Parallel Computing: Vector and SIMD Programming

Raymond Namyst, Pierre-André Wacrenier

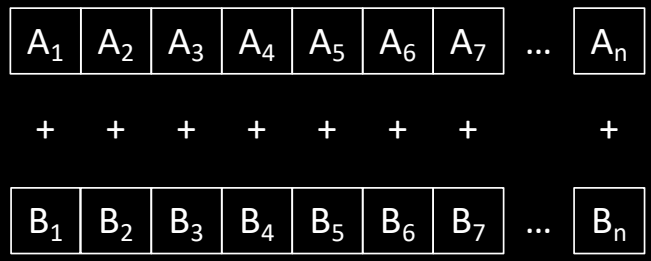
Dept. of Computer Science

University of Bordeaux, France

https://gforgeron.gitlab.io/pap/

Data-level Parallelism

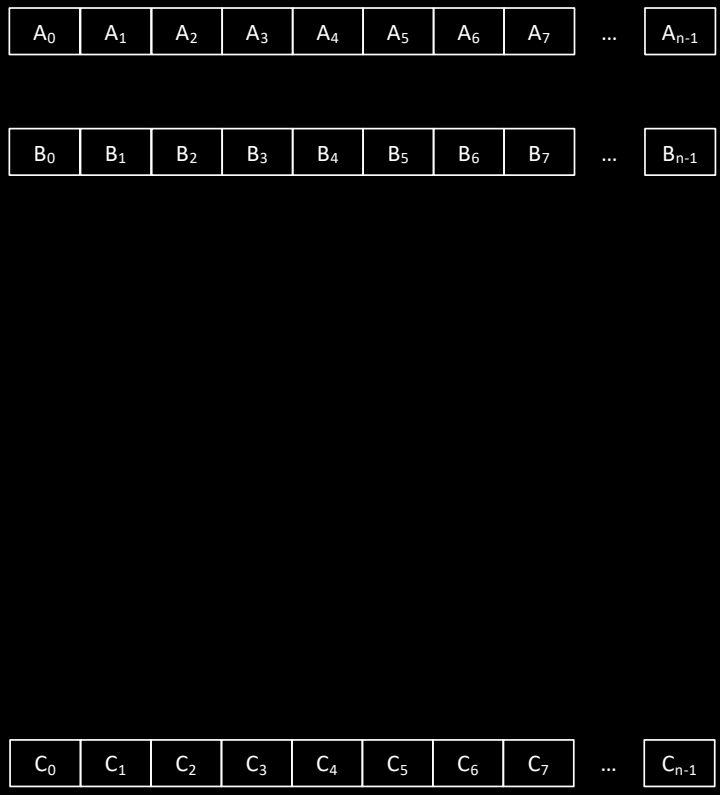
- In many applications, parallelism comes from applying the same operations simultaneously across large sets of data
  - One single thread of control, parallel data operations
    - Matrix- and Vector-oriented programs
    - Image and Sound processing



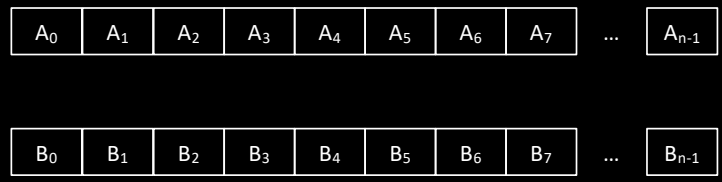
Scalar processors

- Let us consider a simple vector addition

```
static float A[N], B[N], C[N];

for (int i = 0; i < N; i++)
  C[i] = A[i] + B[i];
```



Scalar processors



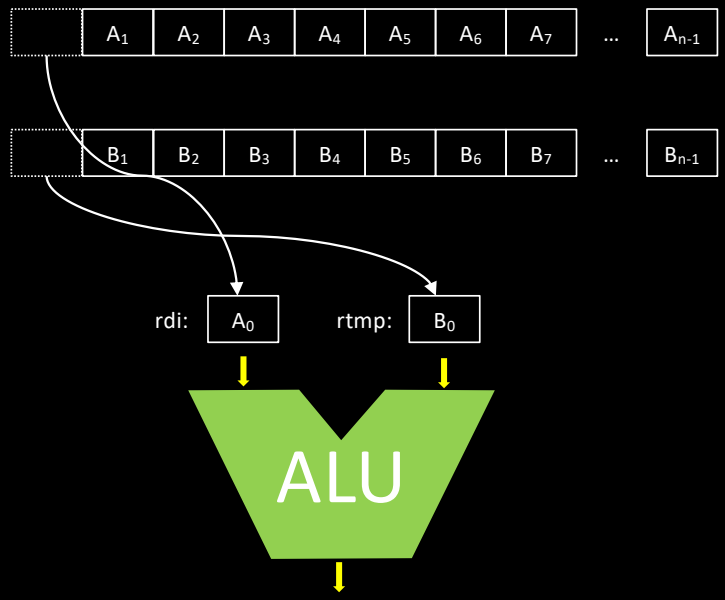
- Let us consider a simple vector addition

```
static float A[N], B[N], C[N];

xorq %rax, %rax
leaq _A(%rip), %rcx
leaq _B(%rip), %rdx
leaq _C(%rip), %rsi
loop:
  movss (%rax,%rcx), %rdi
  addss (%rax,%rdx), %rdi
  movss %rdi, (%rax,%rsi)
  addq $4, %rax
  cmpq 4*N, %rax
  jne loop
```

Scalar processors

- Let us consider a simple vector addition

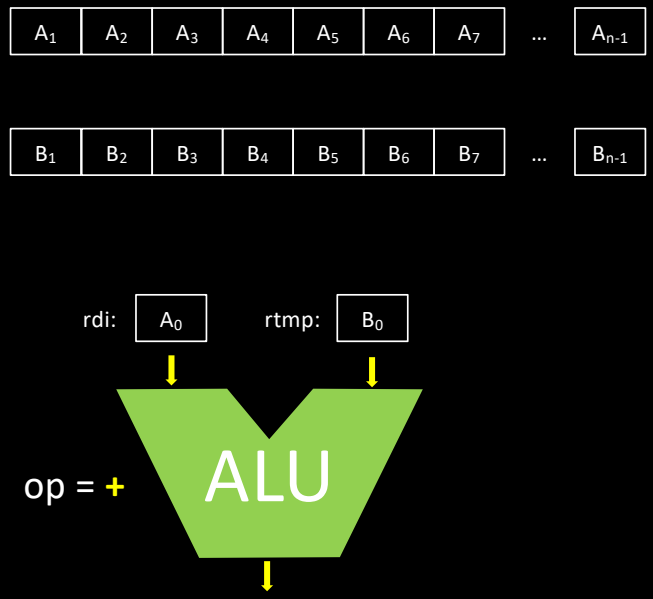
```
static float A[N], B[N], C[N];

xorq %rax, %rax
leaq _A(%rip), %rcx
leaq _B(%rip), %rdx
leaq _C(%rip), %rsi
loop:
  movss (%rax,%rcx), %rdi
  addss (%rax,%rdx), %rdi
  movss %rdi, (%rax,%rsi)
  addq $4, %rax
  cmpq 4*N, %rax
  jne loop
```



Scalar processors

- Let us consider a simple vector addition

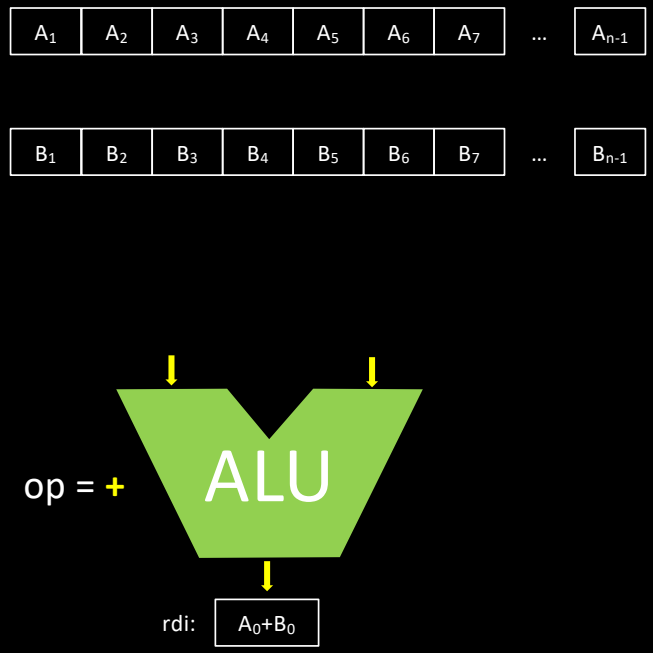
```
static float A[N], B[N], C[N];

xorq %rax, %rax
leaq _A(%rip), %rcx
leaq _B(%rip), %rdx
leaq _C(%rip), %rsi
loop:
  movss (%rax,%rcx), %rdi
  addss (%rax,%rdx), %rdi
  movss %rdi, (%rax,%rsi)
  addq $4, %rax
  cmpq 4*N, %rax
  jne loop
```



Scalar processors

- Let us consider a simple vector addition

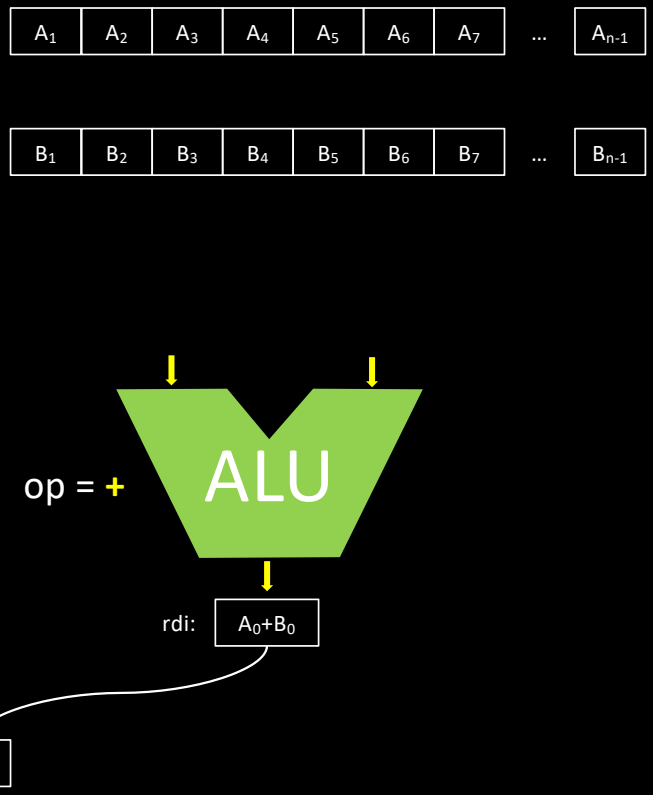
```
static float A[N], B[N], C[N];

xorq %rax, %rax
leaq _A(%rip), %rcx
leaq _B(%rip), %rdx
leaq _C(%rip), %rsi
loop:
  movss (%rax,%rcx), %rdi
  addss (%rax,%rdx), %rdi
  movss %rdi, (%rax,%rsi)
  addq $4, %rax
  cmpq 4*N, %rax
  jne loop
```



Scalar processors

- Let us consider a simple vector addition

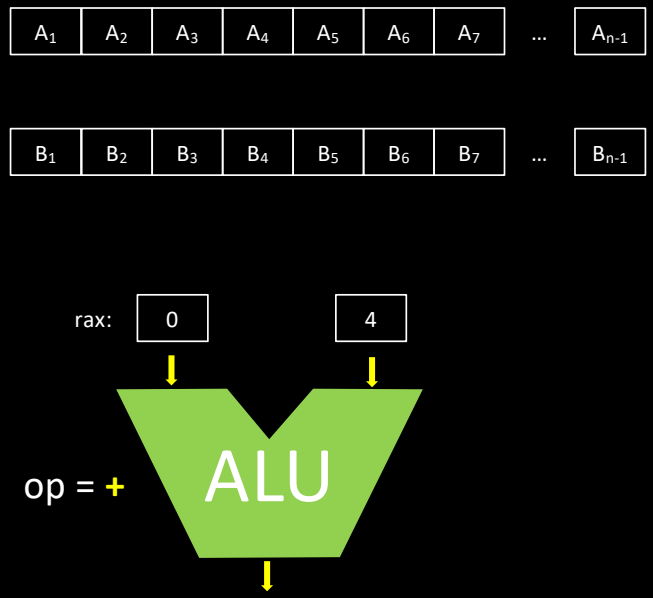
```
static float A[N], B[N], C[N];

xorq %rax, %rax
leaq _A(%rip), %rcx
leaq _B(%rip), %rdx
leaq _C(%rip), %rsi
loop:
  movss (%rax,%rcx), %rdi
  addss (%rax,%rdx), %rdi
  movss %rdi, (%rax,%rsi)
  addq $4, %rax
  cmpq 4*N, %rax
  jne loop
```



Scalar processors

- Let us consider a simple vector addition

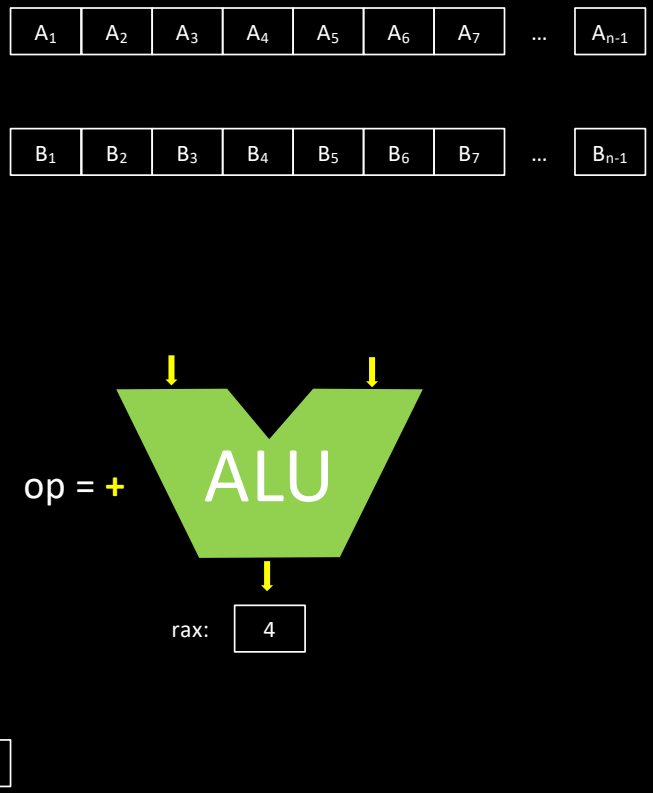
```
static float A[N], B[N], C[N];

xorq %rax, %rax
leaq _A(%rip), %rcx
leaq _B(%rip), %rdx
leaq _C(%rip), %rsi
loop:
  movss (%rax,%rcx), %rdi
  addss (%rax,%rdx), %rdi
  movss %rdi, (%rax,%rsi)
  addq $4, %rax
  cmpq 4*N, %rax
  jne loop
```



Scalar processors

- Let us consider a simple vector addition

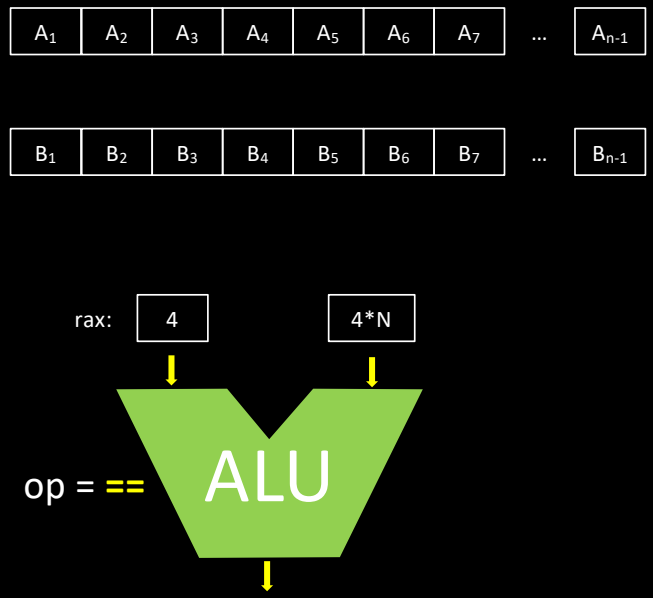
```
static float A[N], B[N], C[N];

xorq %rax, %rax
leaq _A(%rip), %rcx
leaq _B(%rip), %rdx
leaq _C(%rip), %rsi
loop:
  movss (%rax,%rcx), %rdi
  addss (%rax,%rdx), %rdi
  movss %rdi, (%rax,%rsi)
  addq $4, %rax
  cmpq 4*N, %rax
  jne loop
```



Scalar processors

- Let us consider a simple vector addition

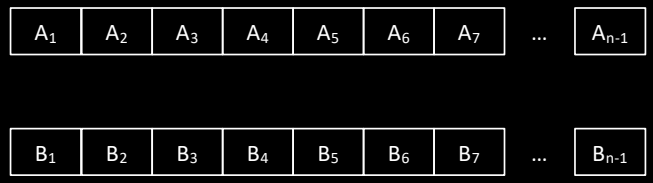
```
static float A[N], B[N], C[N];

xorq %rax, %rax
leaq _A(%rip), %rcx
leaq _B(%rip), %rdx
leaq _C(%rip), %rsi
loop:
  movss (%rax,%rcx), %rdi
  addss (%rax,%rdx), %rdi
  movss %rdi, (%rax,%rsi)
  addq $4, %rax
  cmpq 4*N, %rax
  jne loop
```

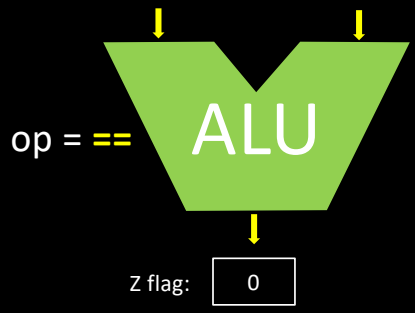


Scalar processors

- Let us consider a simple vector addition

```
static float A[N], B[N], C[N];

xorq %rax, %rax
leaq _A(%rip), %rcx
leaq _B(%rip), %rdx
leaq _C(%rip), %rsi
loop:
  movss (%rax,%rcx), %rdi
  addss (%rax,%rdx), %rdi
  movss %rdi, (%rax,%rsi)
  addq $4, %rax
  cmpq 4*N, %rax
  jne loop
```

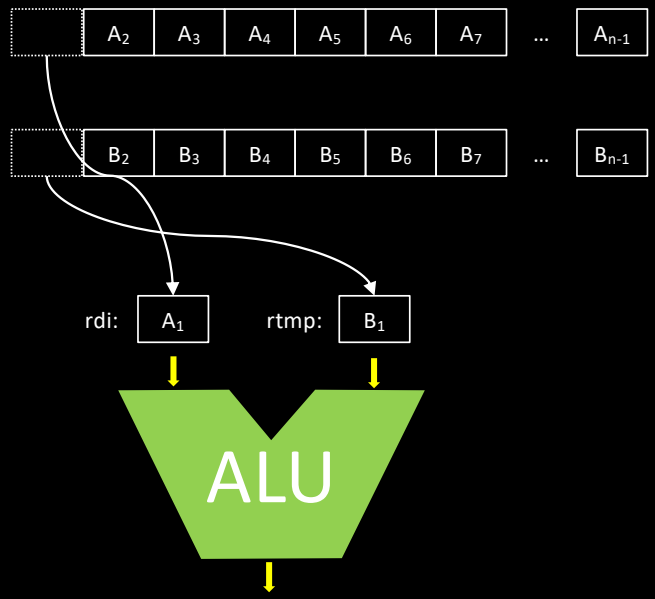




Scalar processors

- Let us consider a simple vector addition

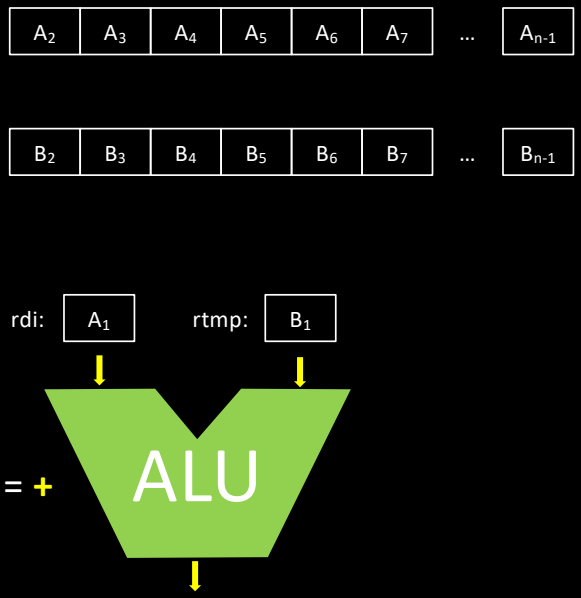
```
static float A[N], B[N], C[N];

xorq %rax, %rax
leaq _A(%rip), %rcx
leaq _B(%rip), %rdx
leaq _C(%rip), %rsi
loop:
  movss (%rax,%rcx), %rdi
  addss (%rax,%rdx), %rdi
  movss %rdi, (%rax,%rsi)
  addq $4, %rax
  cmpq 4*N, %rax
  jne loop
```



Scalar processors

- Let us consider a simple vector addition

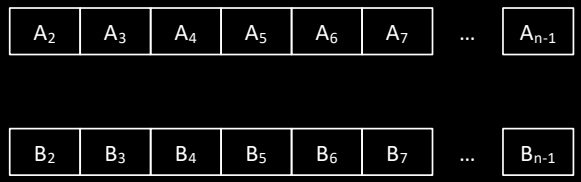
```
static float A[N], B[N], C[N];

xorq %rax, %rax
leaq _A(%rip), %rcx
leaq _B(%rip), %rdx
leaq _C(%rip), %rsi
loop:
  movss (%rax,%rcx), %rdi
  addss (%rax,%rdx), %rdi
  movss %rdi, (%rax,%rsi)
  addq $4, %rax
  cmpq 4*N, %rax
  jne loop
```

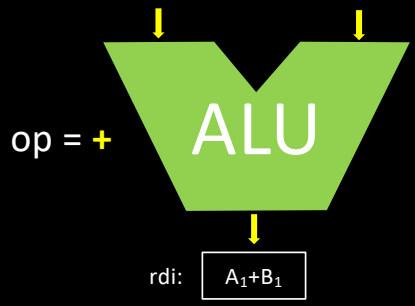


Scalar processors

- Let us consider a simple vector addition

```
static float A[N], B[N], C[N];

xorq %rax, %rax
leaq _A(%rip), %rcx
leaq _B(%rip), %rdx
leaq _C(%rip), %rsi
loop:
  movss (%rax,%rcx), %rdi
  addss (%rax,%rdx), %rdi
  movss %rdi, (%rax,%rsi)
  addq $4, %rax
  cmpq 4*N, %rax
  jne loop
```



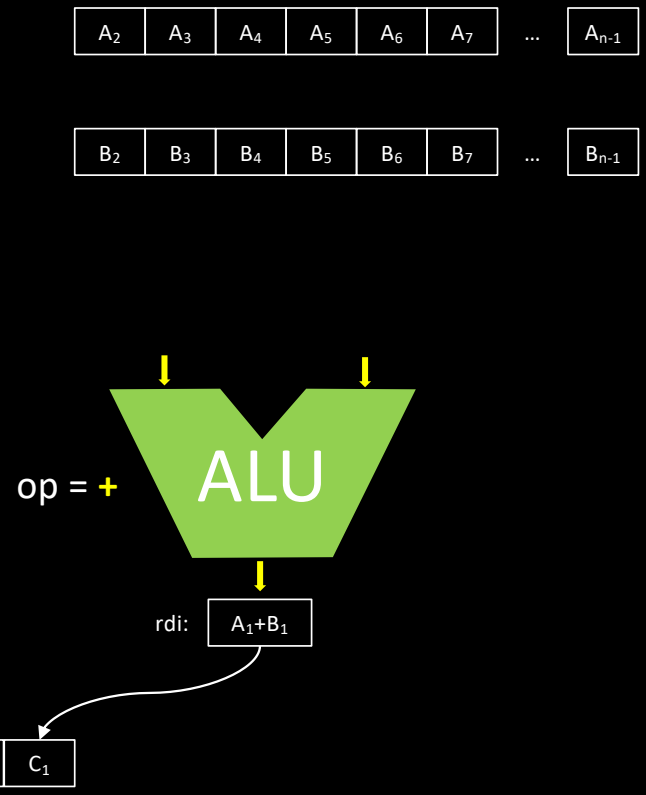


C0

Scalar processors

- Let us consider a simple vector addition

```
static float A[N], B[N], C[N];

xorq %rax, %rax
leaq _A(%rip), %rcx
leaq _B(%rip), %rdx
leaq _C(%rip), %rsi
loop:
  movss (%rax,%rcx), %rdi
  addss (%rax,%rdx), %rdi
  movss %rdi, (%rax,%rsi)
  addq $4, %rax
  cmpq 4*N, %rax
  jne loop
```

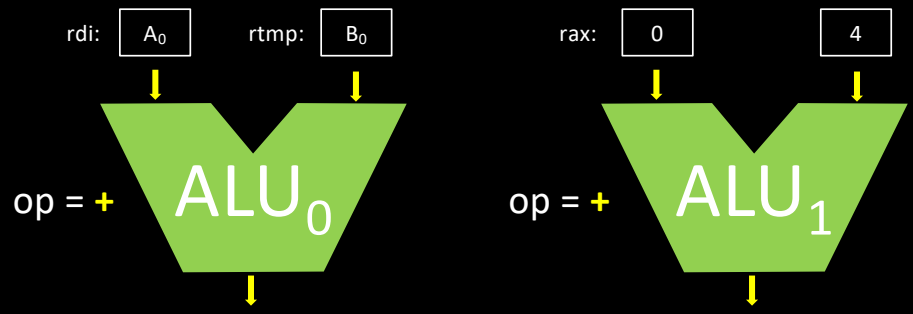


Scalar processors

- Several sources of inefficiency
  - Overhead of loop counter management
    - Lot of extra instructions to fetch, decode, etc.
  - Latency of memory paid for each data element
  - ALU pipeline underutilized

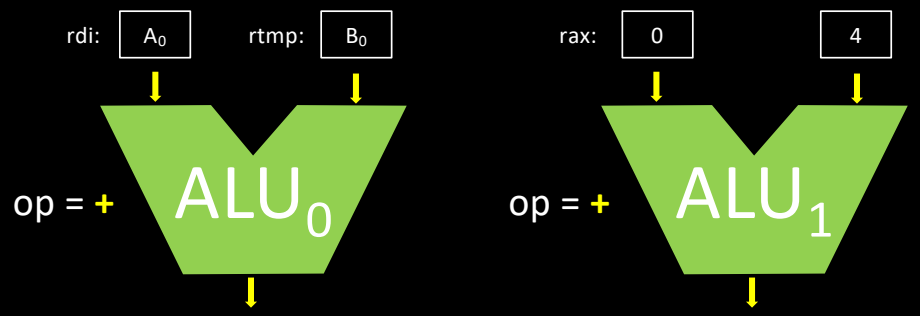
SuperScalar processors

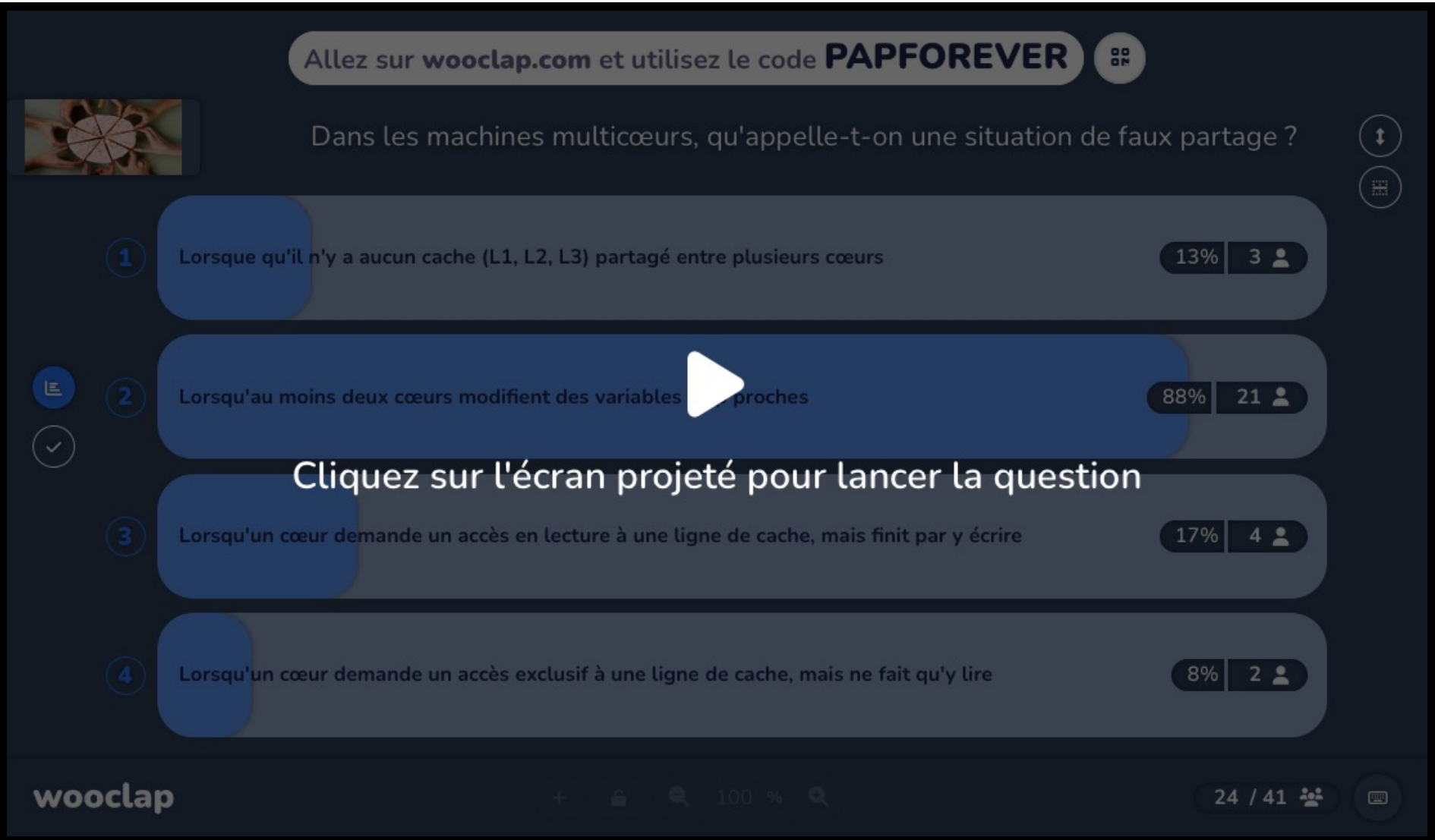
- Multiple ALU pipelines help
  - Independent operations may run in parallel
- Some problems remain
  - Overhead of loop counter management
    - Lot of extra instructions to fetch, decode, etc.
  - Latency of memory paid for each data element
  - Complexity of instruction dispatcher
    - Register renaming, reordering



SuperScalar processors

- Limits
  - For large datasets (and poor locality), caches do not help any more
  - Increasing the *fetch-decode* rate is hard
    - At some point, increasing the number of pipelines does not improve performance any more



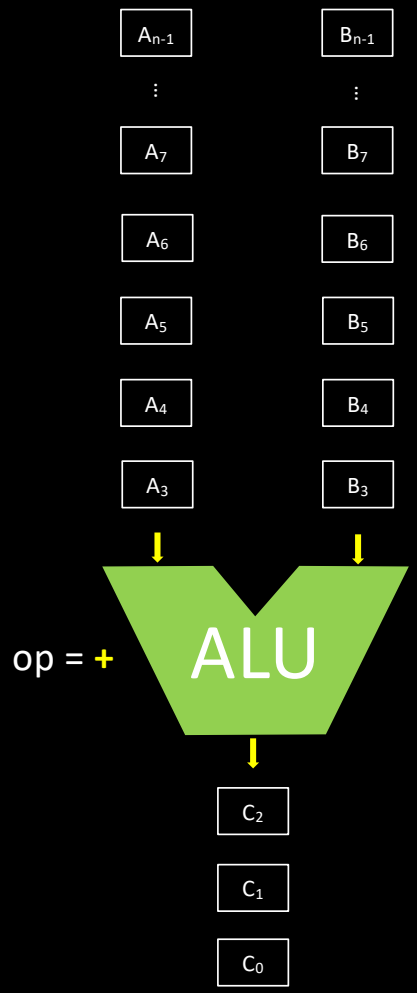

Allez sur wooclap.com et utilisez le code PAPFOREVER
Dans les machines multicœurs, qu'appelle-t-on une situation de faux partage ?
1 Lorsque qu'il n'y a aucun cache (L1, L2, L3) partagé entre plusieurs cœurs 13% 3
2 Lorsqu'au moins deux cœurs modifient des variables proches 88% 21
Cliquez sur l'écran projeté pour lancer la question
3 Lorsqu'un cœur demande un accès en lecture à une ligne de cache, mais finit par y écrire 17% 4
4 Lorsqu'un cœur demande un accès exclusif à une ligne de cache, mais ne fait qu'y lire 8% 2
wooclap
100 %
24 / 41

Data-level Parallelism

- Flynn's Taxonomy [1966]
  - Single instruction stream, single data stream (SISD)
    - Sequential processor
  - Single instruction stream, multiple data streams (SIMD)
    - Vector and SIMD units
  - Multiple instruction streams, single data stream (MISD)
    - Systolic arrays
  - Multiple instruction streams, multiple data streams (MIMD)
    - Multicore architectures

Vector processors

- Idea
  - Stream vectors through pipelined ALUs
    - Single ALU operation applied to multiple data
- Implementation based on
  - Specific vector instructions
    - vload, vstore, vadd, vmul, …
  - High-Bandwidth memory



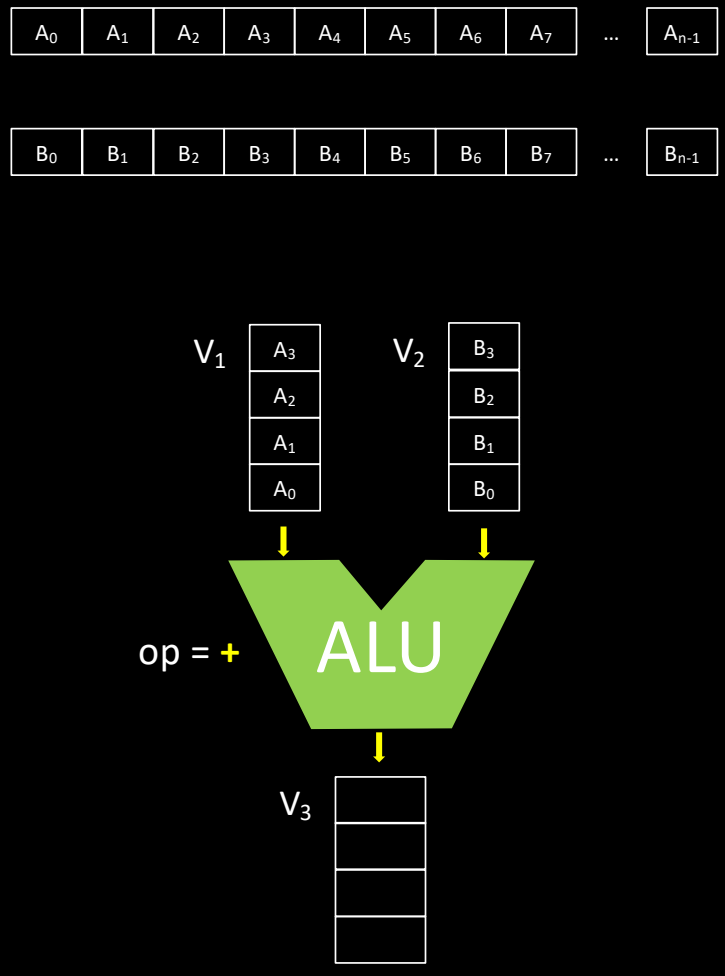
Vector processors

- Some vector processors were working directly with memory operands
  - CDC STAR-100 machine [1966]
  - TI ASC machine [1966]
- These processors could work on vectors of arbitrary length, but
  - Cannot keep values inside the processor for further computations
  - Suffer of high operation startup latency

Vector processors

- Vector register architecture
  - Large vector registers
    - Each register can store multiple scalar elements

```
vload &A[0], v1
vload &B[0], v2
vadd v1, v2, v3
vstore v3, &C[0]
```

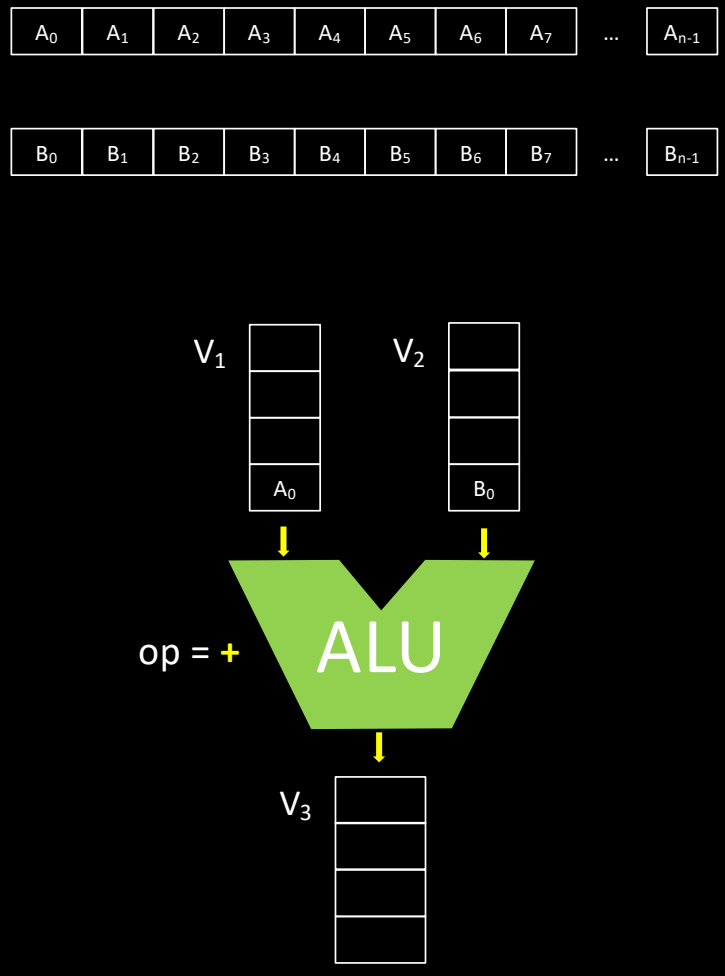


Vector processors

- Vector register architecture
  - Large vector registers
    - Each register can store multiple scalar elements

```
vload &A[0], v1
vload &B[0], v2
vadd v1, v2, v3
vstore v3, &C[0]
```

- To reduce startup penalty, vload/store are also pipelined

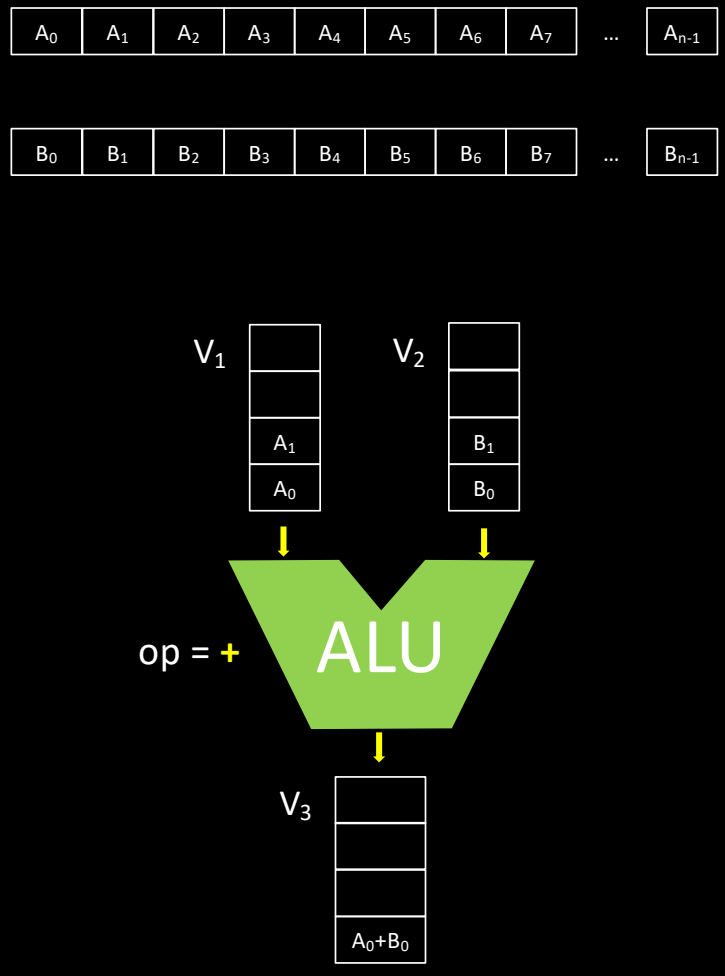


Vector processors

- Vector register architecture
  - Large vector registers
    - Each register can store multiple scalar elements

```
vload &A[0], v1
vload &B[0], v2
vadd v1, v2, v3
vstore v3, &C[0]
```

- To reduce startup penalty, vload/store are also pipelined

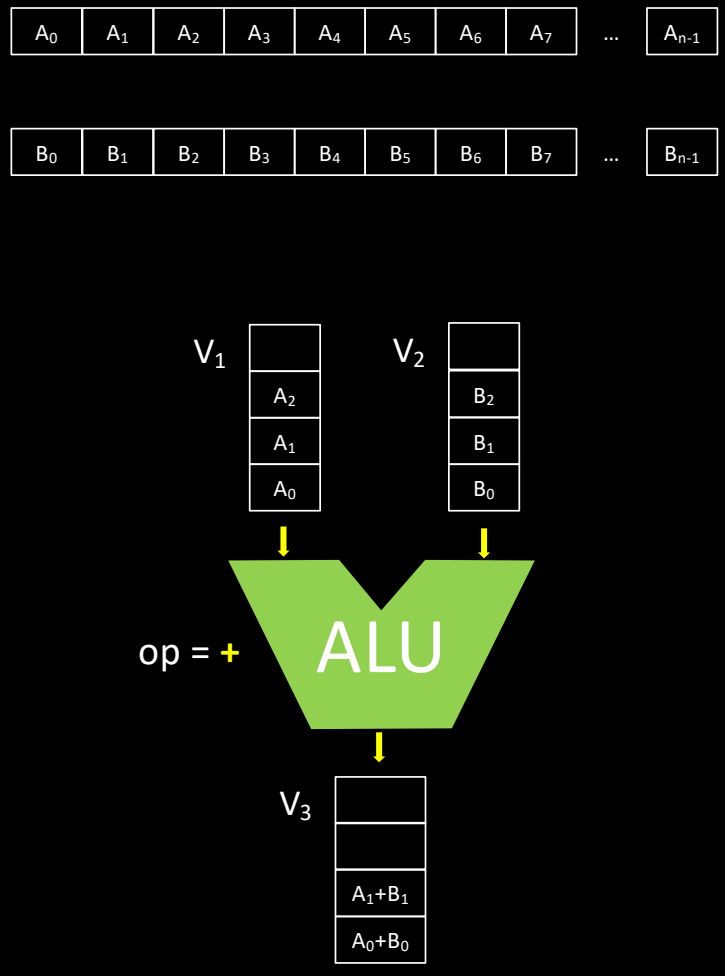


Vector processors

- Vector register architecture
  - Large vector registers
    - Each register can store multiple scalar elements

```
vload &A[0], v1
vload &B[0], v2
vadd v1, v2, v3
vstore v3, &C[0]
```

- To reduce startup penalty, vload/store are also pipelined

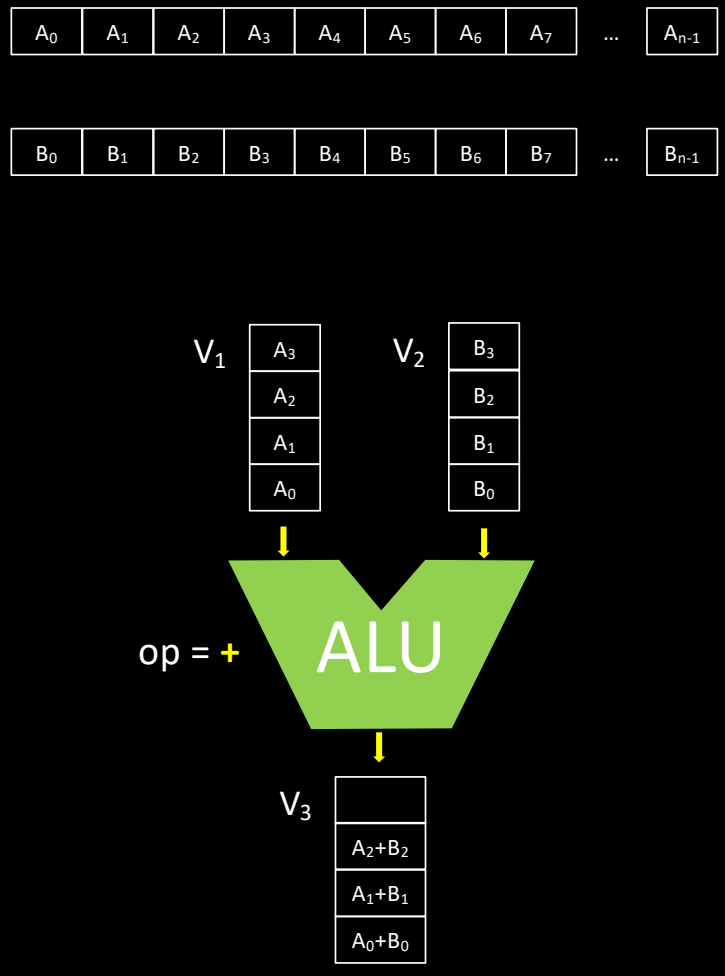


Vector processors

- Vector register architecture
  - Large vector registers
    - Each register can store multiple scalar elements

```
vload &A[0], v1
vload &B[0], v2
vadd v1, v2, v3
vstore v3, &C[0]
```

- To reduce startup penalty, vload/store are also pipelined

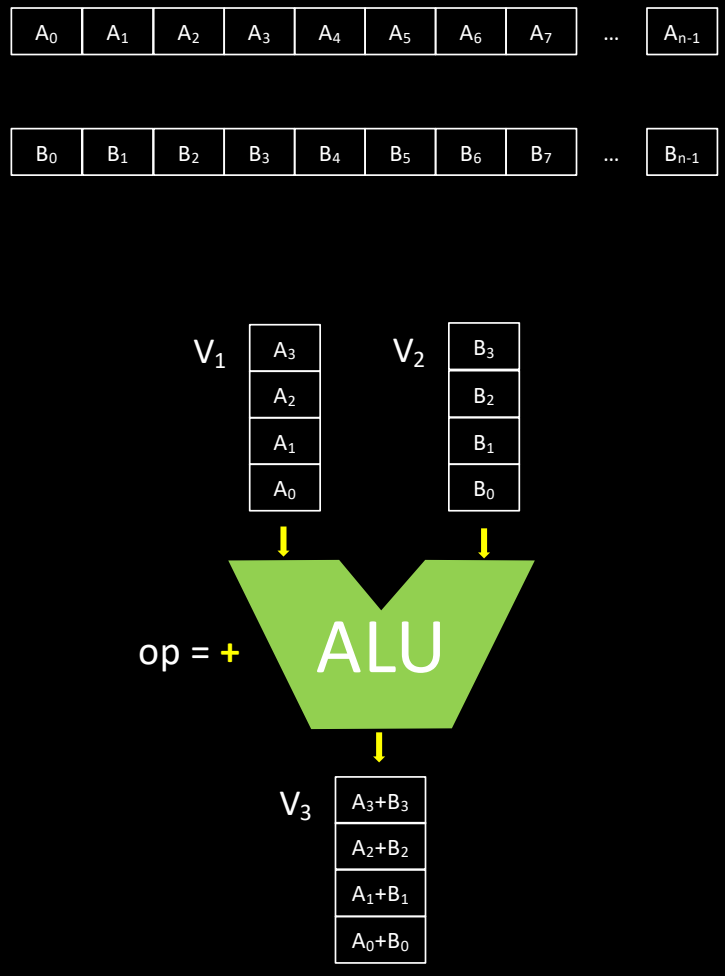


Vector processors

- Vector register architecture
  - Large vector registers
    - Each register can store multiple scalar elements

```
vload &A[0], v1
vload &B[0], v2
vadd v1, v2, v3
vstore v3, &C[0]
```

- To reduce startup penalty, vload/store are also pipelined

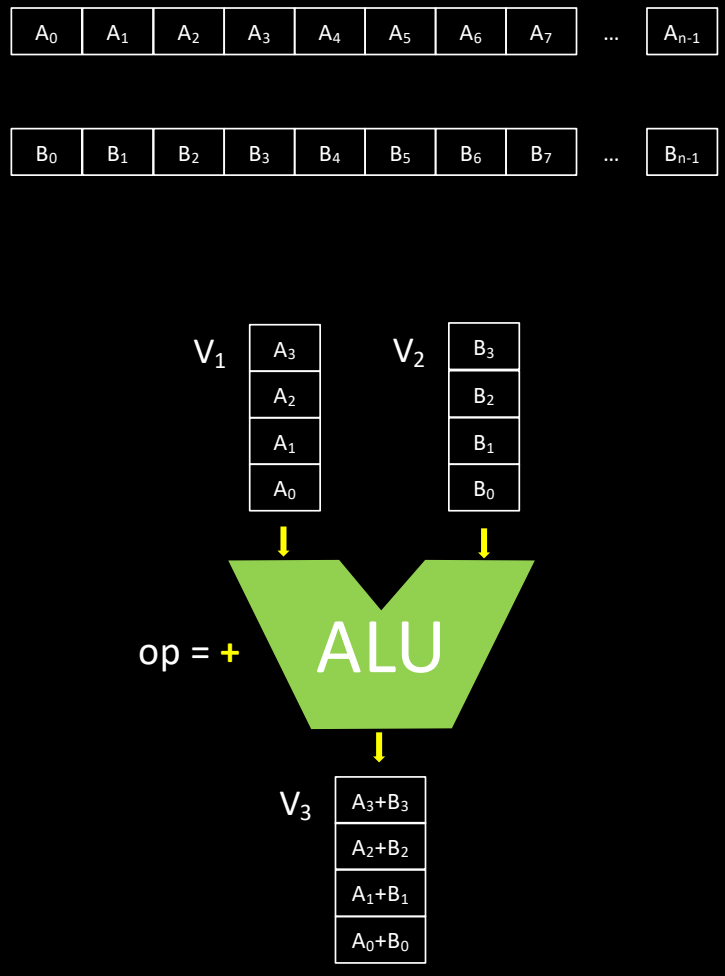


Vector processors

- Coping with large arrays
  - "strip mining"
    - Each register can store multiple scalar elements

```
xor r1, r1
loop:
  vload &A[r1], v1
  vload &B[r1], v2
  vadd v1, v2, v3
  vstore v3, &C[r1]
  add r1, 4, r1
  cmp r1, N
jne loop
```

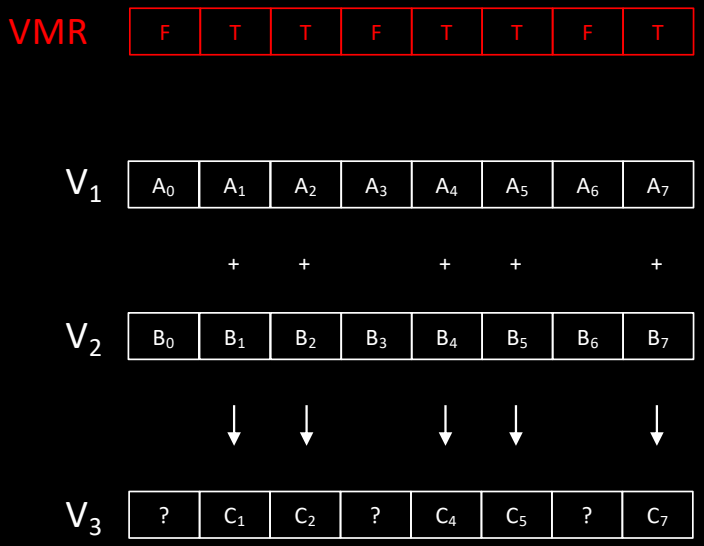
- If (N mod 4) ≠ 0, use the *Vector Length Register* to handle the last chunk



Vector processors

- Coping with divergence
  - A *Vector Mask Register* contains Booleans
    - Can be filled with the result of a vector comparison
    - Vector operations are performed if the corresponding Boolean is TRUE

```
static float A[N], B[N], C[N];

for (int i = 0; i < N; i++)
  if (i % 3 != 0)
    C[i] = A[i] + B[i];
```

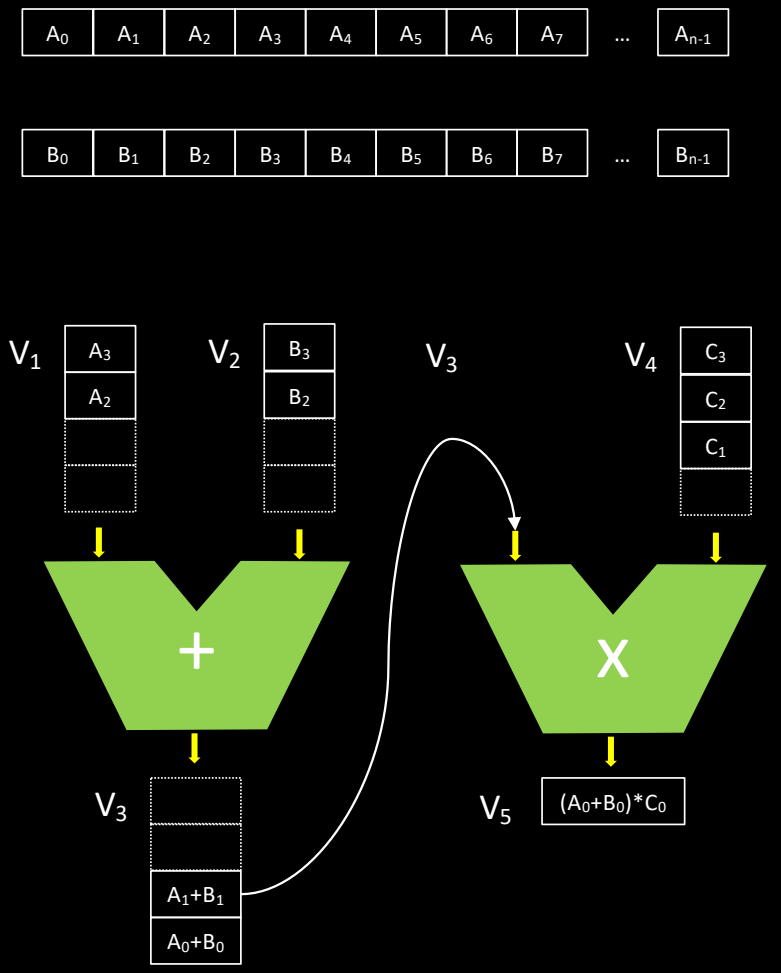


Vector processors

- Vector chaining
  - Use data as soon as available
    - Stream data between different functional units

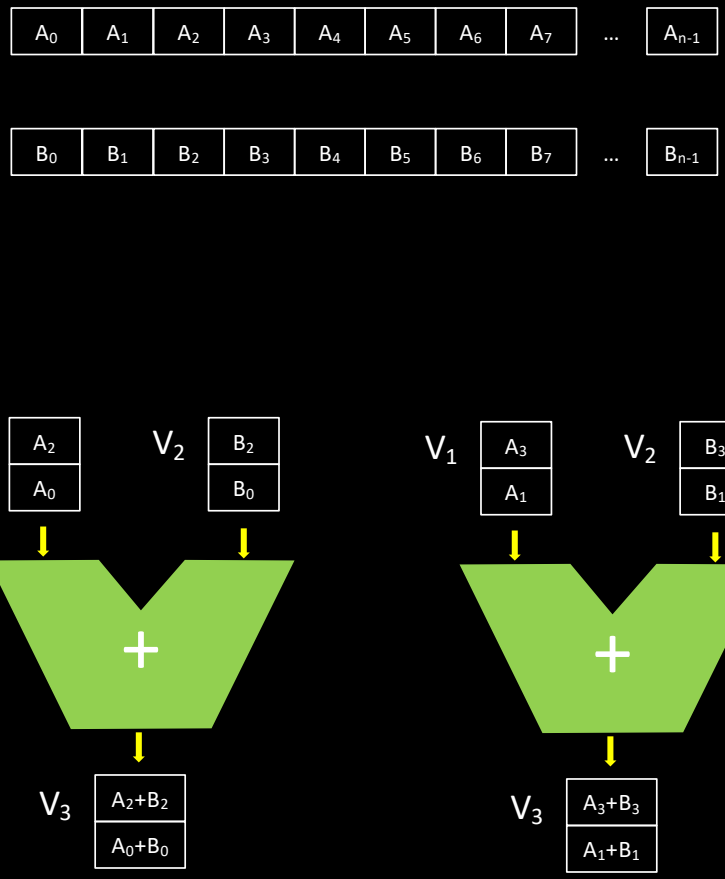
```
vadd v1, v2, v3
vmul v3, v4, v5
```

- Performance mostly depends on memory bandwidth
  - Use of many memory banks



Vector processors

- Multiple lanes processors
  - Vectors are interleaved across the lanes
    - Multiple “add” instructions per cycle

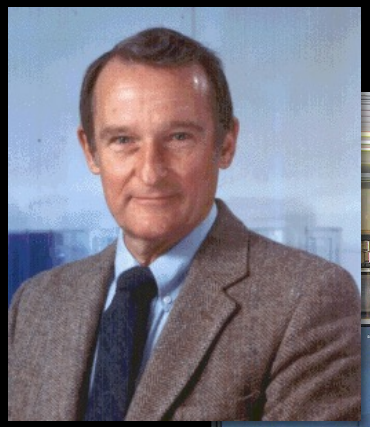


Vector Processors

- How to exploit such architectures efficiently?
  - Use vectorized libraries
    - Developed in assembly language
  - Rely on auto-vectorizing compilers
    - The Fortran language
      - Dimension arrays is known by the compiler

Vector processors

- Cray-1 [1976]
  - Designed by Seymour Cray
  - Single processor
    - 80 MHz, 14-stage pipeline
    - 8 registers of 64 64-bit elements
    - Vector chaining
    - No cache 🙃
  - Cray X-MP (1983, 4 procs, $15M), Cray-2 (1985), Cray Y-MP (1988, 8 procs), Cray C90 (1991, 16 procs)…
- NEC, Fujitsu



Vector processors

- The decline of vector machines happened in the 1990s
  - Vector machines were very expensive
  - Memory performance could hardly follow processor evolution
- *Attack of the killer micros*
  - Using parallel machines composed of off-the-shelf microprocessors provides higher performance
    - Beginning the Cluster Era…

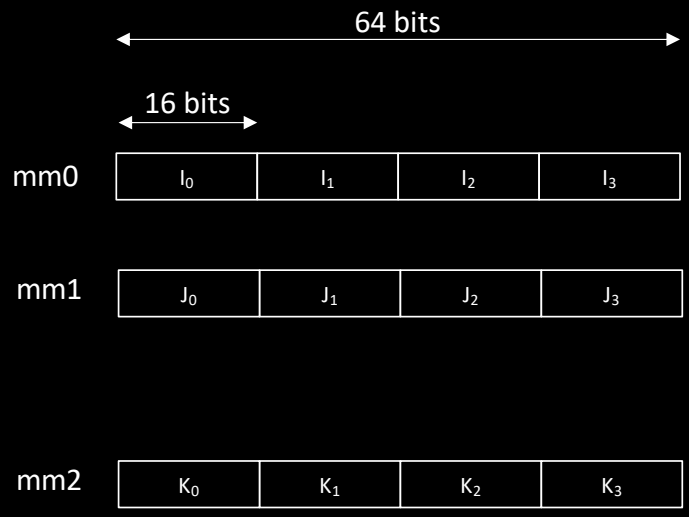

Allez sur wooclap.com et utilisez le code PAPFOREVER
Les années 70 ont vu émerger les révolutionaires supercalculateurs Cray avec cette forme cylindrique reconnaissable entre mille. Mais pourquoi une telle form...
1 Cela permettait de minimiser la longueurs des câbles
2 La machine devient ainsi un canapé sur lequel il fait bon s'asseoir et se réchauffer l'hiver
3 Pour une machine de 5 tonnes, il est ingénieux de pouvoir la déplacer en la faisant rouler sur le sol
4 La colonne centrale abrite le système de refroidissement liquide qui est ainsi en contact avec une large partie de la machine
5 Il y avait souvent un technicien à l'intérieur
Cliquez sur l'écran projeté pour lancer la question
wooclap
100 %
23 / 41

SIMD extensions

- In 1997, Intel releases the Pentium MMX processor
  - MultiMedia eXtension
- New instructions to speed up video & sound decoding
  - Watch DVDs
    even if you don't have a GPU!

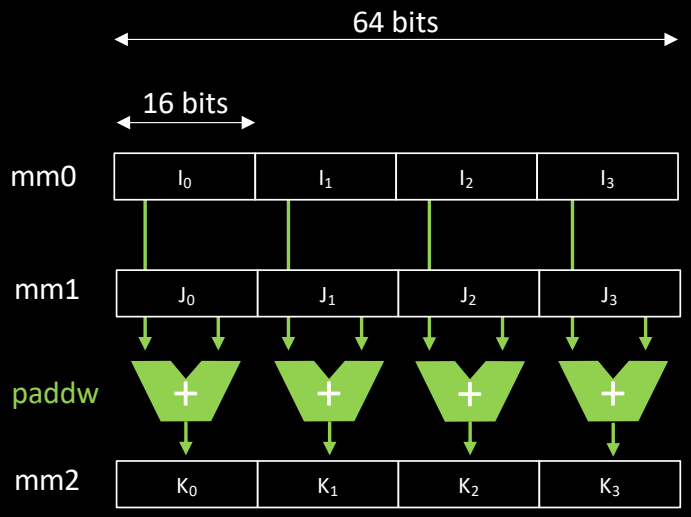
SIMD extensions

- MMX technology
  - 57 new instructions
  - 8 64-bit vector registers
    - mm0, mm1, … mm7
    - Vectors of integers only
      - 2 x 32-bit / 4 x 16-bit / 8 x 8-bit
  - Instruction variants for each size
    - paddb: packed add byte integers
    - paddw: packed add word integers
    - paddd: packed add double integers
    - paddq: packed add quadword integers



SIMD extensions

- MMX technology
  - 57 new instructions
  - 8 64-bit vector registers
    - mm0, mm1, … mm7
    - Vectors of integers only
      - 2 x 32-bit / 4 x 16-bit / 8 x 8-bit
  - Instruction variants for each size
    - paddb: packed add byte integers
    - paddw: packed add word integers
    - paddd: packed add double integers
    - paddq: packed add quadword integers



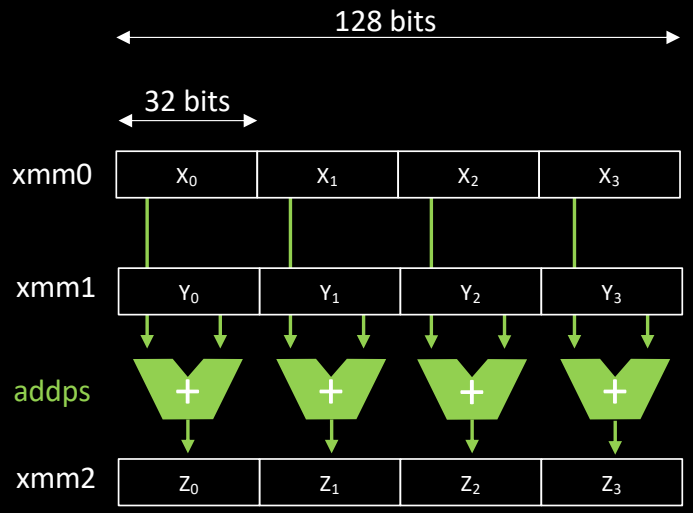
SIMD extensions

- MMX registers reuse the mantissa part of x87 FP registers
  - Using MMX instructions simultaneously with floating point computations is not possible
    - Programs usually switch between MMX-mode and FP-mode…
- AMD 3DNow! [1998]
  - MMX + support for 32-bit floating point data

SIMD extensions

- Intel SSE [1999]
  - Streaming SIMD Extension
  - 70 new instructions
  - 8 128-bit registers (not shared)
    - xmm{0..7}
    - 4 float
  - SSE2 [2000]
    - SEE + 144 instructions!
    - 2 double / 2 long int
    - 4 float / 4 int
    - 8 short int
    - 16 char
  - 16 registers in x86_64 architectures (AMD in 1999, Intel in 2004)
    - xmm{0..15}

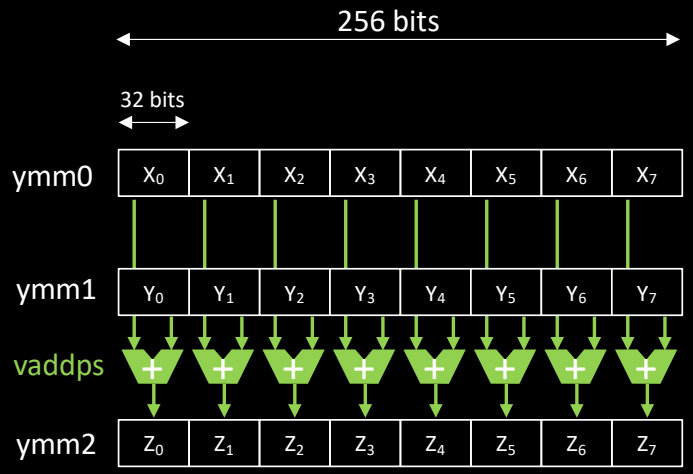
Add packed simple-precision floating point



SIMD extensions

- Wait, there's more!
  - SSE3, SSSE3, SSE4.1, SSE4.2
- Intel AVX [1999]
  - Advanced Vector Extensions
  - +70 new instructions
  - 16 256-bit registers (not shared)
    - ymm{0..15}
    - 4 double / 8 float
  - AVX2 [2013]
    - Most integer operations available
    - Fused Multiply-Add (FMA)
    - Gather load
  - AVX512 [2013]
    - 32 512-bits registers
      - zmm{0..31}
      - Vector size = cache line size!

AVX: add packed simple-precision floating point



SIMD extensions

- Compatibility notes
  - SSE, AVX and AXV-512 instructions can be mixed
    - xmm registers are lower parts of ymm registers, which are lower parts of zmm registers…
  - Each generation of extensions increases the instruction set…
    - Using a uniform instruction set and a Vector Length Register is probably a better approach
      - Intel AVX10 ?

AVX-512 register scheme as extension from the AVX (YMM0-YMM15) and SSE (XMM0-XMM15) registers

511 256	255 128	127 0
ZMM0	YMM0	XMM0
ZMM1	YMM1	XMM1
ZMM2	YMM2	XMM2
ZMM3	YMM3	XMM3
ZMM4	YMM4	XMM4
ZMM5	YMM5	XMM5
ZMM6	YMM6	XMM6
ZMM7	YMM7	XMM7
ZMM8	YMM8	XMM8
ZMM9	YMM9	XMM9
ZMM10	YMM10	XMM10
ZMM11	YMM11	XMM11
ZMM12	YMM12	XMM12
ZMM13	YMM13	XMM13
ZMM14	YMM14	XMM14
ZMM15	YMM15	XMM15
ZMM16	YMM16	XMM16
ZMM17	YMM17	XMM17
ZMM18	YMM18	XMM18
ZMM19	YMM19	XMM19
ZMM20	YMM20	XMM20
ZMM21	YMM21	XMM21
ZMM22	YMM22	XMM22
ZMM23	YMM23	XMM23
ZMM24	YMM24	XMM24
ZMM25	YMM25	XMM25
ZMM26	YMM26	XMM26
ZMM27	YMM27	XMM27
ZMM28	YMM28	XMM28
ZMM29	YMM29	XMM29
ZMM30	YMM30	XMM30
ZMM31	YMM31	XMM31

Source: wikipedia

SIMD extensions

- How to use these instructions in our C programs?
  - How about going back to the essentials?

SIMD extensions

- How to use these instructions in our C programs?
  - How about going back to the essentials?
    - Assembly language 😇

```
LBB4_9:
 vcvtsi2ss %edx, %xmm14, %xmm2
 vbroadcastss %xmm2, %ymm2
 vaddps %ymm4, %ymm2, %ymm2
 vfmadd213ps %ymm1, %ymm0, %ymm2
 vpxor %xmm8, %xmm8, %xmm8
 vxorps %xmm11, %xmm11, %xmm11
 movl $4096, %eax
 vxorps %xmm10, %xmm10, %xmm10
 LBB4_10:
 vmulps %ymm11, %ymm11, %ymm12
 vmovaps %ymm10, %ymm13
 vfmadd213ps %ymm12, %ymm10,
 %ymm13
 vcmpleps %ymm5, %ymm13, %ymm13
 vptest %ymm13, %ymm13
 je LBB4_12
 vpsubd %ymm13, %ymm8, %ymm8
 vmulps %ymm10, %ymm11, %ymm11
 vfnmadd213ps %ymm2, %ymm10,
 %ymm10
 vaddps %ymm10, %ymm12, %ymm10
 vfmadd213ps %ymm7, %ymm6, %ymm11
 vmulps %ymm10, %ymm10, %ymm12
 vmovaps %ymm11, %ymm13
 vfmadd213ps %ymm12, %ymm11,
 %ymm13
 vcmpleps %ymm5, %ymm13, %ymm13
 vptest %ymm13, %ymm13
 je LBB4_12
 vpsubd %ymm13, %ymm8, %ymm8
 vmulps %ymm11, %ymm10, %ymm10
 vfnmadd213ps %ymm2, %ymm11,
 %ymm11
 vaddps %ymm11, %ymm12, %ymm11
 vfmadd213ps %ymm7, %ymm6, %ymm10
 addl $-2, %eax
 jne LBB4_10
```

SIMD extensions

- How to use these instructions in our C programs?
  - How about going back to the essentials?
    - Assembly language 😇
  - Use Intel intrinsics!
    - C style functions that provide access to specific Intel instructions
    - The *Intel Instrinsics Guide* is your friend
      - https://software.intel.com/sites/landingpage/IntrinsicsGuide/

```
LBB4_9:
 vcvtsi2ss %edx, %xmm14, %xmm2
 vbroadcastss %xmm2, %ymm2
 vaddps %ymm4, %ymm2, %ymm2
 vfmadd213ps %ymm1, %ymm0, %ymm2
 vpxor %xmm8, %xmm8, %xmm8
 vxorps %xmm11, %xmm11, %xmm11
 movl $4096, %eax
 vxorps %xmm10, %xmm10, %xmm10
 LBB4_10:
 vmulps %ymm11, %ymm11, %ymm12
 vmovaps %ymm10, %ymm13
 vfmadd213ps %ymm12, %ymm10,
 %ymm13
 vcmpleps %ymm5, %ymm13, %ymm13
 vptest %ymm13, %ymm13
 je LBB4_12
 vpsubd %ymm13, %ymm8, %ymm8
 vmulps %ymm10, %ymm11, %ymm11
 vfnmadd213ps %ymm2, %ymm10,
 %ymm10
 vaddps %ymm10, %ymm12, %ymm10
 vfmadd213ps %ymm7, %ymm6, %ymm11
 vmulps %ymm10, %ymm10, %ymm12
 vmovaps %ymm11, %ymm13
 vfmadd213ps %ymm12, %ymm11,
 %ymm13
 vcmpleps %ymm5, %ymm13, %ymm13
 vptest %ymm13, %ymm13
 je LBB4_12
 vpsubd %ymm13, %ymm8, %ymm8
 vmulps %ymm11, %ymm10, %ymm10
 vfnmadd213ps %ymm2, %ymm11,
 %ymm11
 vaddps %ymm11, %ymm12, %ymm11
 vfmadd213ps %ymm7, %ymm6, %ymm10
 addl $-2, %eax
 jne LBB4_10
```

Intel Intrinsics
https://software.intel.com/sites/landingpage/IntrinsicsGuide/

- Use special types to manipulate vectors
  - E.g., with AVX2
    - `__m256i` holds a vector of integers
    - `__m256` holds a vector of floats
    - `__m256d` holds a vector of double
- Each vector instruction has a corresponding intrinsic function

```
__m256i _mm256_load_si256 (const __m256i *__p);
__m256i _mm256_add_epi32 (__m256i __a, __m256i __b);
```

Intel Intrinsics

https://software.intel.com/sites/landingpage/IntrinsicsGuide/

- Use special types to manipulate vectors
  - E.g., with AVX2
    - `__m256i` holds a vector of integers
    - `__m256` holds a vector of floats
    - `__m256d` holds a vector of double
- Each vector instruction has a corresponding intrinsic function

```
__m256i _mm256_load_si256 (const __m256i *__p);
__m256i _mm256_add_epi32 (__m256i __a, __m256i __b);
```

```
#include <immintrin.h>

float A [8] = { … };
float B [8] = { … };
float C [8];

  __m256 a, b, c;

  a = _mm256_load_ps (A);
  b = _mm256_load_ps (B);

  c = _mm256_add_ps (a, b); // vaddps

  _mm256_store_ps (C, c);
```

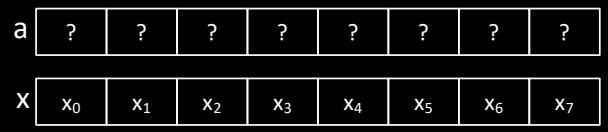
Intel Intrinsics

- Our first *saxpy* kernel
  - "Single precision A X plus Y"
    - X and Y are vectors
    - A is a scalar value

```
float A = …;
float X [8] = { … };
float Y [8] = { … };
float Z [8];

  __m256 x, y, z;

  x = _mm256_load_ps (X);
  y = _mm256_load_ps (Y);

  z = ??? // A * X
```

Intel Intrinsics

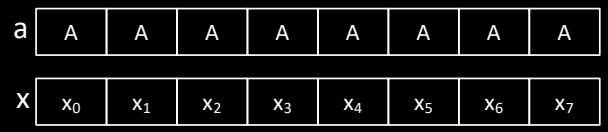
- Our first *saxpy* kernel
  - "Single precision A X plus Y"
    - X and Y are vectors
    - A is a scalar value
- The `_mm256_mul_ps` instruction takes two vectors as inputs




```
float A = …;
float X [8] = { … };
float Y [8] = { … };
float Z [8];

  __m256 a, x, y, z;

  x = _mm256_load_ps (X);
  y = _mm256_load_ps (Y);
  a = ???

  z = _mm256_mul_ps (a, x); // A * X
```

Intel Intrinsics

- Our first *saxpy* kernel
  - "Single precision A X plus Y"
    - X and Y are vectors
    - A is a scalar value
- We need to form a vector by *broadcasting* a scalar value to each element
  - `_mm256_set1_ps`




```
float A = …;
float X [8] = { … };
float Y [8] = { … };
float Z [8];

  __m256 a, x, y, z;

  x = _mm256_load_ps (X);
  y = _mm256_load_ps (Y);
  a = _mm256_set1_ps (A);

  z = _mm256_mul_ps (a, x); // A * X
```

Intel Intrinsics

- Our first *saxpy* kernel
  - "Single precision A X plus Y"
    - X and Y are vectors
    - A is a scalar value
- We could probably gain a few cycles by fusing `mul` and `add`

```
float A = …;
float X [8] = { … };
float Y [8] = { … };
float Z [8];

  __m256 a, x, y, z;

  x = _mm256_load_ps (X);
  y = _mm256_load_ps (Y);
  a = _mm256_set1_ps (A);

  z = _mm256_mul_ps (a, x);
  z = _mm256_add_ps (z, y);

  _mm256_store_ps (Z, z);
```

Intel Intrinsics

- Our first *saxpy* kernel
  - "Single precision A X plus Y"
    - X and Y are vectors
    - A is a scalar value
- We could probably gain a few cycles by fusing `mul` and `add`
  - That is: `fmadd`

```
float A = …;
float X [8] = { … };
float Y [8] = { … };
float Z [8];

  __m256 a, x, y, z;

  x = _mm256_load_ps (X);
  y = _mm256_load_ps (Y);
  a = _mm256_set1_ps (A);

  z = _mm256_fmadd_ps (a, x, y);

  _mm256_store_ps (Z, z);
```

Intel Intrinsics

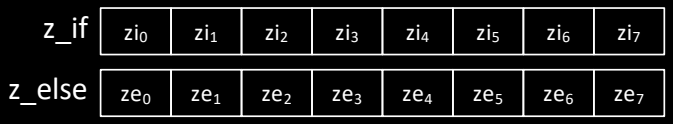
- Divergence inside vectors
  - Use masks to combine results of "*if*" and "*else*" branches

```
for (int i = 0; i < N; i++)
  if (X[i] < 50)
    Z[i] = A * X[i] + Y[i];
  else
    Z[i] = A * X[i] – Y[i];
```

Intel Intrinsics

- Divergence inside vectors
  - Use masks to combine results of "*if*" and "*else*" branches

```
for (int i = 0; i < N; i++)
  if (X[i] < 50)
    Z[i] = A * X[i] + Y[i];
  else
    Z[i] = A * X[i] - Y[i];
```




```
__m256 a, x, y, z_if, z_else, z;
```

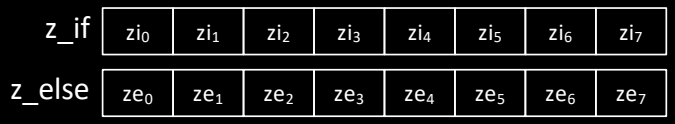
```
…
z_if = _mm256_fmadd_ps (a, x, y);
z_else = _mm256_fmsub_ps (a, x, y);
```

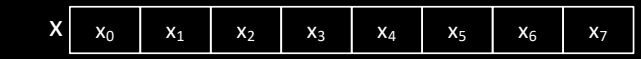
Intel Intrinsics

- Divergence inside vectors
  - Use masks to combine results of "*if*" and "*else*" branches

```
for (int i = 0; i < N; i++)
  if (X[i] < 50)
    Z[i] = A * X[i] + Y[i];
  else
    Z[i] = A * X[i] - Y[i];
```

```
__m256 a, x, y, z_if, z_else, z;
const __m256 fifty = _mm256_set1_ps (50.0);

  …
  z_if = _mm256_fmadd_ps (a, x, y);
  z_else = _mm256_fmsub_ps (a, x, y);
```



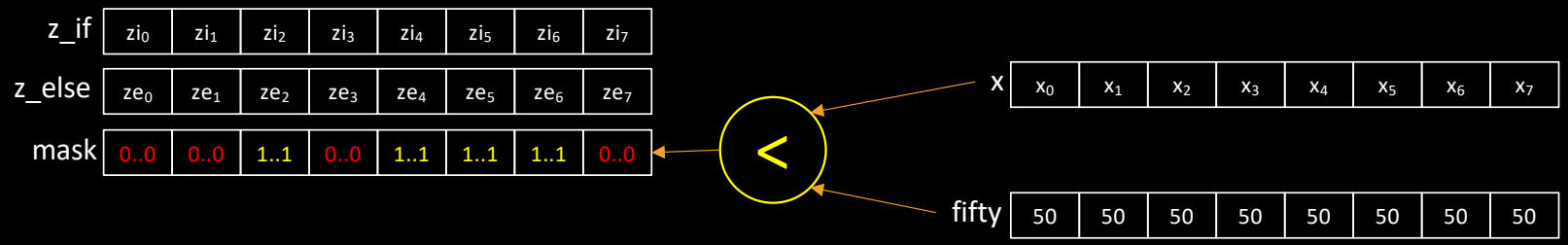


Intel Intrinsics

- Divergence inside vectors
  - Use masks to combine results of "*if*" and "*else*" branches

```
for (int i = 0; i < N; i++)
  if (X[i] < 50)
    Z[i] = A * X[i] + Y[i];
  else
    Z[i] = A * X[i] – Y[i];
```

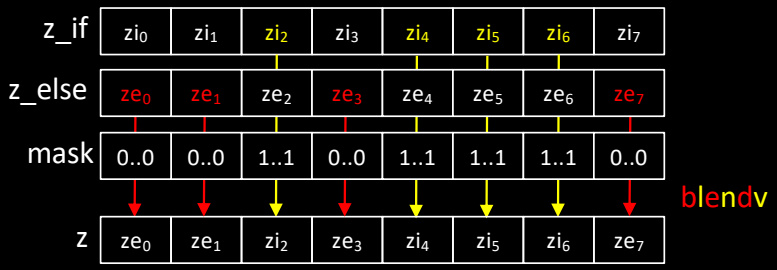
```
__m256 a, x, y, z_if, z_else, z;
const __m256 fifty = _mm256_set1_ps (50.0);

…
z_if = _mm256_fmadd_ps (a, x, y);
z_else = _mm256_fmsub_ps (a, x, y);

__m256 mask = _mm256_cmp_ps (x, fifty, _CMP_LT_OS);
```



Intel Intrinsics

- Divergence inside vectors
  - Use masks to combine results of "*if*" and "*else*" branches

```
for (int i = 0; i < N; i++)
  if (X[i] < 50)
    Z[i] = A * X[i] + Y[i];
  else
    Z[i] = A * X[i] – Y[i];
```




```
__m256 a, x, y, z_if, z_else, z;
const __m256 fifty = _mm256_set1_ps (50.0);

…
z_if = _mm256_fmadd_ps (a, x, y);
z_else = _mm256_fmsub_ps (a, x, y);

__m256 mask = _mm256_cmp_ps (x, fifty, _CMP_LT_OS);

z = _mm256_blendv_ps (z_else, z_if, mask);
```

Intel Intrinsics

- Divergence inside vectors
  - Use masks to combine results of "*if*" and "*else*" branches

```
for (int i = 0; i < N; i++)
  if (X[i] != 0)
    Y[i] = 1.0 / X[i];
  else
    Y[i] = A;
```

  - What happens when dividing by zero for some vector components?
    😱

SIMD extensions

- Are we forced to use assembly/intrinsics to get performance? 😱
  - See `saxpy.c`
  - See `invert.c` (EasyPAP)
- When the loop are *simple enough*, the compiler is able to vectorize
  - gcc can report useful information

```
> gcc -o perf.o -O3 -march=native -fopt-info-vec-optimized -c perf.c
perf.c:26:3: optimized: loop vectorized using 32 byte vectors
perf.c:41:3: optimized: loop vectorized using 32 byte vectors
```

- But several things may prevent auto-vectorization
  - Too much divergence (`mandel.c` 🤔)
  - Complex memory layout
  - Intra-loop dependencies

Energy consumption and Frequency Scaling

- Heavy use of SIMD units may lead to frequency decrease
  - Example: Intel processors
    - Level 0: normal frequency
    - Level 1: 97% to 85% of L0
    - Level 2: 91% to 62% of L0

- L0
  - scalar
  - 128-bits wide simd
  - “light” 256-bit wide simd
- L1
  - “heavy” 256-bit wide simd
  - “light” 512-bit wide simd
- L2
  - “heavy” 512-bit wide simd

Mode	Base	Turbo Frequency/Active Cores (Xeon Gold 5120)													
		1	2	3	4	5	6	7	8	9	10	11	12	13	14
Normal	2,200 MHz	3,200 MHz	3,200 MHz	3,000 MHz	3,000 MHz	2,900 MHz	2,900 MHz	2,900 MHz	2,900 MHz	2,700 MHz	2,700 MHz	2,700 MHz	2,700 MHz	2,600 MHz	2,600 MHz
AVX2	1,800 MHz	3,100 MHz	3,100 MHz	2,900 MHz	2,900 MHz	2,700 MHz	2,700 MHz	2,700 MHz	2,700 MHz	2,300 MHz	2,300 MHz	2,300 MHz	2,300 MHz	2,200 MHz	2,200 MHz
AVX512	1,200 MHz	2,900 MHz	2,900 MHz	2,500 MHz	2,500 MHz	1,900 MHz	1,900 MHz	1,900 MHz	1,900 MHz	1,600 MHz	1,600 MHz	1,600 MHz	1,600 MHz	1,600 MHz	1,600 MHz

SIMD extensions

- SIMD is a strong trend in processor architecture
  - Data layout has deep implications on performance
- SIMD shares a lot with GPU programming, as we will see
  - Think data-parallel! 😉

SIMD Quizz


Vous ne pouvez plus voter
for (int i = 4; i < N; i++)
tab[i] = f (tab[i - 4]);
Le tableau contient des float. Cette boucle séquentielle peut-elle être vectorisé...
1 Toutes les boucles sont vectorisables si on y réfléchit bien 7% 2
2 Hélas non: elle fait partie de la famille des boucles qui aiment prendre leur temps 29% 8
sable avec des registres 128 bits 39% 11
Réponse correcte
4 Elle est vectorisable avec des registres 256 bits 14% 4
5 Elle est vectorisable, mais avec des registres qui n'existent pas encore 11% 3
wooclap
100 %
39% correct
28 / 41

Additional resources
available on
http://gforgeron.gitlab.io/pap/