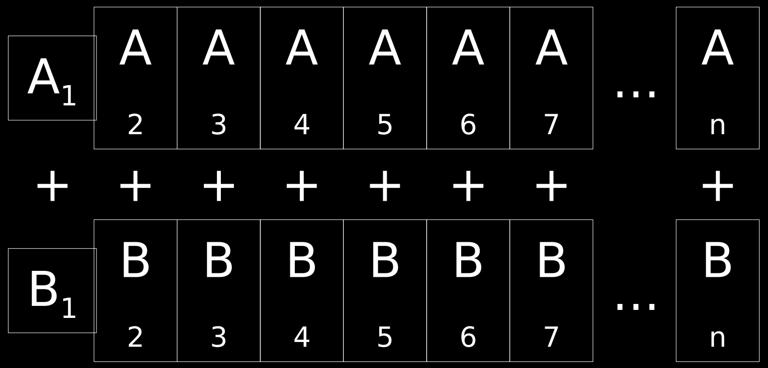
Parallel Computing: Vector and SIMD Programming

Raymond Namyst, Pierre-André Wacrenier
Dept. of Computer Science
University of Bordeaux, France

Data-level Parallelism

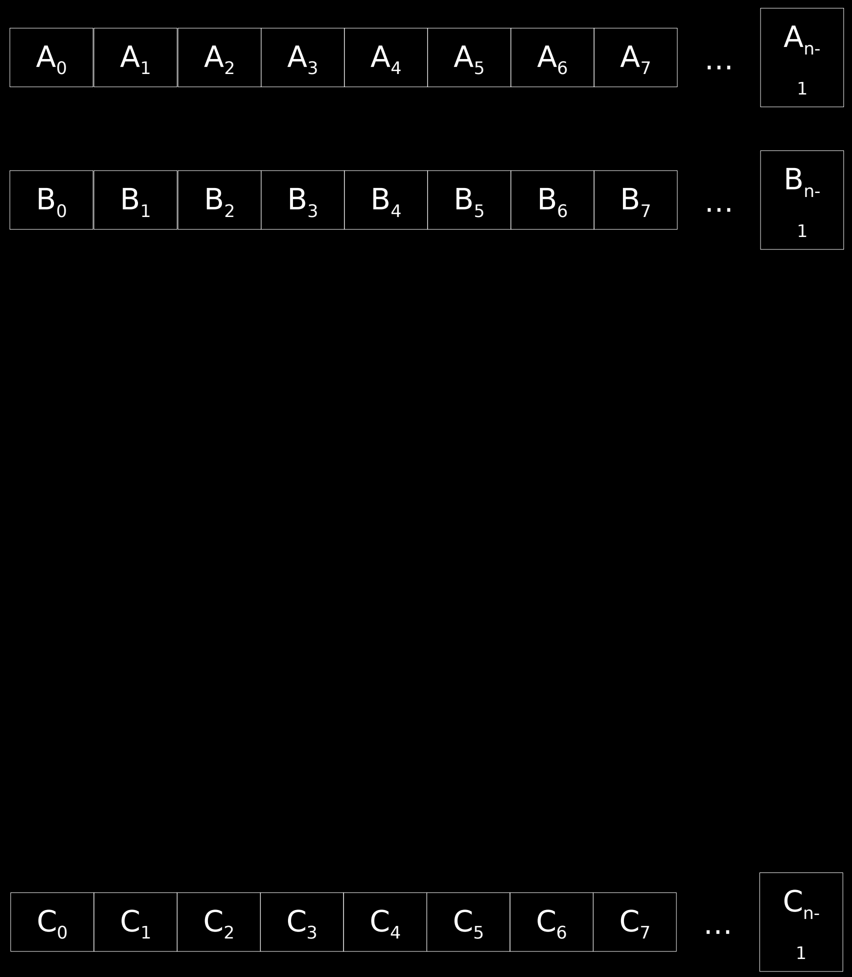
- In many applications, parallelism comes from applying the same operations simultaneously across large sets of data
  - One single thread of control, parallel data operations
    - Matrix- and Vector-oriented programs
    - Image and Sound processing



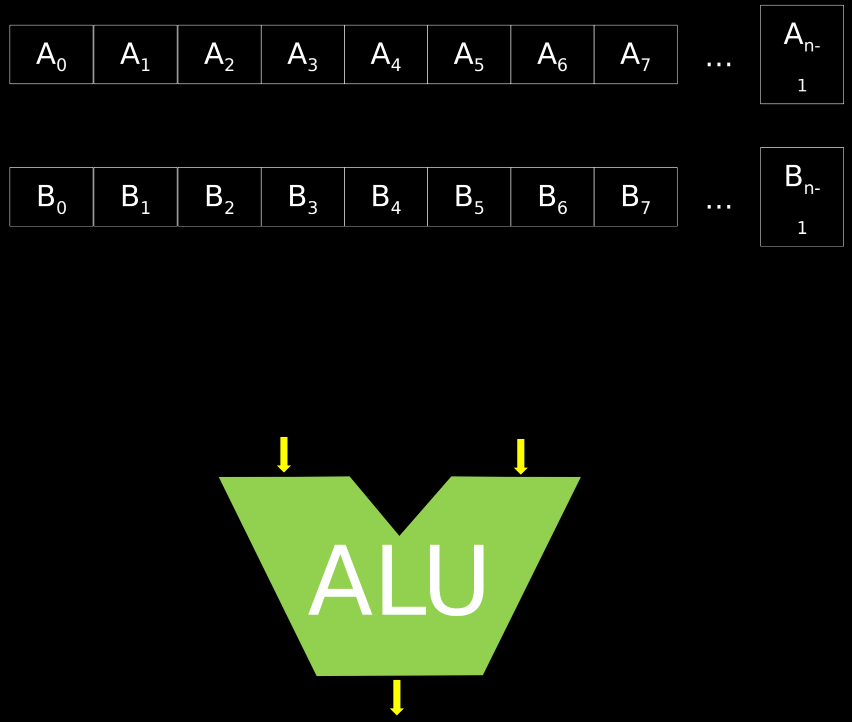
Scalar processors

- 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



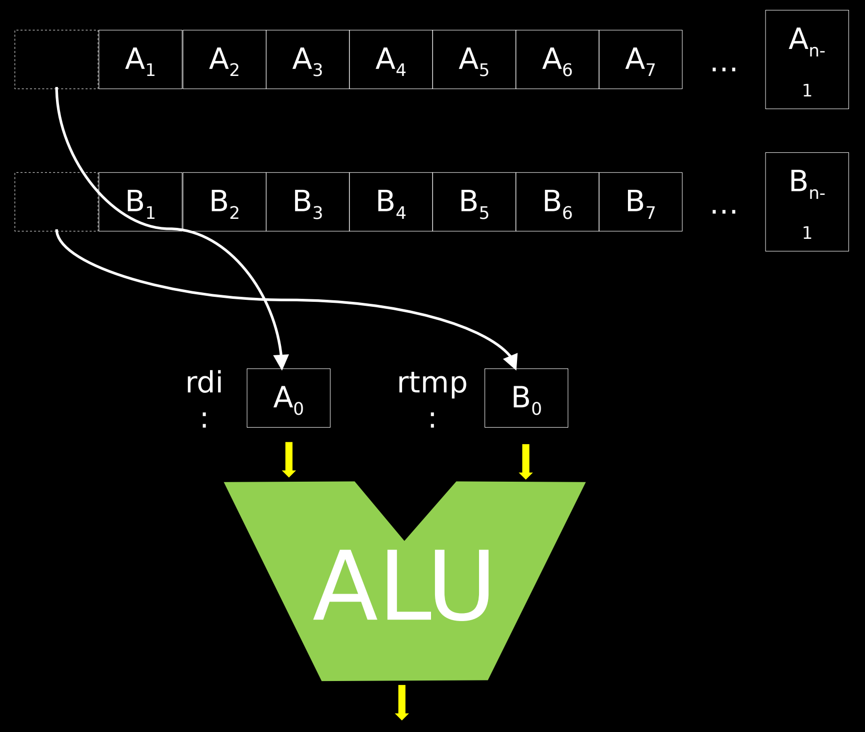
- Let us consider a simple vector addition

```
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

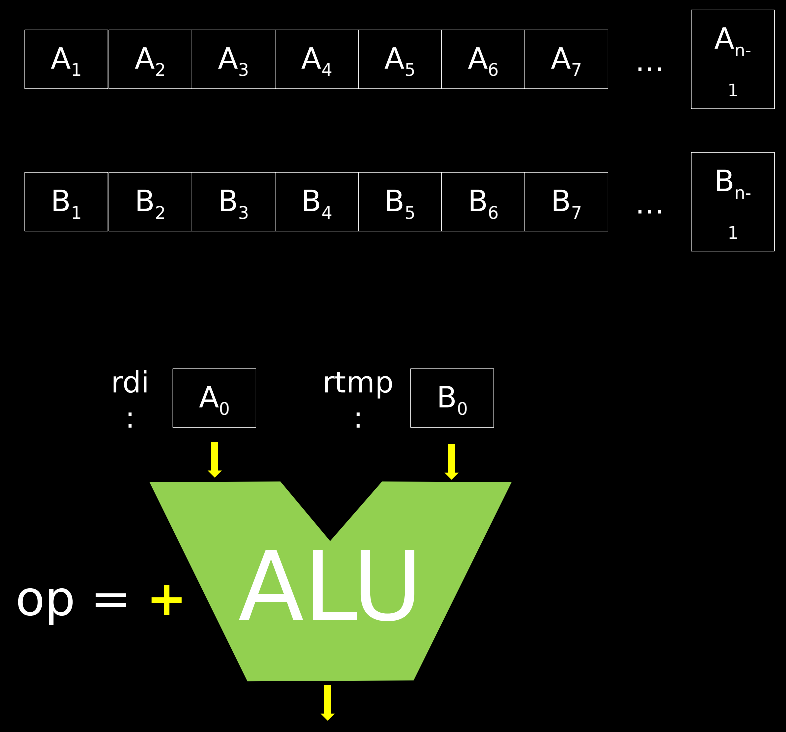
```
static float A[N], B[N], C[N];

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

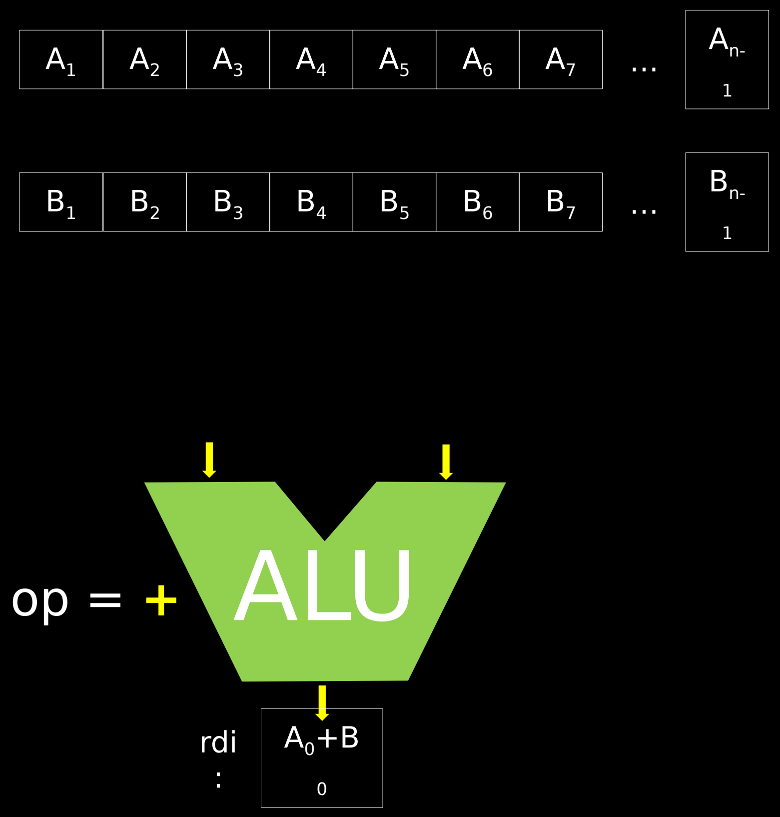
```
static float A[N], B[N], C[N];

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

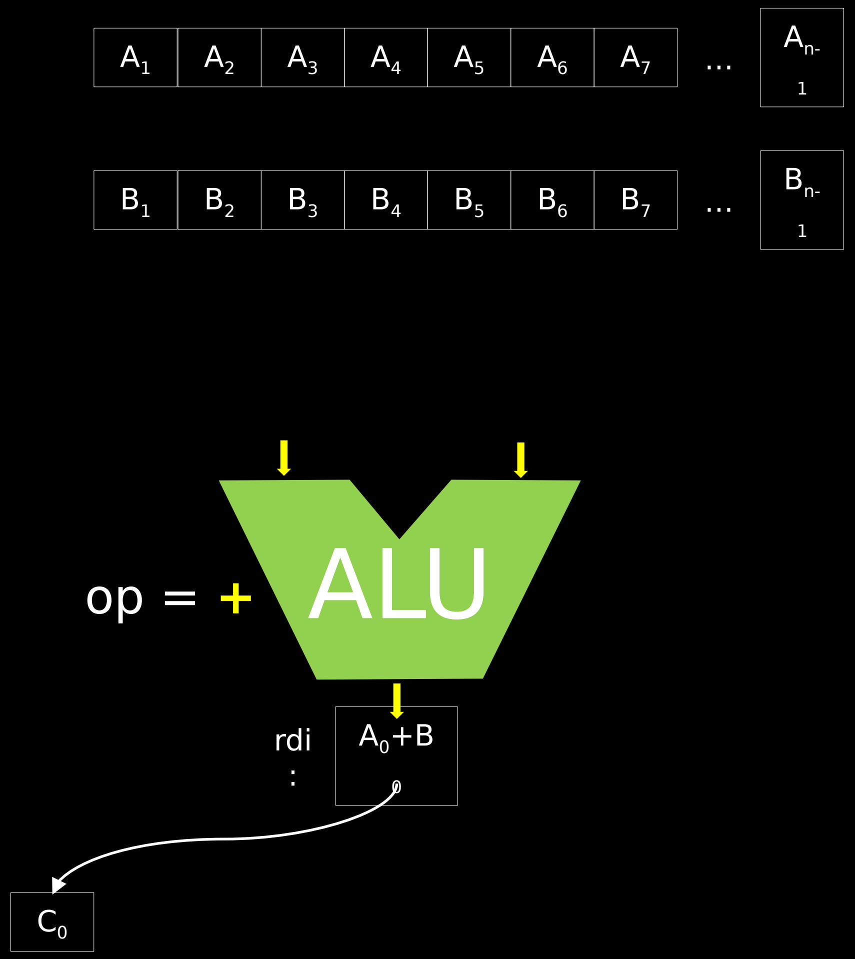
```
static float A[N], B[N], C[N];

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

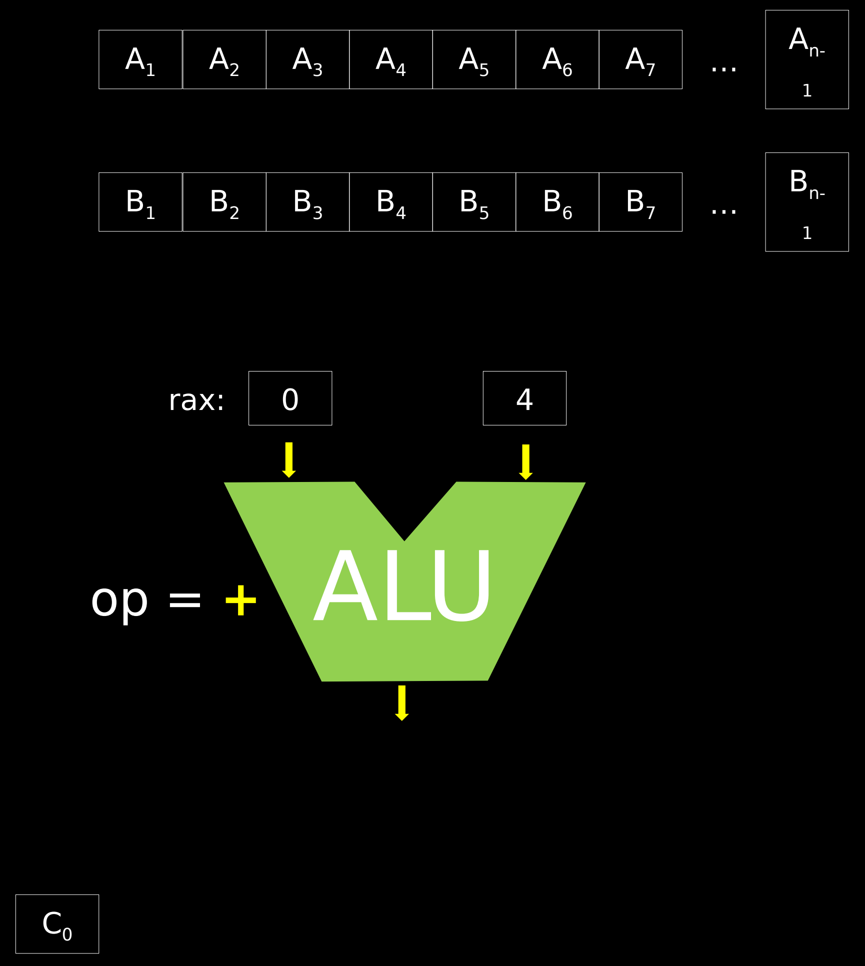
```
static float A[N], B[N], C[N];

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

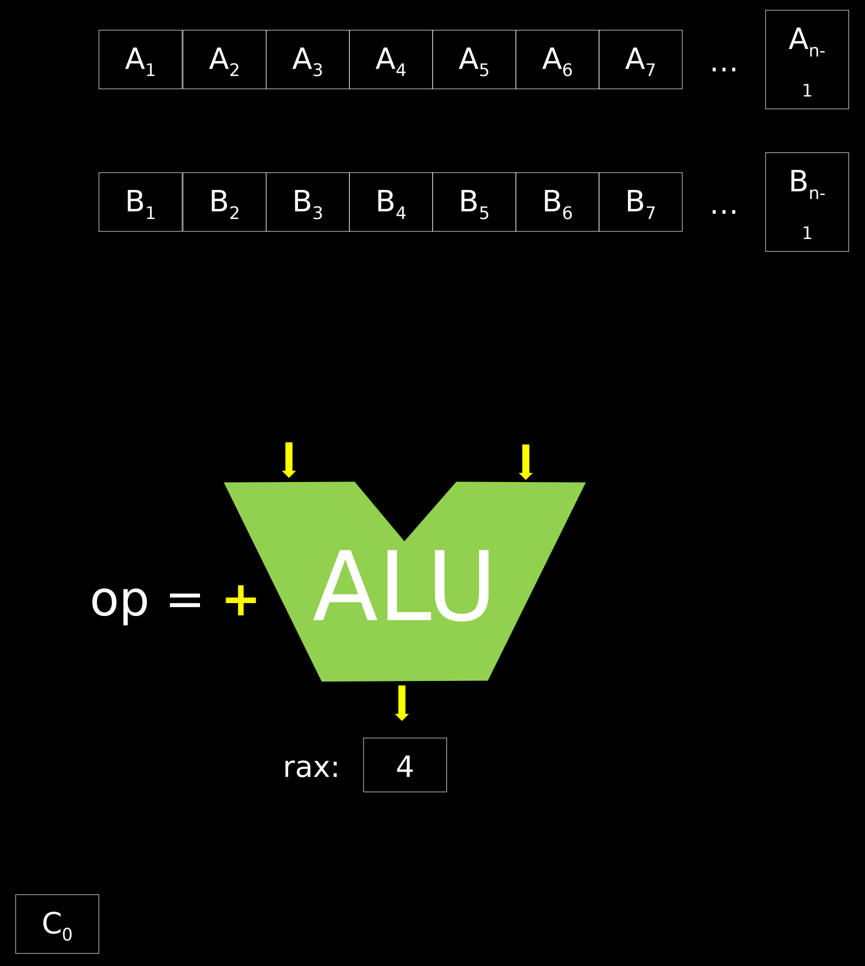
```
static float A[N], B[N], C[N];

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

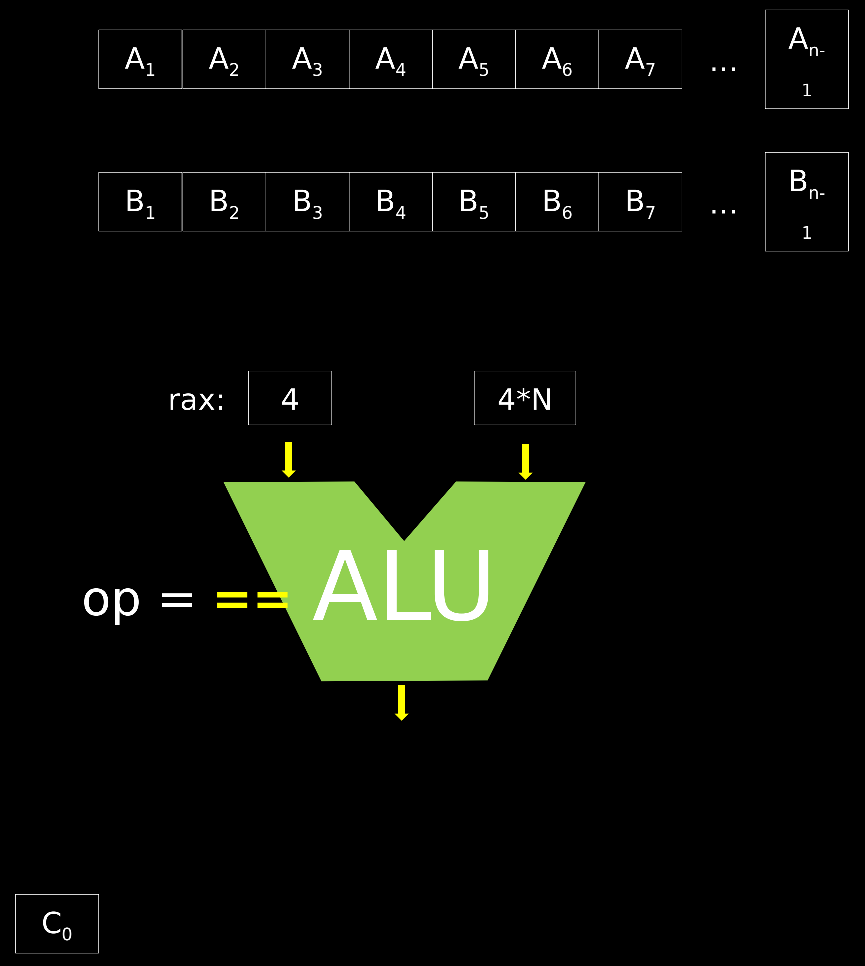
```
static float A[N], B[N], C[N];

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

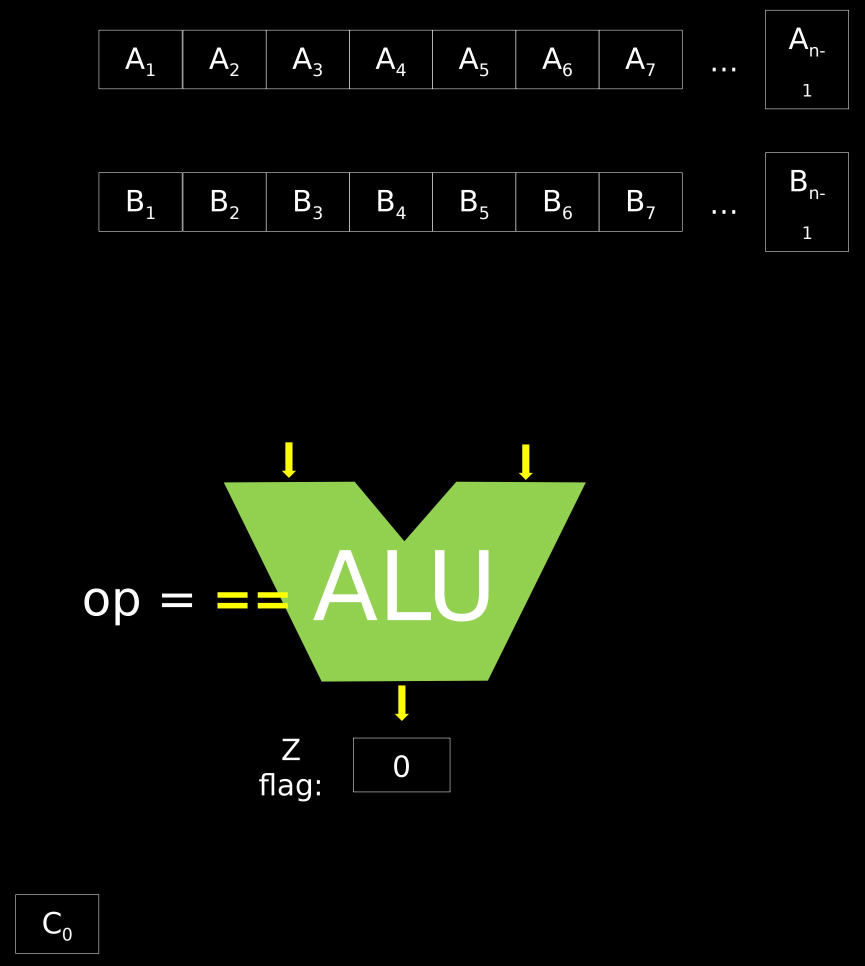
```
static float A[N], B[N], C[N];

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

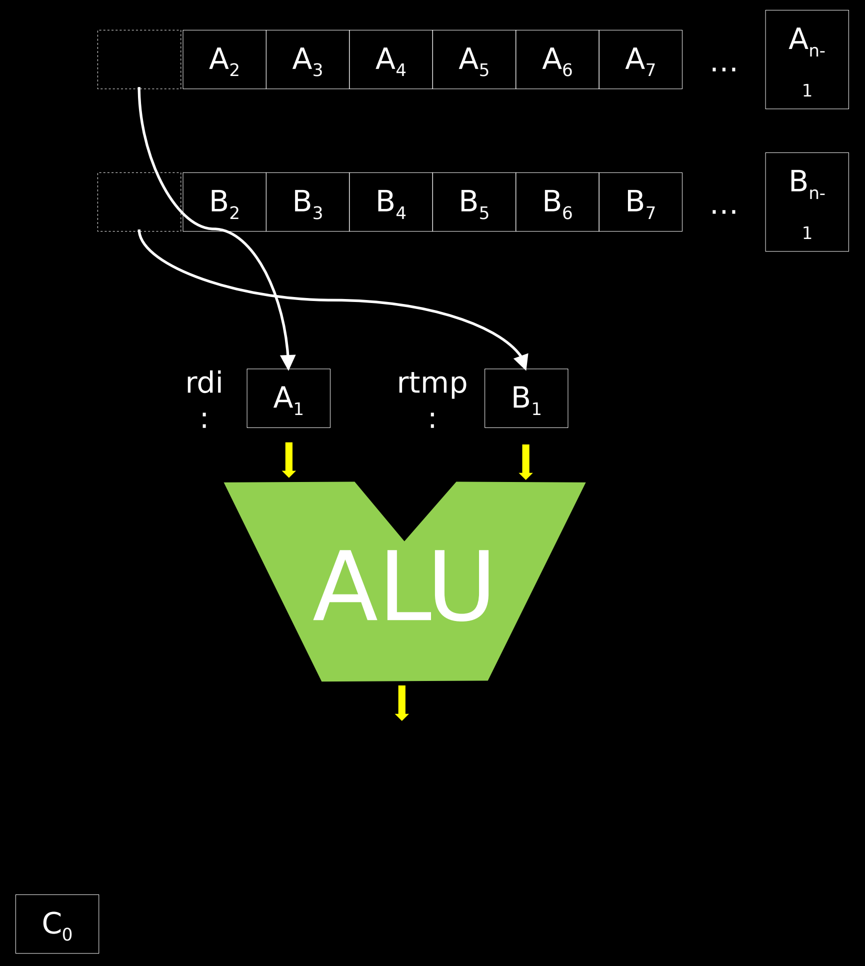
```
static float A[N], B[N], C[N];

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

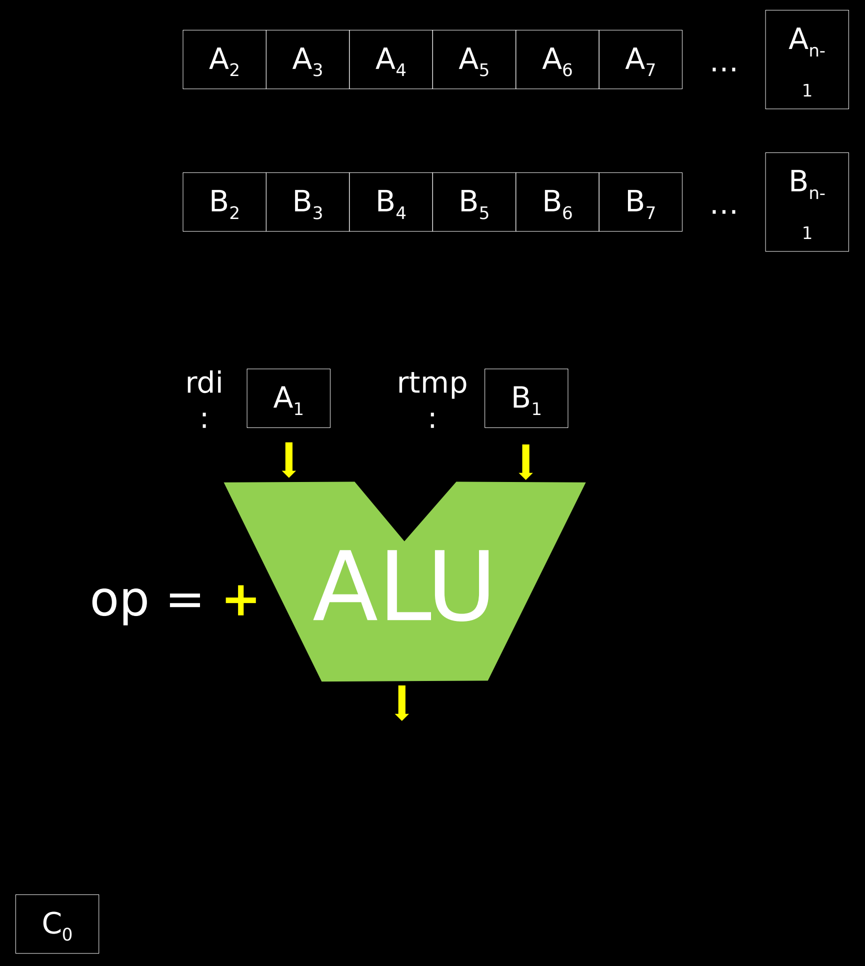
```
static float A[N], B[N], C[N];

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

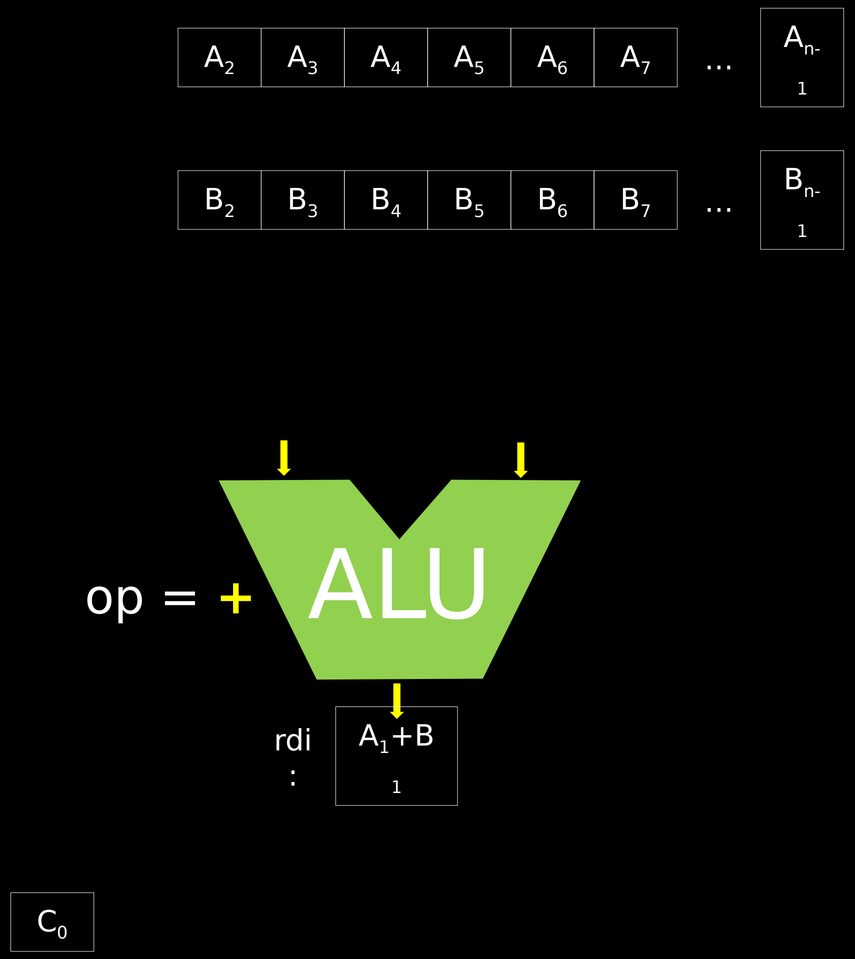
```
static float A[N], B[N], C[N];

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

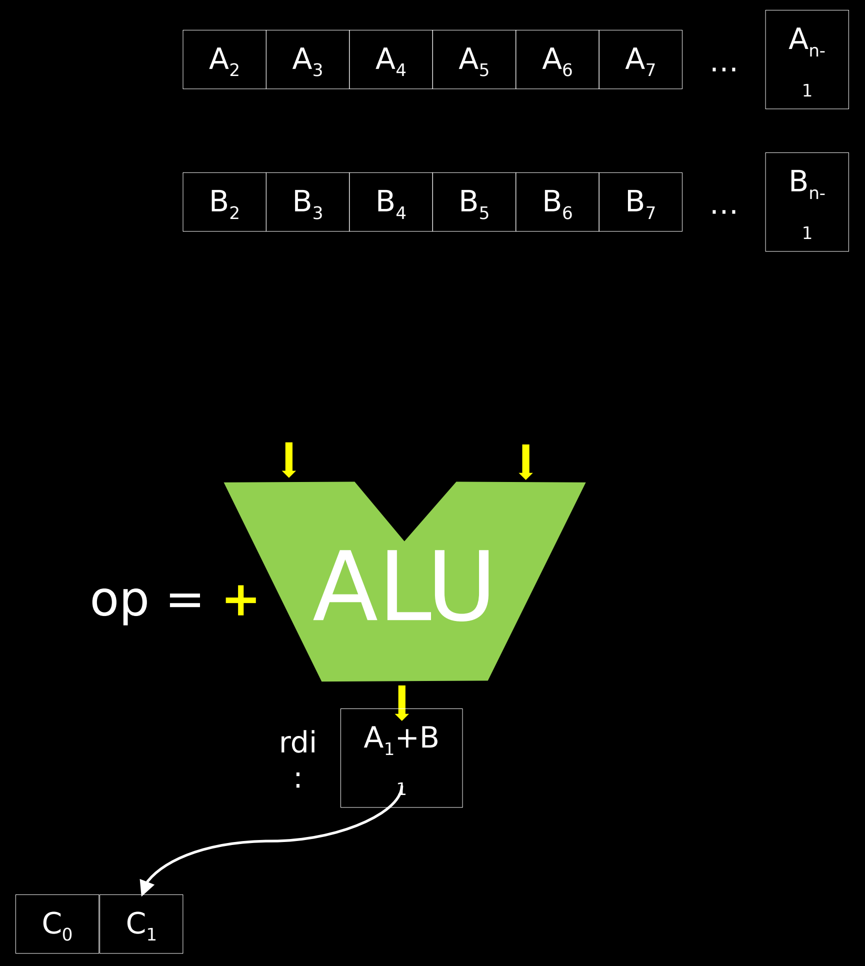
```
static float A[N], B[N], C[N];

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
```



Scalar processors

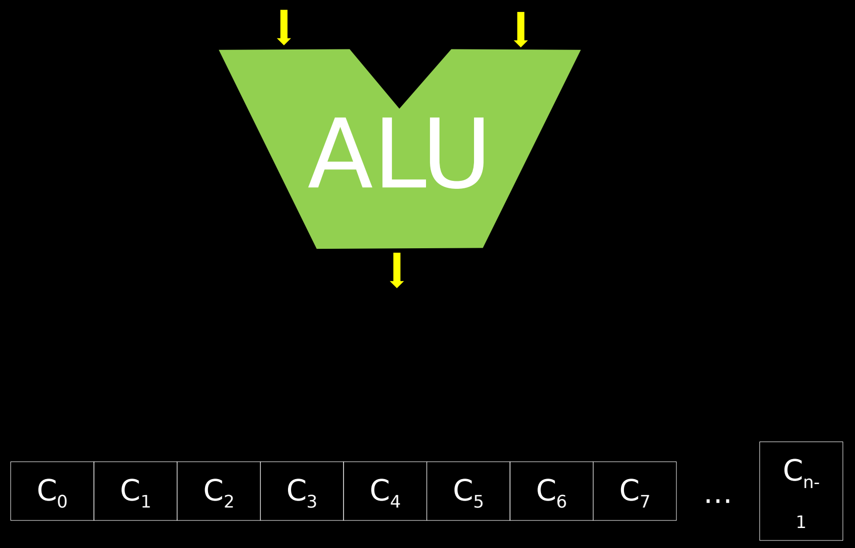
- Let us consider a simple vector addition

```
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
```



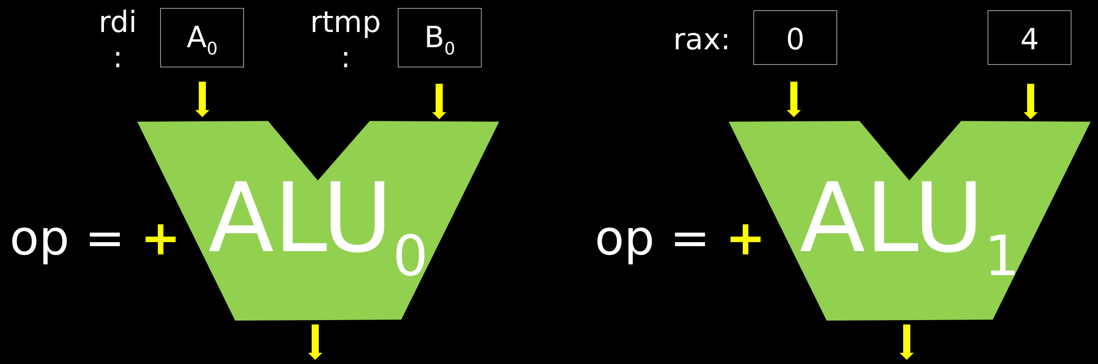
Scalar processors

- 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

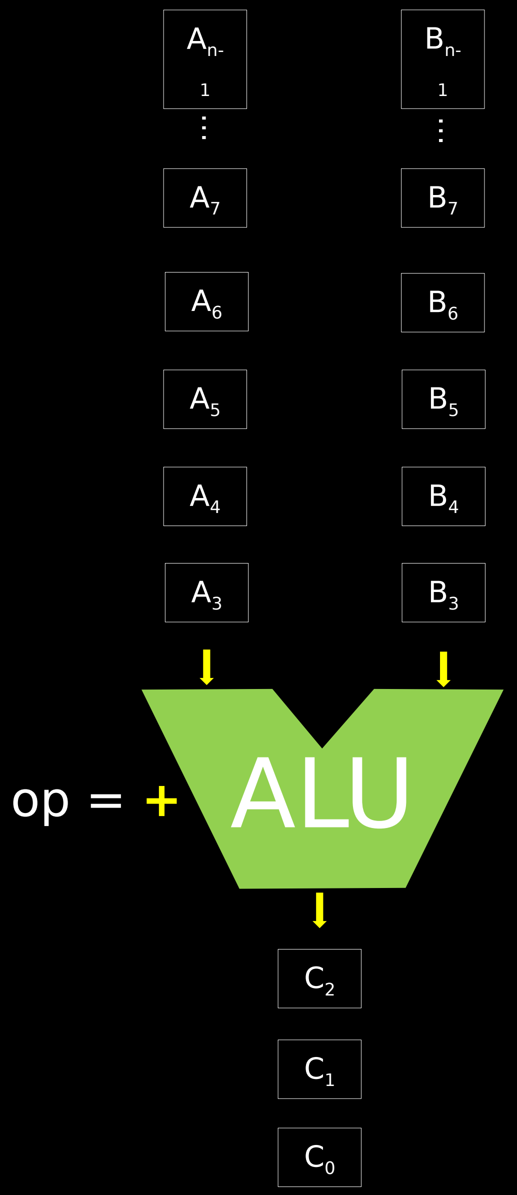


Data-level Parallelism

- 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

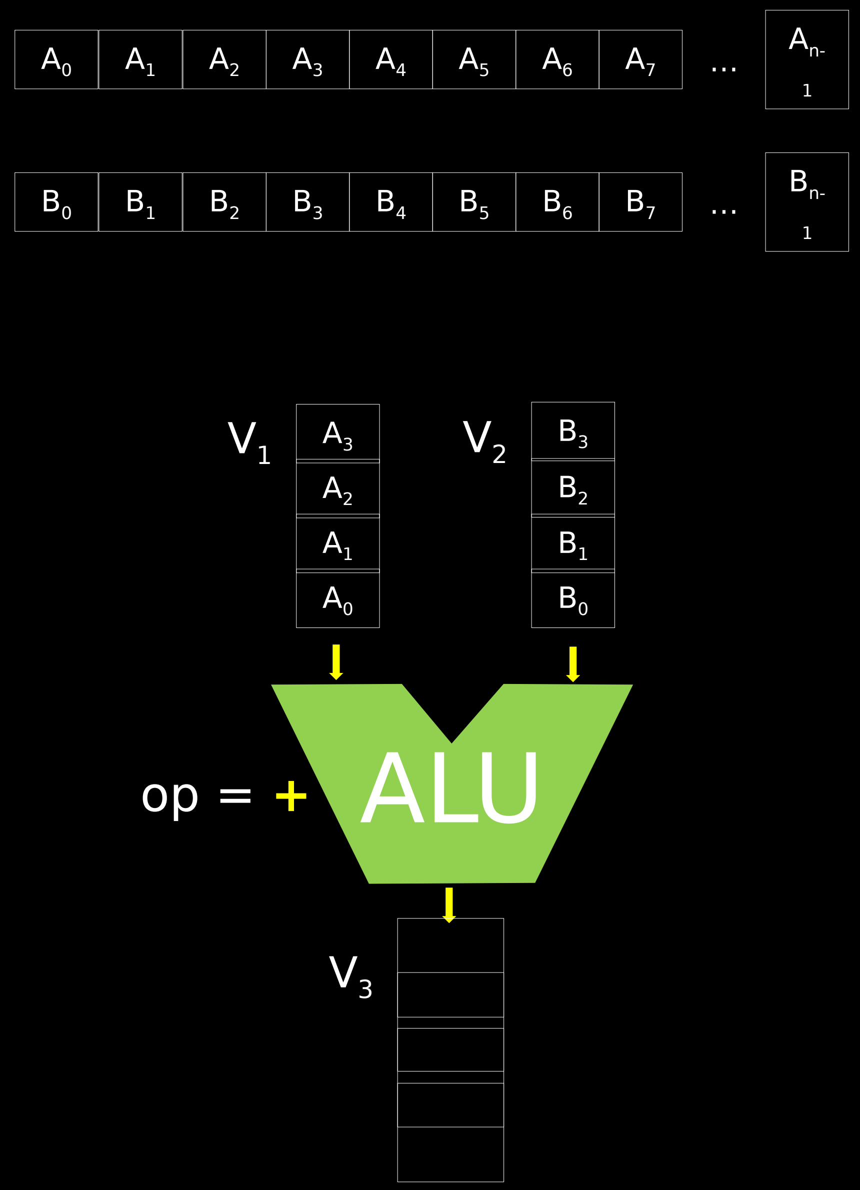
- Idea
  - Stream vectors through pipelined ALUs
    - Single ALU operation applied to multiple data
- Implementation based on
  - Specific vector instructions
    - vload, vstore, vadd, vmul, …
  - High-Bandwidth memory



Vector processors

- 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]
```

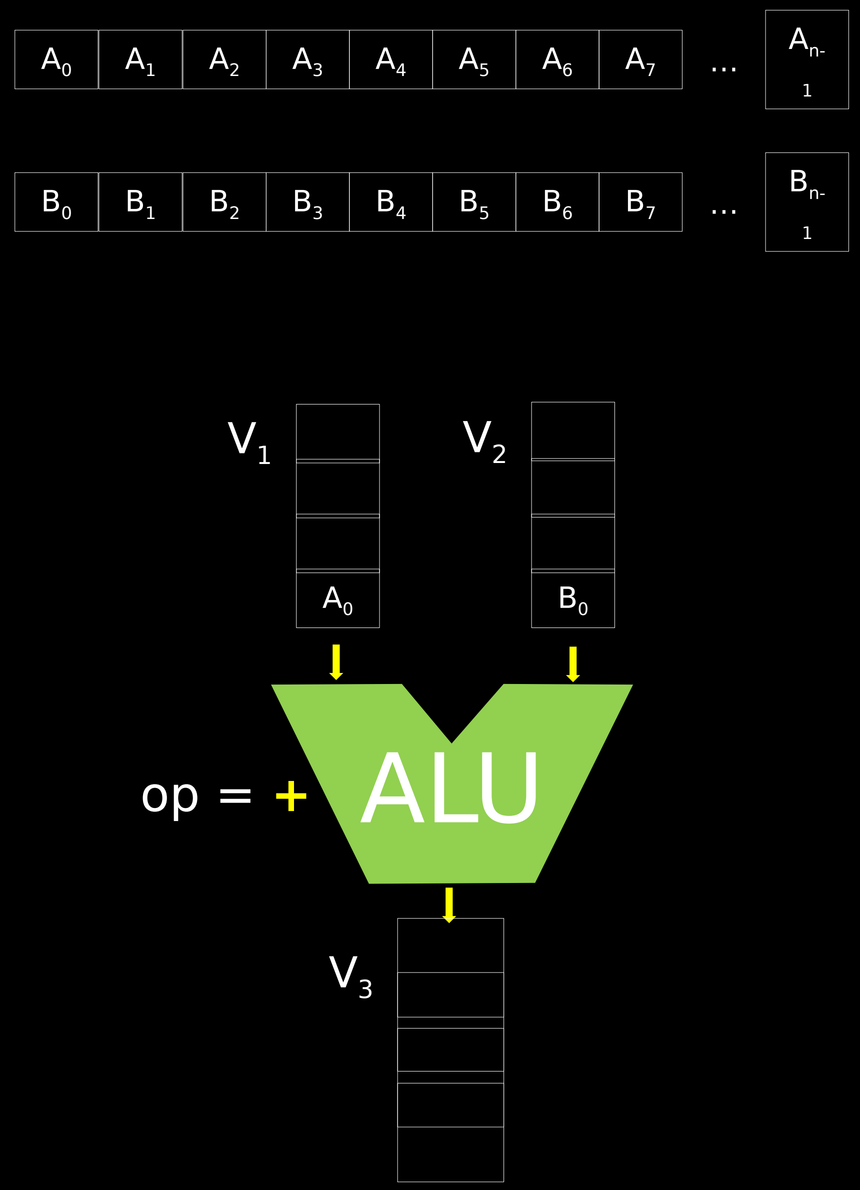


Vector processors

- 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

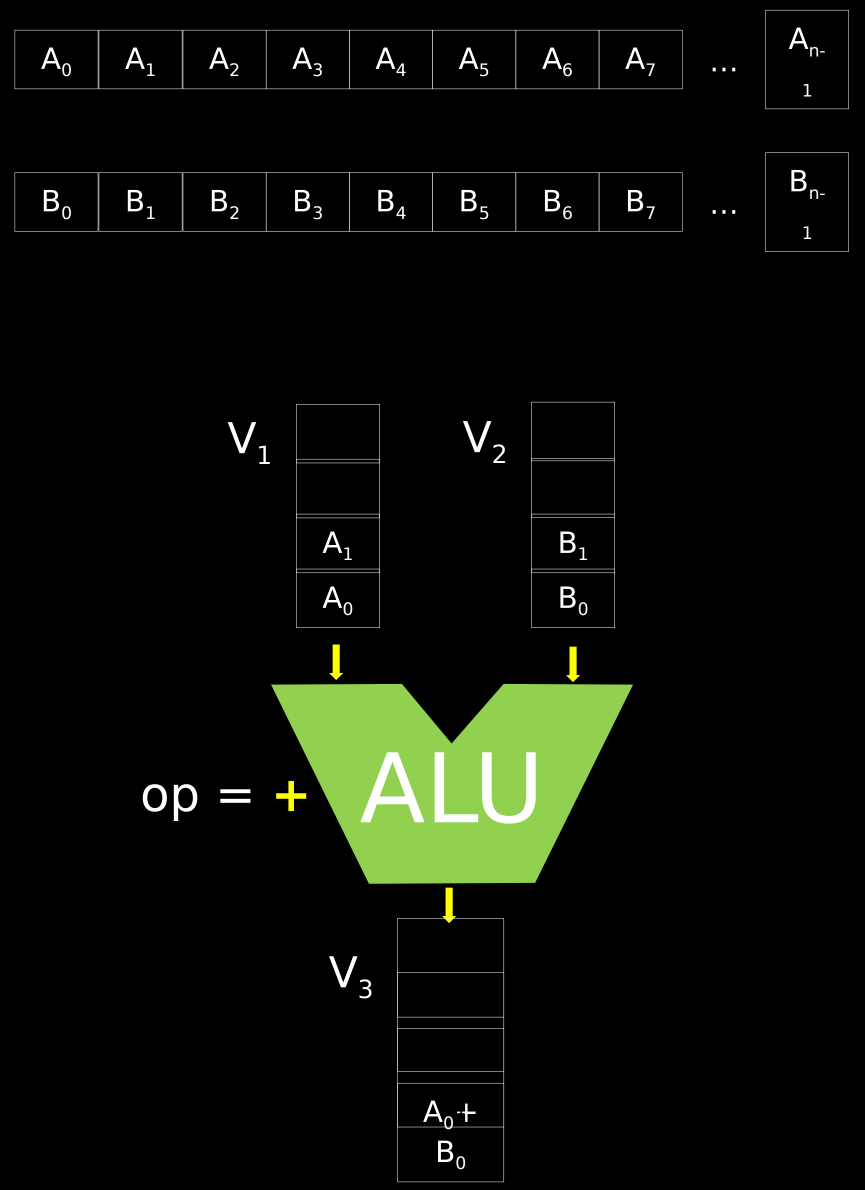


Vector processors

- 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

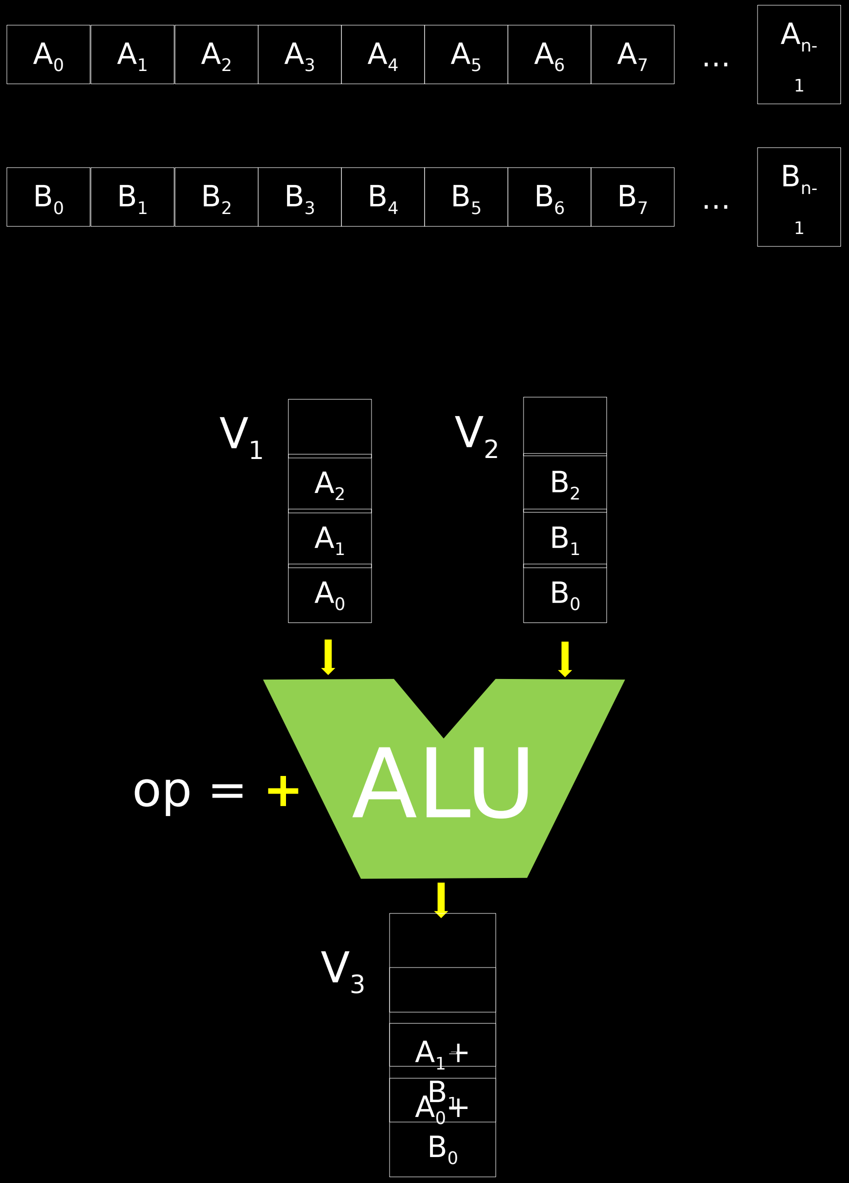


Vector processors

- 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

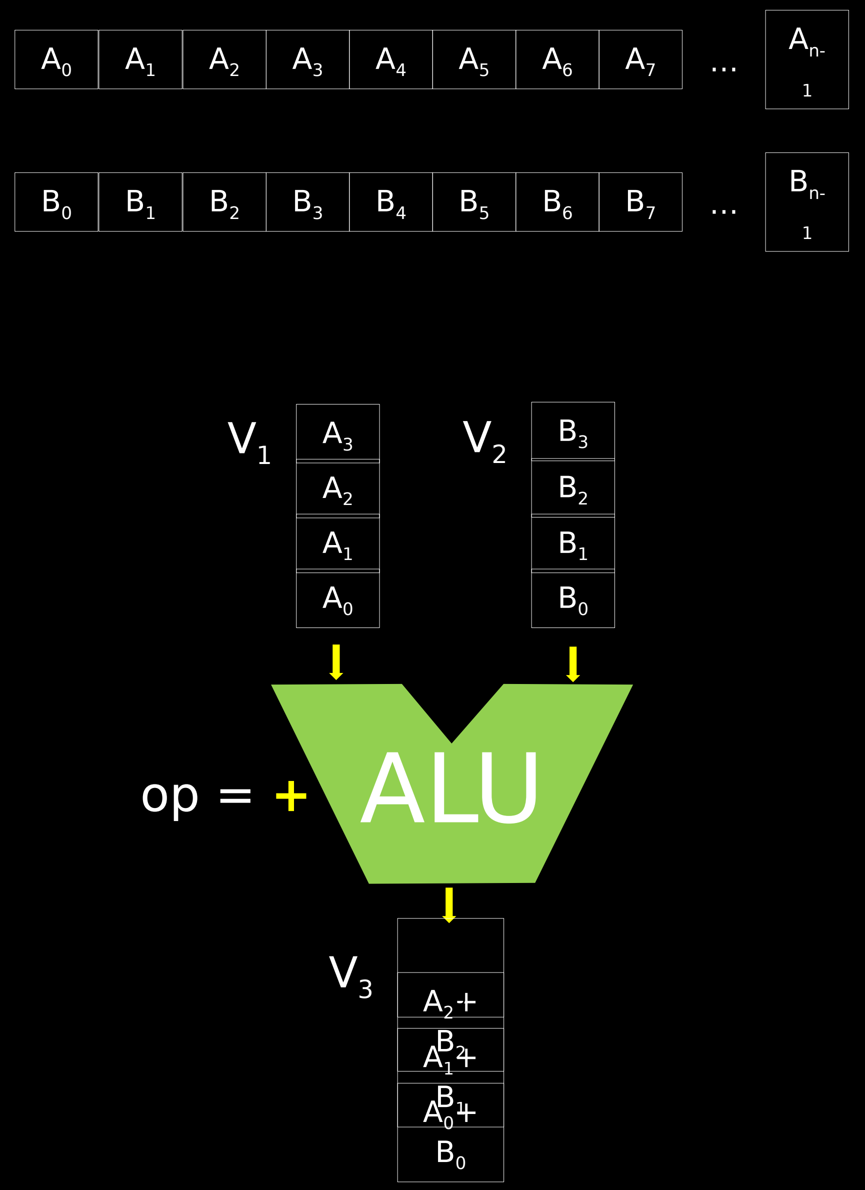


Vector processors

- 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

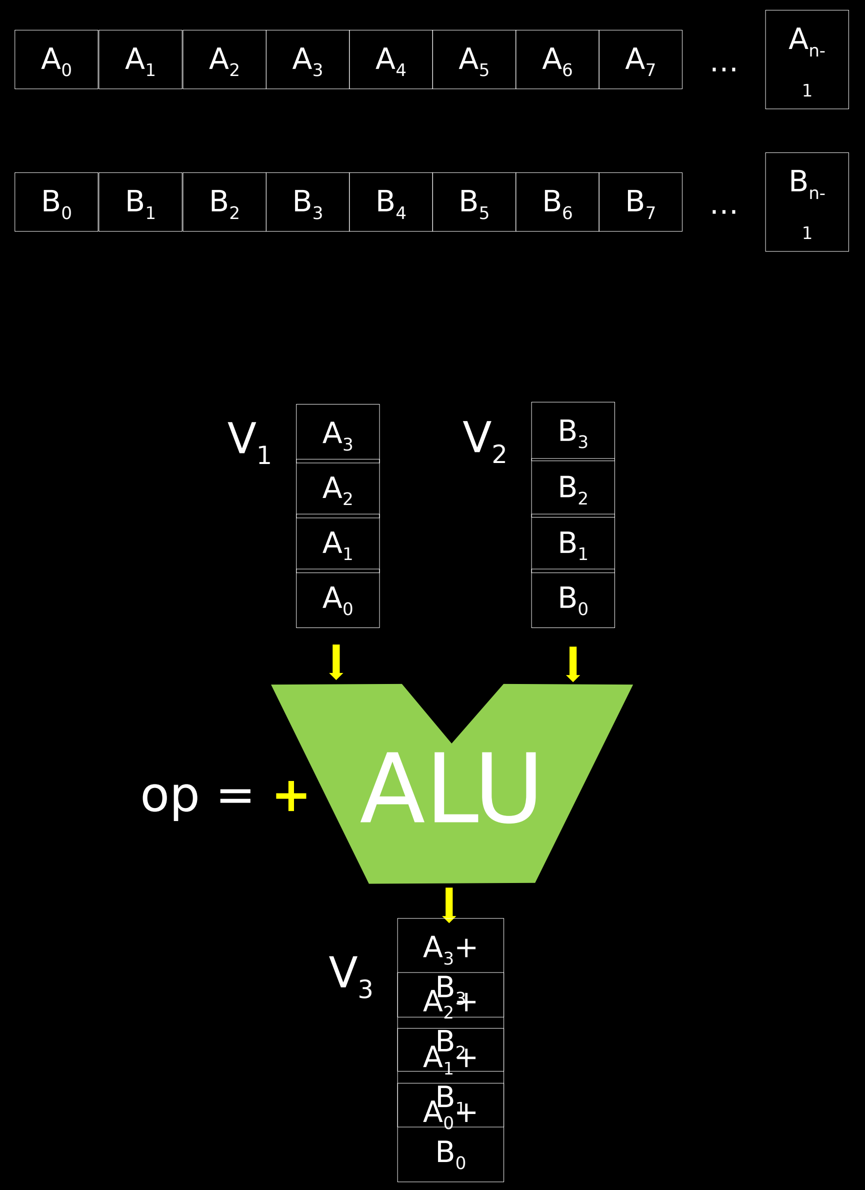


Vector processors

- 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

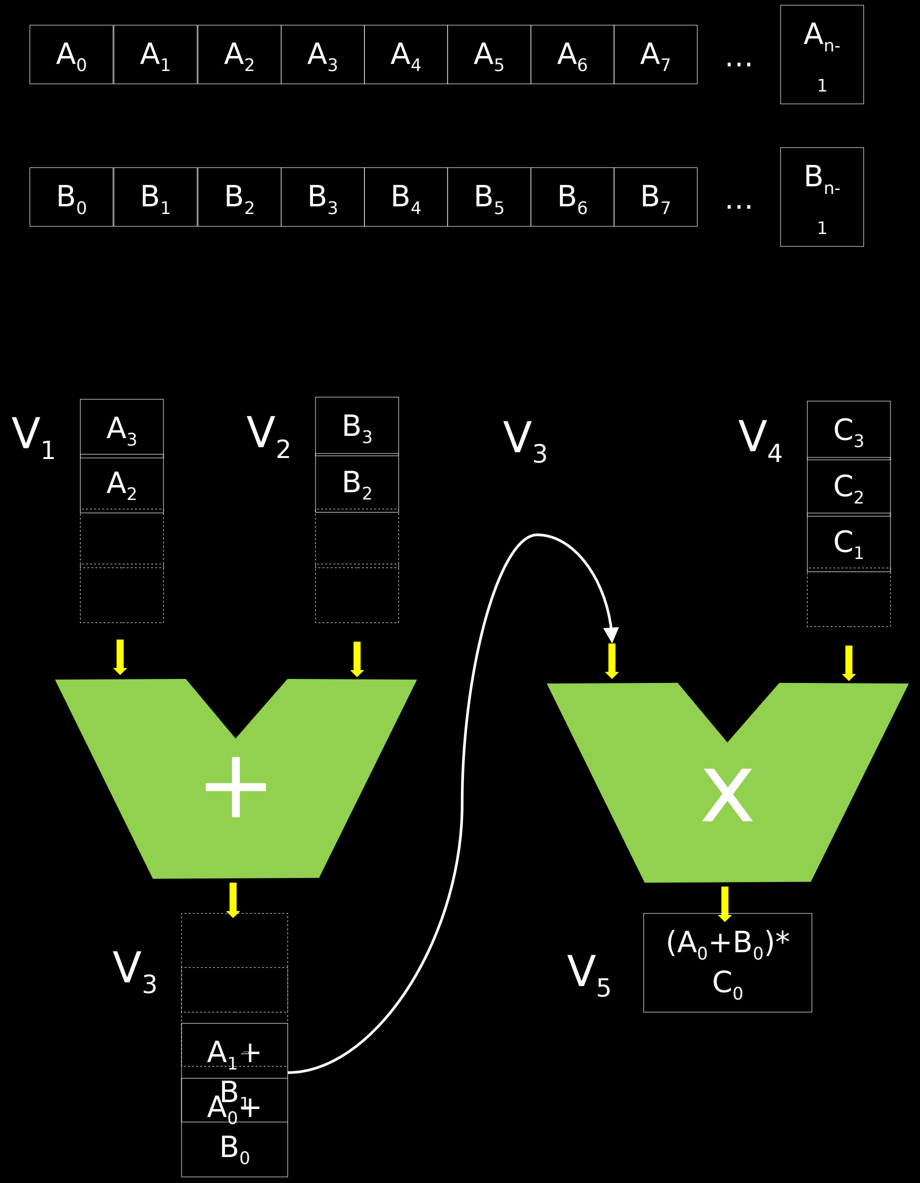


Vector processors

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

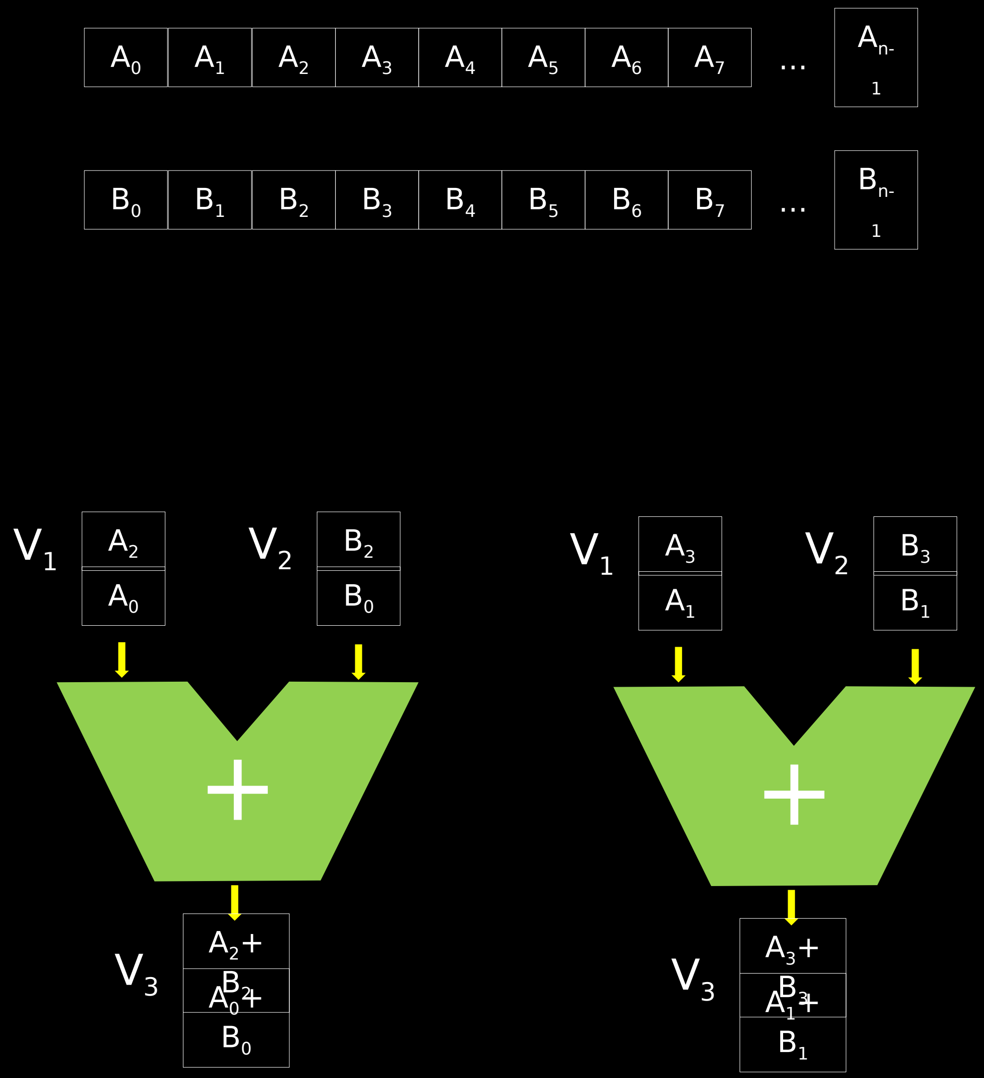
```
vadd v1, v2, v3
vmul v3, v4, v5
```

- 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

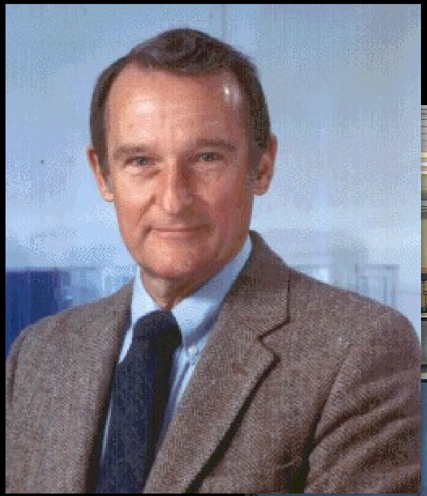


Vector Processors

- 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…

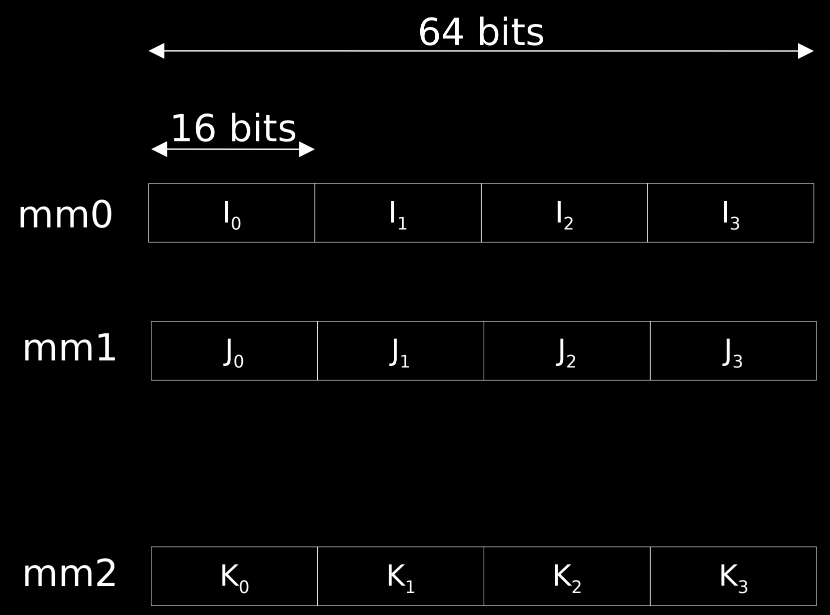
SIMD extensions

- 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!

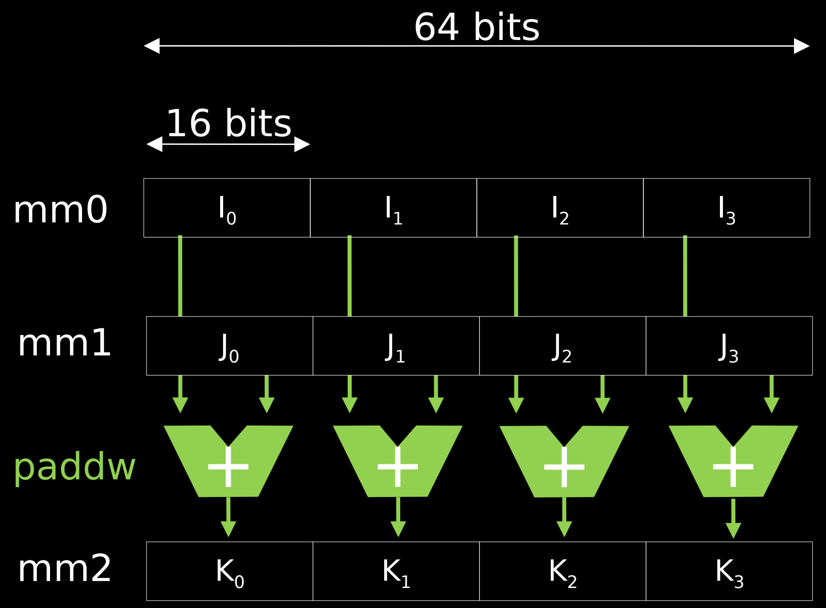
SIMD extensions

- 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



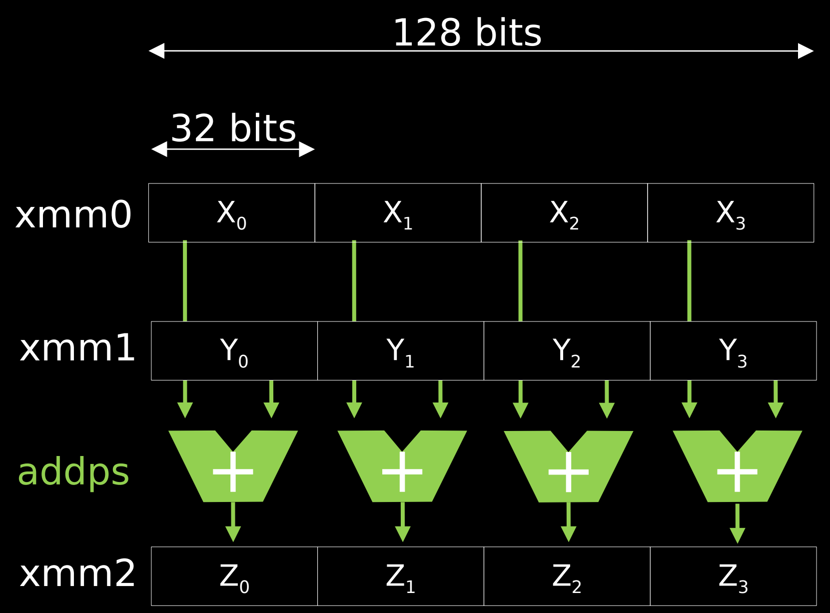
SIMD extensions

- 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}

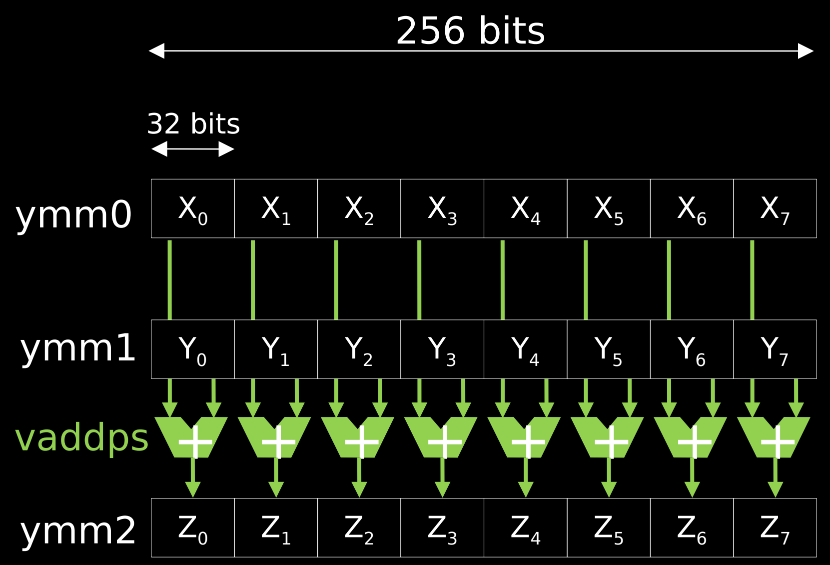
Add packed simple-precision floating point



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 😇

```
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
```

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/

```
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
```

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 <u>vectors</u> as inputs

a	?	?	?	?	?	?	?	?
x	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7

```
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

a	A	A	A	A	A	A	A	A
x	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7

```
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];
```

z_if	zi_0	zi_1	zi_2	zi_3	zi_4	zi_5	zi_6	zi_7
z_else	ze_0	ze_1	ze_2	ze_3	ze_4	ze_5	ze_6	ze_7

```
__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];
```

z_if	zi_0	zi_1	zi_2	zi_3	zi_4	zi_5	zi_6	zi_7
z_else	ze_0	ze_1	ze_2	ze_3	ze_4	ze_5	ze_6	ze_7

```
__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);
```

x	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7

fifty	50	50	50	50	50	50	50	50

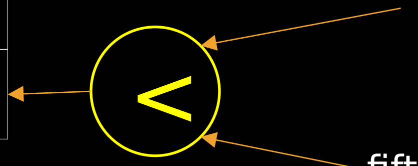
Intel Intrinsics

- 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_if	zi_0	zi_1	zi_2	zi_3	zi_4	zi_5	zi_6	zi_7
z_else	ze_0	ze_1	ze_2	ze_3	ze_4	ze_5	ze_6	ze_7
mask	0..0	0..0	1..1	0..0	1..1	1..1	1..1	0..0



x	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7
fifty	50	50	50	50	50	50	50	50

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];
```

z_if	zi_0	zi_1	zi_2	zi_3	zi_4	zi_5	zi_6	zi_7
z_else	ze_0	ze_1	ze_2	ze_3	ze_4	ze_5	ze_6	ze_7
mask	0..0	0..0	1..1	0..0	1..1	1..1	1..1	0..0
z	ze_0	ze_1	zi_2	ze_3	zi_4	zi_5	zi_6	ze_7

blendv

```
__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);
```

SIMD extensions

- 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
  - Complex memory layout
  - Intra-loop dependencies

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! 😉