lecture11-vector.pdf