Vector instructions

Wojciech Muła, 0x80.pl January 2021

Thanks to Roman Kurc & Daniel Lemire for valuable feedback

Vector instructions

Wojciech Muła, 0x80.pl January 2021
Thanks to Roman Kurc & Daniel Lemire for valuable feedback

What do we cover?

Vector instructions

Wojciech Muła, 0x80.pl January 2021
Thanks to Roman Kurc & Daniel Lemire for valuable feedback

What do we cover?

- What are **vector instructions/SIMD instructions**

Vector instructions

Wojciech Muła, 0x80.pl January 2021

Thanks to Roman Kurc & Daniel Lemire for valuable feedback

What do we cover?

- What are **vector instructions/SIMD instructions**
- Why are they important

Vector instructions

Wojciech Muła, 0x80.pl January 2021
Thanks to Roman Kurc & Daniel Lemire for valuable feedback

What do we cover?

- What are **vector instructions/SIMD instructions**
- Why are they important
- How do they work

Vector instructions

Wojciech Muła, 0x80.pl January 2021

Thanks to Roman Kurc & Daniel Lemire for valuable feedback

What do we cover?

- What are **vector instructions**/**SIMD instructions**
- Why are they important
- How do they work
- What are they good for

Where can we find vectors?

Where can we find vectors?

- Shortly: vectors are *everywhere*

Where can we find vectors?

- Shortly: vectors are *everywhere*
- All kinds of engineering simulations (mechanical, electrical, chemical)

Where can we find vectors?

- Shortly: vectors are *everywhere*
- All kinds of engineering simulations (mechanical, electrical, chemical)
- Scientific simulations (weather forecasting, detecting black holes, looking for new particles)

Where can we find vectors?

- Shortly: vectors are *everywhere*
- All kinds of engineering simulations (mechanical, electrical, chemical)
- Scientific simulations (weather forecasting, detecting black holes, looking for new particles)
- Digital signal processing (sound, video, 3D graphics, computer vision, compression)

Where can we find vectors?

- Shortly: vectors are *everywhere*
- All kinds of engineering simulations (mechanical, electrical, chemical)
- Scientific simulations (weather forecasting, detecting black holes, looking for new particles)
- Digital signal processing (sound, video, 3D graphics, computer vision, compression)
- Machine Learning

Where can we find vectors?

- Shortly: vectors are *everywhere*
- All kinds of engineering simulations (mechanical, electrical, chemical)
- Scientific simulations (weather forecasting, detecting black holes, looking for new particles)
- Digital signal processing (sound, video, 3D graphics, computer vision, compression)
- Machine Learning (sorry for the buzzword)

What is a vector?

- In maths a **vector** is a sequence of numbers (*scalars*)

What is a vector?

- In maths a **vector** is a sequence of numbers (*scalars*)
- The IT world uses the term **array** for vectors

What is a vector?

- In maths a **vector** is a sequence of numbers (*scalars*)
- The IT world uses the term **array** for vectors
- In programming an array of numbers models a vector
  $v = (1, 2, 3) \Rightarrow$ `int v[3] = {1, 2, 3};` (C/C++/Java)
  $v = (1, 2, 3) \Rightarrow$ `v = [1, 2, 3]` (Python)

What is a vector?

- In maths a **vector** is a sequence of numbers (*scalars*)
- The IT world uses the term **array** for vectors
- In programming an array of numbers models a vector
  $v = (1, 2, 3) \Rightarrow$ `int v[3] = {1, 2, 3};` (C/C++/Java)
  $v = (1, 2, 3) \Rightarrow$ `v = [1, 2, 3]` (Python)
- An array can hold anything, not only bare numbers, but also pixels (images), samples (sound), points (3D models), characters (text)

How would we add two vectors? – part 1

Suppose we have two vectors of size 8:

How would we add two vectors? – part 1

Suppose we have two vectors of size 8:

$$a = (1, 2, 3, 4, 5, 6, 7, 8)$$

How would we add two vectors? – part 1

Suppose we have two vectors of size 8:

$$a = (1, 2, 3, 4, 5, 6, 7, 8)$$

$$b = (7, 1, 4, 2, 3, 5, 1, 0)$$

How would we add two vectors? – part 1

Suppose we have two vectors of size 8:

$$a = (1, 2, 3, 4, 5, 6, 7, 8)$$

$$b = (7, 1, 4, 2, 3, 5, 1, 0)$$

Their sum c is:

$$c = a + b = (8, 3, 7, 6, 8, 11, 8, 8)$$

How would we add two vectors? – part 3

A program that performs vector addition is quite simple:

```
c[0] = a[0] + b[0]
c[1] = a[1] + b[1]
c[2] = a[2] + b[2]
c[3] = a[3] + b[3]
c[4] = a[4] + b[4]
c[5] = a[5] + b[5]
c[6] = a[6] + b[6]
c[7] = a[7] + b[7]
```

How would we add two vectors? – part 3

A program that performs vector addition is quite simple:

```
c[0] = a[0] + b[0]
c[1] = a[1] + b[1]
c[2] = a[2] + b[2]
c[3] = a[3] + b[3]
c[4] = a[4] + b[4]
c[5] = a[5] + b[5]
c[6] = a[6] + b[6]
c[7] = a[7] + b[7]
```

It can be written with a loop:

```
for (int i=0; i < 8; i++)
    c[i] = a[i] + b[i];
```

How would we add two vectors? – part 4

```
for (int i=0; i < 8; i++)
    c[i] = a[i] + b[i];
```

How would we add two vectors? – part 4

```
for (int i=0; i < 8; i++)
    c[i] = a[i] + b[i];
```

Despite the form, the number of basic instruction is:

How would we add two vectors? – part 4

```
for (int i=0; i < 8; i++)
    c[i] = a[i] + b[i];
```

Despite the form, the number of basic instruction is:

- 16 loads from memory (`a[0..7]` and `b[0..7]`)

How would we add two vectors? – part 4

```
for (int i=0; i < 8; i++)
    c[i] = a[i] + b[i];
```

Despite the form, the number of basic instruction is:

- 16 loads from memory (`a[0..7]` and `b[0..7]`)
- 8 additions (+)

How would we add two vectors? – part 4

```
for (int i=0; i < 8; i++)
    c[i] = a[i] + b[i];
```

Despite the form, the number of basic instruction is:

- 16 loads from memory (`a[0..7]` and `b[0..7]`)
- 8 additions (+)
- 8 stores to memory (`c[0..7]`)

Hardware dedicated to vector operations

Hardware dedicated to vector operations

- CPUs can boost vector operations by providing dedicated **vector instructions**

Hardware dedicated to vector operations

- CPUs can boost vector operations by providing dedicated **vector instructions**
- . . . often called **SIMD** instructions

Hardware dedicated to vector operations

- CPUs can boost vector operations by providing dedicated **vector instructions**
- ... often called **SIMD** instructions
- A vector instruction is a hardware implementation of given basic vector operation

Hardware dedicated to vector operations

- CPUs can boost vector operations by providing dedicated **vector instructions**
- . . . often called **SIMD** instructions
- A vector instruction is a hardware implementation of given basic vector operation
- Vector instructions were provided by supercomputers even in 1960's

Hardware dedicated to vector operations

- CPUs can boost vector operations by providing dedicated **vector instructions**
- ...often called **SIMD** instructions
- A vector instruction is a hardware implementation of given basic vector operation
- Vector instructions were provided by supercomputers even in 1960's
- In commodity hardware used in PC, laptops, phones the first vector instructions appeared in late 1990's

Hardware dedicated to vector operations

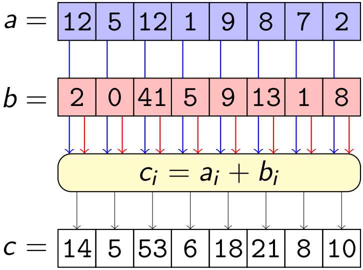
- CPUs can boost vector operations by providing dedicated **vector instructions**
- . . . often called **SIMD** instructions
- A vector instruction is a hardware implementation of given basic vector operation
- Vector instructions were provided by supercomputers even in 1960's
- In commodity hardware used in PC, laptops, phones the first vector instructions appeared in late 1990's
- GPUs usually have SIMD execution units

Hardware dedicated to vector operations

- CPUs can boost vector operations by providing dedicated **vector instructions**
- . . . often called **SIMD** instructions
- A vector instruction is a hardware implementation of given basic vector operation
- Vector instructions were provided by supercomputers even in 1960's
- In commodity hardware used in PC, laptops, phones the first vector instructions appeared in late 1990's
- GPUs usually have SIMD execution units
- Vector instructions are not the only mean of speeding up vector calculation, there are CPUs having *vector architectures* built entirely around the concept of arbitrary length vectors

How vector operations work?

Suppose vectors have 8 elements
Operation is $c = a + b$



What is SIMD?

What is SIMD?

- James Flynn in 1970's introduced rough classification of *computer systems*

What is SIMD?

- James Flynn in 1970's introduced rough classification of *computer systems*
- A computer can have **single** or **multiple** CPUs

What is SIMD?

- James Flynn in 1970's introduced rough classification of *computer systems*
- A computer can have **single** or **multiple** CPUs
- ... thus can execute single/multiple instructions at once

What is SIMD?

- James Flynn in 1970's introduced rough classification of *computer systems*
- A computer can have **single** or **multiple** CPUs
- . . . thus can execute single/multiple instructions at once
- An instruction can deal with either **single** or **multiple** data at once

What is SIMD?

- James Flynn in 1970's introduced rough classification of *computer systems*
- A computer can have **single** or **multiple** CPUs
- ...thus can execute single/multiple instructions at once
- An instruction can deal with either **single** or **multiple** data at once
- SIMD means: *Single Instructions, Multiple Data*

What is SIMD?

- James Flynn in 1970's introduced rough classification of *computer systems*
- A computer can have **single** or **multiple** CPUs
- ... thus can execute single/multiple instructions at once
- An instruction can deal with either **single** or **multiple** data at once
- SIMD means: *Single Instructions, Multiple Data*
- Here "multiple data" means "a vector"

What is SIMD?

- James Flynn in 1970's introduced rough classification of *computer systems*
- A computer can have **single** or **multiple** CPUs
- …thus can execute single/multiple instructions at once
- An instruction can deal with either **single** or **multiple** data at once
- SIMD means: *Single Instructions, Multiple Data*
- Here "multiple data" means "a vector"
- Most **CPU cores** work in SISD model: *Single Instruction, Single Data*

How it looks like in code

How it looks like in code

```
Va = vector_load(a);        // Va holds 8 elements
Vb = vector_load(b);        // ... likewise Vb
Vc = vector_add(Va, Vb);    // execute 8 additions
vector_store(c, Vc);        // save 8 elements from Vc
```

How it looks like in code

```
Va = vector_load(a);        // Va holds 8 elements
Vb = vector_load(b);        // ... likewise Vb
Vc = vector_add(Va, Vb);    // execute 8 additions
vector_store(c, Vc);        // save 8 elements from Vc
```

- Both approaches (slide 6) perform exactly the same program: the same amount of data is transferred from/to memory, the same number of additions is executed

How it looks like in code

```
Va = vector_load(a);        // Va holds 8 elements
Vb = vector_load(b);        // ... likewise Vb
Vc = vector_add(Va, Vb);    // execute 8 additions
vector_store(c, Vc);        // save 8 elements from Vc
```

- Both approaches (slide 6) perform exactly the same program: the same amount of data is transferred from/to memory, the same number of additions is executed
- In traditional approach we had 32 instructions

How it looks like in code

```
Va = vector_load(a);        // Va holds 8 elements
Vb = vector_load(b);        // ... likewise Vb
Vc = vector_add(Va, Vb);    // execute 8 additions
vector_store(c, Vc);        // save 8 elements from Vc
```

- Both approaches (slide 6) perform exactly the same program: the same amount of data is transferred from/to memory, the same number of additions is executed
- In traditional approach we had 32 instructions
- ...`vector_foo` represents a **single CPU instruction**

How it looks like in code

```
Va = vector_load(a);        // Va holds 8 elements
Vb = vector_load(b);        // ... likewise Vb
Vc = vector_add(Va, Vb);    // execute 8 additions
vector_store(c, Vc);        // save 8 elements from Vc
```

- Both approaches (slide 6) perform exactly the same program: the same amount of data is transferred from/to memory, the same number of additions is executed
- In traditional approach we had 32 instructions
- ... `vector_foo` represents a **single CPU instruction**
- Only four instructions are executed in total

How it looks like in code

```
Va = vector_load(a);       // Va holds 8 elements
Vb = vector_load(b);       // ... likewise Vb
Vc = vector_add(Va, Vb);   // execute 8 additions
vector_store(c, Vc);       // save 8 elements from Vc
```

- Both approaches (slide 6) perform exactly the same program: the same amount of data is transferred from/to memory, the same number of additions is executed
- In traditional approach we had 32 instructions
- ...`vector_foo` represents a **single CPU instruction**
- Only four instructions are executed in total
- What we pay for are **instuctions**

Is SIMD really faster?

Is SIMD really faster?

- "a vector instruction" doesn't just mean "a hardware loop"

Is SIMD really faster?

- "a vector instruction" doesn't just mean "a hardware loop"
- vector instructions do have dedicated execution units

Is SIMD really faster?

- "a vector instruction" doesn't just mean "a hardware loop"
- vector instructions do have dedicated execution units
- ...which perform elementary operations **in parallel**

Is SIMD really faster?

- "a vector instruction" doesn't just mean "a hardware loop"
- vector instructions do have dedicated execution units
- ...which perform elementary operations **in parallel**
- for example on Intel Skylake-X:

Is SIMD really faster?

- "a vector instruction" doesn't just mean "a hardware loop"
- vector instructions do have dedicated execution units
- ...which perform elementary operations **in parallel**
- for example on Intel Skylake-X:
- ...addition of two 8-bit numbers takes

Is SIMD really faster?

- "a vector instruction" doesn't just mean "a hardware loop"
- vector instructions do have dedicated execution units
- ...which perform elementary operations **in parallel**
- for example on Intel Skylake-X:
- ...addition of two 8-bit numbers takes 1 CPU cycle

Is SIMD really faster?

- "a vector instruction" doesn't just mean "a hardware loop"
- vector instructions do have dedicated execution units
- ...which perform elementary operations **in parallel**
- for example on Intel Skylake-X:
- ...addition of two 8-bit numbers takes 1 CPU cycle
- ...addition of two **16** x 8-bit vectors takes

Is SIMD really faster?

- "a vector instruction" doesn't just mean "a hardware loop"
- vector instructions do have dedicated execution units
- ...which perform elementary operations **in parallel**
- for example on Intel Skylake-X:
- ...addition of two 8-bit numbers takes 1 CPU cycle
- ...addition of two **16** x 8-bit vectors takes 1 CPU cycle

Is SIMD really faster?

- "a vector instruction" doesn't just mean "a hardware loop"
- vector instructions do have dedicated execution units
- ...which perform elementary operations **in parallel**
- for example on Intel Skylake-X:
- ...addition of two 8-bit numbers takes 1 CPU cycle
- ...addition of two **16** x 8-bit vectors takes 1 CPU cycle
- ...addition of two **32** x 8-bit vectors takes

Is SIMD really faster?

- "a vector instruction" doesn't just mean "a hardware loop"
- vector instructions do have dedicated execution units
- . . . which perform elementary operations **in parallel**
- for example on Intel Skylake-X:
- . . . addition of two 8-bit numbers takes 1 CPU cycle
- . . . addition of two **16** x 8-bit vectors takes 1 CPU cycle
- . . . addition of two **32** x 8-bit vectors takes 1 CPU cycle

Is SIMD really faster?

- "a vector instruction" doesn't just mean "a hardware loop"
- vector instructions do have dedicated execution units
- . . . which perform elementary operations **in parallel**
- for example on Intel Skylake-X:
- . . . addition of two 8-bit numbers takes 1 CPU cycle
- . . . addition of two **16** x 8-bit vectors takes 1 CPU cycle
- . . . addition of two **32** x 8-bit vectors takes 1 CPU cycle
- . . . addition of two **64** x 8-bit vectors takes

Is SIMD really faster?

- "a vector instruction" doesn't just mean "a hardware loop"
- vector instructions do have dedicated execution units
- ...which perform elementary operations **in parallel**
- for example on Intel Skylake-X:
- ...addition of two 8-bit numbers takes 1 CPU cycle
- ...addition of two **16** x 8-bit vectors takes 1 CPU cycle
- ...addition of two **32** x 8-bit vectors takes 1 CPU cycle
- ...addition of two **64** x 8-bit vectors takes 1 CPU cycle

Is SIMD really faster?

- "a vector instruction" doesn't just mean "a hardware loop"
- vector instructions do have dedicated execution units
- ...which perform elementary operations **in parallel**
- for example on Intel Skylake-X:
- ...addition of two 8-bit numbers takes 1 CPU cycle
- ...addition of two **16** x 8-bit vectors takes 1 CPU cycle
- ...addition of two **32** x 8-bit vectors takes 1 CPU cycle
- ...addition of two **64** x 8-bit vectors takes 1 CPU cycle
- Not for all operations such nice scaling is possible

Is SIMD really faster?

- "a vector instruction" doesn't just mean "a hardware loop"
- vector instructions do have dedicated execution units
- ...which perform elementary operations **in parallel**
- for example on Intel Skylake-X:
- ...addition of two 8-bit numbers takes 1 CPU cycle
- ...addition of two **16** x 8-bit vectors takes 1 CPU cycle
- ...addition of two **32** x 8-bit vectors takes 1 CPU cycle
- ...addition of two **64** x 8-bit vectors takes 1 CPU cycle
- Not for all operations such nice scaling is possible
- ...but we can expect significant boost over most of regular CPU instructions

What is a hardware vector?

What is a hardware vector?

- In SIMD model, the hardware vectors have **fixed size**

What is a hardware vector?

- In SIMD model, the hardware vectors have **fixed size**
- The size depends on CPU architectures: usually it is 128 bits, 256 bits or 512 bits

What is a hardware vector?

- In SIMD model, the hardware vectors have **fixed size**
- The size depends on CPU architectures: usually it is 128 bits, 256 bits or 512 bits
- Hardware **interprets** these bits as
  1) an array of integers or
  2) an array of floating point numbers

What is a hardware vector?

- In SIMD model, the hardware vectors have **fixed size**
- The size depends on CPU architectures: usually it is 128 bits, 256 bits or 512 bits
- Hardware **interprets** these bits as
  1) an array of integers or
  2) an array of floating point numbers
- Unlike regular CPUs there is no distinction between integer and floating-point registers — there are just "vector/SIMD registers"

What is a hardware vector?

- In SIMD model, the hardware vectors have **fixed size**
- The size depends on CPU architectures: usually it is 128 bits, 256 bits or 512 bits
- Hardware **interprets** these bits as
  1) an array of integers or
  2) an array of floating point numbers
- Unlike regular CPUs there is no distinction between integer and floating-point registers — there are just "vector/SIMD registers"
- Some CPU architectures support only integer operations

Hardware vs software vectors

For instance a 256-bit vector can be used in a program as the following vectors (C/C++ types)

- `int8_t[32]`, `uint8_t[32]`
- `int16_t[16]`, `uint16_t[16]`
- `int32_t[8]`, `uint32_t[8]`
- `int64_t[4]`, `uint64_t[4]`
- `float[8]`
- `double[4]`

Existing SIMD implementations

cryptic name	vendor	year	vector width [bits]
MMX	Intel	1997	64
3DNow	AMD	1998	64
AltiVec	many	1998	128
SSE	Intel	1999	128
—	ARM	2002	32
SSE2	Intel	2001	128
SSE3	Intel	2004	128
SSSE3	Intel	2006	128
SSE4	Intel	2007	128
AVX	Intel	2008	256
XOP	AMD	2010	128
Neon	ARM	2011	64
AVX2	Intel	2013	256
Neon	ARM	2014	128
AVX-512	Intel	2015	512
SVE	ARM	???	1024-4096

Most common SIMD operations

Most common SIMD operations

- store in memory / load from memory

Most common SIMD operations

- store in memory / load from memory
- addition / subtraction / multiplication / division (only floating-point division)

Most common SIMD operations

- store in memory / load from memory
- addition / subtraction / multiplication / division (only floating-point division)
- comparison ($=$, $\neq$, $<$, $\leq$, $>$, $\geq$)

Most common SIMD operations

- store in memory / load from memory
- addition / subtraction / multiplication / division (only floating-point division)
- comparison ($=$, $\neq$, $<$, $\leq$, $>$, $\geq$)
- square root / reciprocal ($1/x$)

Most common SIMD operations

- store in memory / load from memory
- addition / subtraction / multiplication / division (only floating-point division)
- comparison ($=$, $\neq$, $<$, $\leq$, $>$, $\geq$)
- square root / reciprocal ($1/x$)
- min / max / abolute value / average

Most common SIMD operations

- store in memory / load from memory
- addition / subtraction / multiplication / division (only floating-point division)
- comparison ($=$, $\neq$, $<$, $\leq$, $>$, $\geq$)
- square root / reciprocal ($1/x$)
- min / max / abolute value / average
- casts between vectors of floating point and integers

Most common SIMD operations

- store in memory / load from memory
- addition / subtraction / multiplication / division (only floating-point division)
- comparison ($=$, $\neq$, $<$, $\leq$, $>$, $\geq$)
- square root / reciprocal ($1/x$)
- min / max / abolute value / average
- casts between vectors of floating point and integers
- untyped operation on bits (like `and`, `xor`, `or`)

Most common SIMD operations

- store in memory / load from memory
- addition / subtraction / multiplication / division (only floating-point division)
- comparison ($=$, $\neq$, $<$, $\leq$, $>$, $\geq$)
- square root / reciprocal ($1/x$)
- min / max / abolute value / average
- casts between vectors of floating point and integers
- untyped operation on bits (like `and`, `xor`, `or`)
- **shuffle** or **permute** vector — change order of elements

Most common SIMD operations

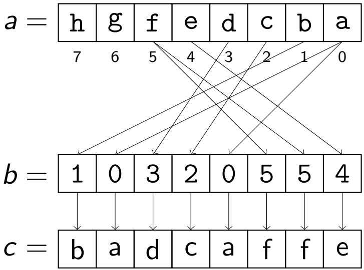
- store in memory / load from memory
- addition / subtraction / multiplication / division (only floating-point division)
- comparison ($=$, $\neq$, $<$, $\leq$, $>$, $\geq$)
- square root / reciprocal ($1/x$)
- min / max / abolute value / average
- casts between vectors of floating point and integers
- untyped operation on bits (like `and`, `xor`, `or`)
- **shuffle** or **permute** vector — change order of elements
- **blending** two vectors (ternary operation `s ? a : b`)

Most common SIMD operations

- store in memory / load from memory
- addition / subtraction / multiplication / division (only floating-point division)
- comparison ($=$, $\neq$, $<$, $\leq$, $>$, $\geq$)
- square root / reciprocal ($1/x$)
- min / max / abolute value / average
- casts between vectors of floating point and integers
- untyped operation on bits (like `and`, `xor`, `or`)
- **shuffle** or **permute** vector — change order of elements
- **blending** two vectors (ternary operation `s ? a : b`)
- integer addition / subtraction / type casts using **saturated arithmetic**

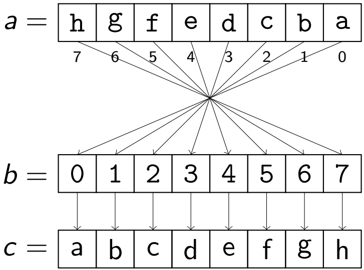
Example of shuffle — arbitrary order of elements

Operation is $c = \mathrm{shuffle}(a, b)$



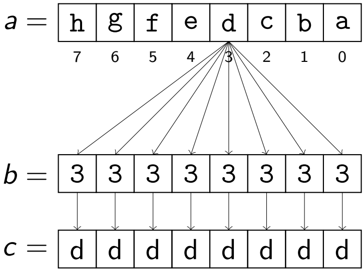
Example of shuffle — reverse

Operation is $c = \mathrm{shuffle}(a, b)$



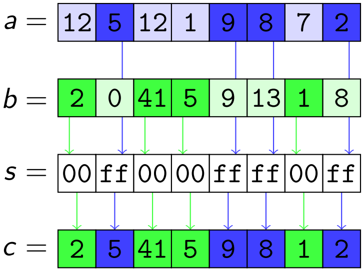
Example of shuffle — broadcast

Operation is $c = \mathrm{shuffle}(a, b)$



Example of blend

Operation is $c = s \, ? \, a : b$



As a raw bit operations $c = (s \,\mathrm{and}\, a) \,\mathrm{or} (\mathrm{not}\, s \,\mathrm{and}\, b)$

Integer saturated arithmetic

Integer saturated arithmetic

- A unique feature of SIMD

Integer saturated arithmetic

- A unique feature of SIMD
- . . . but something popular in the DSP world

Integer saturated arithmetic

- A unique feature of SIMD
- ... but something popular in the DSP world
- Prevents from overflows during integer operations

Integer saturated arithmetic

- A unique feature of SIMD
- ...but something popular in the DSP world
- Prevents from overflows during integer operations
- ...saves min or max value for given type

Integer saturated arithmetic

- A unique feature of SIMD
- . . . but something popular in the DSP world
- Prevents from overflows during integer operations
- . . . saves min or max value for given type
- . . . 0 or 255 for 8-bit unsigned integers

Integer saturated arithmetic

- A unique feature of SIMD
- ... but something popular in the DSP world
- Prevents from overflows during integer operations
- ... saves min or max value for given type
- ... 0 or 255 for 8-bit unsigned integers
- Wrap-around (modulo) arithmetic:

Integer saturated arithmetic

- A unique feature of SIMD
- ... but something popular in the DSP world
- Prevents from overflows during integer operations
- ... saves min or max value for given type
- ... 0 or 255 for 8-bit unsigned integers
- Wrap-around (modulo) arithmetic:
- $(240 + 100) \mod 256 = 84$

Integer saturated arithmetic

- A unique feature of SIMD
- ... but something popular in the DSP world
- Prevents from overflows during integer operations
- ... saves min or max value for given type
- ... 0 or 255 for 8-bit unsigned integers
- Wrap-around (modulo) arithmetic:
- $(240 + 100) \mod 256 = 84$
- Saturated arithmetics:

Integer saturated arithmetic

- A unique feature of SIMD
- ... but something popular in the DSP world
- Prevents from overflows during integer operations
- ... saves min or max value for given type
- ... 0 or 255 for 8-bit unsigned integers
- Wrap-around (modulo) arithmetic:
- $(240 + 100) \mod 256 = 84$
- Saturated arithmetics:
- $\min(240 + 100, 255) = \min(340, 255) = 255$

Integer saturated arithmetic

- A unique feature of SIMD
- ... but something popular in the DSP world
- Prevents from overflows during integer operations
- ... saves min or max value for given type
- ... 0 or 255 for 8-bit unsigned integers
- Wrap-around (modulo) arithmetic:
- $$(240 + 100) \mod 256 = 84$$
- Saturated arithmetics:
- $$\min(240 + 100, 255) = \min(340, 255) = 255$$
- Saturated arithmetic is as fast as wrap-around one

Saturated addition example — increase image brightness

Saturated addition example — increase image brightness

Wrap-around arithmetic

 $+ 180 =$ 

Saturated addition example — increase image brightness

Wrap-around arithmetic

$$+\ 180 =$$

Saturated arithmetic

$$+\ 180 =$$

SIMD instructions in real life

How about adding two vectors of arbitrary size? ($c = a + b$)

SIMD instructions in real life

How about adding two vectors of arbitrary size? ($c = a + b$)

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

SIMD instructions in real life

How about adding two vectors of arbitrary size? ($c = a + b$)

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

Function `vector_add` can rewritten (*vectorized*) as:

```
void vector_add(float* a, float* b, float* c, size_t N) {
    for (size_t i=0; i < N; i += 8) {
        auto Va = vector_load(a + i);
        auto Vb = vector_load(b + i);
        auto Vc = vector_add(Va, Vb);
        vector_store(c + i, Vc);
    }

    for (size_t i=(N / 8) * 8; i < N; i++)
        c[i] = a[i] + b[i];
}
```

Is it really better?

```
void vector_add(float* a, float* b, float* c, size_t N) {
    for (size_t i=0; i < N; i += 8) {
        auto Va = vector_load(a + i);
        auto Vb = vector_load(b + i);
        auto Vc = vector_add(Va, Vb);
        vector_store(c + i, Vc);
    }

    for (size_t i=(N / 8) * 8; i < N; i++)
        c[i] = a[i] + b[i];
}
```

Is it really better?

```
void vector_add(float* a, float* b, float* c, size_t N) {
    for (size_t i=0; i < N; i += 8) {
        auto Va = vector_load(a + i);
        auto Vb = vector_load(b + i);
        auto Vc = vector_add(Va, Vb);
        vector_store(c + i, Vc);
    }

    for (size_t i=(N / 8) * 8; i < N; i++)
        c[i] = a[i] + b[i];
}
```

- Now it is more complicated, as there are **two loops**

Is it really better?

```
void vector_add(float* a, float* b, float* c, size_t N) {
    for (size_t i=0; i < N; i += 8) {
        auto Va = vector_load(a + i);
        auto Vb = vector_load(b + i);
        auto Vc = vector_add(Va, Vb);
        vector_store(c + i, Vc);
    }

    for (size_t i=(N / 8) * 8; i < N; i++)
        c[i] = a[i] + b[i];
}
```

- Now it is more complicated, as there are **two loops**
- We need to know how these magic `vector_foo` functions work to reason about the code

Is it really better?

```
void vector_add(float* a, float* b, float* c, size_t N) {
    for (size_t i=0; i < N; i += 8) {
        auto Va = vector_load(a + i);
        auto Vb = vector_load(b + i);
        auto Vc = vector_add(Va, Vb);
        vector_store(c + i, Vc);
    }

    for (size_t i=(N / 8) * 8; i < N; i++)
        c[i] = a[i] + b[i];
}
```

- Now it is more complicated, as there are **two loops**
- We need to know how these magic `vector_foo` functions work to reason about the code
- The additional loop processes the **tail** of input — it will execute just $0 \ldots 7$ iterations, but has to (re)implement whole logic of the main loop

Is it really better?

```
void vector_add(float* a, float* b, float* c, size_t N) {
    for (size_t i=0; i < N; i += 8) {
        auto Va = vector_load(a + i);
        auto Vb = vector_load(b + i);
        auto Vc = vector_add(Va, Vb);
        vector_store(c + i, Vc);
    }

    for (size_t i=(N / 8) * 8; i < N; i++)
        c[i] = a[i] + b[i];
}
```

- Now it is more complicated, as there are **two loops**
- We need to know how these magic `vector_foo` functions work to reason about the code
- The additional loop processes the **tail** of input — it will execute just $0 \ldots 7$ iterations, but has to (re)implement whole logic of the main loop
- What if we wanted to port it for another CPU, which is capable to process 16, 32 or 64 numbers?

But SIMD is faster!

But SIMD is faster!

More complex code pays off in peformance boost

Results gathered from several articles from my website

But SIMD is faster!

More complex code pays off in peformance boost

Results gathered from several articles from my website

- base64 decoding: 2 x faster

But SIMD is faster!

More complex code pays off in peformance boost

Results gathered from several articles from my website

- base64 decoding: 2 x faster
- reverse table: 3 x faster

But SIMD is faster!

More complex code pays off in peformance boost

Results gathered from several articles from my website

- base64 decoding: 2 x faster
- reverse table: 3 x faster
- summing bytes: 4 x faster

But SIMD is faster!

More complex code pays off in peformance boost

Results gathered from several articles from my website

- base64 decoding: 2 x faster
- reverse table: 3 x faster
- summing bytes: 4 x faster
- base64 encoding: 4 x faster

But SIMD is faster!

More complex code pays off in peformance boost

Results gathered from several articles from my website

- base64 decoding: 2 x faster
- reverse table: 3 x faster
- summing bytes: 4 x faster
- base64 encoding: 4 x faster
- population count: 5 x faster

But SIMD is faster!

More complex code pays off in peformance boost

Results gathered from several articles from my website

base64 decoding:	2 x faster
reverse table:	3 x faster
summing bytes:	4 x faster
base64 encoding:	4 x faster
population count:	5 x faster
pixel format conversions:	5 x faster

But SIMD is faster!

More complex code pays off in peformance boost

Results gathered from several articles from my website

- base64 decoding: 2 x faster
- reverse table: 3 x faster
- summing bytes: 4 x faster
- base64 encoding: 4 x faster
- population count: 5 x faster
- pixel format conversions: 5 x faster
- images mixing: 7 x faster

But SIMD is faster!

More complex code pays off in peformance boost

Results gathered from several articles from my website

base64 decoding:	2 x faster
reverse table:	3 x faster
summing bytes:	4 x faster
base64 encoding:	4 x faster
population count:	5 x faster
pixel format conversions:	5 x faster
images mixing:	7 x faster
JPEG zig-zag transformation:	7 x faster

But SIMD is faster!

More complex code pays off in peformance boost

Results gathered from several articles from my website

- base64 decoding: 2 x faster
- reverse table: 3 x faster
- summing bytes: 4 x faster
- base64 encoding: 4 x faster
- population count: 5 x faster
- pixel format conversions: 5 x faster
- images mixing: 7 x faster
- JPEG zig-zag transformation: 7 x faster
- `std::is_sorted`: 8 x faster

But SIMD is faster!

More complex code pays off in peformance boost

Results gathered from several articles from my website

base64 decoding:	2 x faster
reverse table:	3 x faster
summing bytes:	4 x faster
base64 encoding:	4 x faster
population count:	5 x faster
pixel format conversions:	5 x faster
images mixing:	7 x faster
JPEG zig-zag transformation:	7 x faster
`std::is_sorted`:	8 x faster
finding index of min element:	14 x faster

When does SIMD shine?

When does SIMD shine?

- SIMD fits well for arithmetic-insensitive problems

When does SIMD shine?

- SIMD fits well for arithmetic-insensitive problems
- . . . dead simple calculations

When does SIMD shine?

- SIMD fits well for arithmetic-insensitive problems
- . . . dead simple calculations — "simple" is unspecified

When does SIMD shine?

- SIMD fits well for arithmetic-insensitive problems
- ...dead simple calculations — "simple" is unspecified
- ...no `if` / `switch`

When does SIMD shine?

- SIMD fits well for arithmetic-insensitive problems
- ...dead simple calculations — "simple" is unspecified
- ...no `if` / `switch`
- ...no data dependencies between iterations

When does SIMD shine?

- SIMD fits well for arithmetic-insensitive problems
- ...dead simple calculations — "simple" is unspecified
- ...no `if` / `switch`
- ...no data dependencies between iterations
- SIMD requires regular data access

When does SIMD shine?

- SIMD fits well for arithmetic-insensitive problems
- ... dead simple calculations — "simple" is unspecified
- ... no `if` / `switch`
- ... no data dependencies between iterations
- SIMD requires regular data access
- ... preferably sequential access

When does SIMD shine?

- SIMD fits well for arithmetic-insensitive problems
- ...dead simple calculations — "simple" is unspecified
- ...no `if` / `switch`
- ...no data dependencies between iterations
- SIMD requires regular data access
- ...preferably sequential access
- ...no pointers / computed indices

When does SIMD shine?

- SIMD fits well for arithmetic-insensitive problems
- ...dead simple calculations — "simple" is unspecified
- ...no `if` / `switch`
- ...no data dependencies between iterations
- SIMD requires regular data access
- ...preferably sequential access
- ...no pointers / computed indices
- ...no calls to external functions

When does SIMD shine?

- SIMD fits well for arithmetic-insensitive problems
- ...dead simple calculations — "simple" is unspecified
- ...no `if` / `switch`
- ...no data dependencies between iterations
- SIMD requires regular data access
- ...preferably sequential access
- ...no pointers / computed indices
- ...no calls to external functions
- SIMD requires designed data structures to use full power

When does SIMD shine?

- SIMD fits well for arithmetic-insensitive problems
- ...dead simple calculations — "simple" is unspecified
- ...no `if` / `switch`
- ...no data dependencies between iterations
- SIMD requires regular data access
- ...preferably sequential access
- ...no pointers / computed indices
- ...no calls to external functions
- SIMD requires designed data structures to use full power
- ...typical example: instead of a single array of records use separate array for each record's field

More realistic example — signal mixing

Following vector code mixes two signals (image, sound) using linear interpolation

$$c = a \cdot p + b \cdot (1 - p), p \in [0, 1]$$

```
void vector_lerp(float* a, float* b, float* c, size_t N, float p) {
    auto Vp = vector_broadcast(p);      // Vp = [p, ..., p]
    auto Vq = vector_broadcast(1 - p);  // Vq = [1-p, ..., 1-p]
    for (size_t i=0; i < N; i += 8) {
        auto Va  = vector_load(a + i);
        auto Vb  = vector_load(b + i);
        auto Vt0 = vector_mul(Va, Vp); // a * p
        auto Vt1 = vector_mul(Vb, Vq); // b * (1 - p)
        auto Vc  = vector_add(Vt0, Vt1);
        vector_store(c + i, Vc);
    }

    for (size_t i=(N / 8) * 8; i < N; i++)
        c[i] = (a[i] * p) + (b[i] * (1 - p));
}
```

SIMD for programmers

SIMD for programmers

- Currently C/C++ do not directly support SIMD data types

SIMD for programmers

- Currently C/C++ do not directly support SIMD data types
- ... exception is a C extension **ispc** (ispc.github.io)

SIMD for programmers

- Currently C/C++ do not directly support SIMD data types
- … exception is a C extension **ispc** (ispc.github.io)
- … Swift, Rust and C# have builtin support; Java – depends on JIT

SIMD for programmers

- Currently C/C++ do not directly support SIMD data types
- ...exception is a C extension **ispc** (ispc.github.io)
- ...Swift, Rust and C# have builtin support; Java – depends on JIT
- SIMD instructions are available for programmers as so called **intrinsic** functions (in our examples `vector_foo`)

SIMD for programmers

- Currently C/C++ do not directly support SIMD data types
- ... exception is a C extension **ispc** (ispc.github.io)
- ... Swift, Rust and C# have builtin support; Java – depends on JIT
- SIMD instructions are available for programmers as so called **intrinsic** functions (in our examples `vector_foo`)
- Intrinsic functions are translated by compilers into a predefined sequence of CPU instructions, often just one

SIMD for programmers

- Currently C/C++ do not directly support SIMD data types
- ... exception is a C extension **ispc** (ispc.github.io)
- ... Swift, Rust and C# have builtin support; Java – depends on JIT
- SIMD instructions are available for programmers as so called **intrinsic** functions (in our examples `vector_foo`)
- Intrinsic functions are translated by compilers into a predefined sequence of CPU instructions, often just one
- There are attempts to hide multitude of SIMD flavours behind some generic API, may be worth to check out.

SIMD for programmers

- Currently C/C++ do not directly support SIMD data types
- ... exception is a C extension **ispc** (ispc.github.io)
- ... Swift, Rust and C# have builtin support; Java – depends on JIT
- SIMD instructions are available for programmers as so called **intrinsic** functions (in our examples `vector_foo`)
- Intrinsic functions are translated by compilers into a predefined sequence of CPU instructions, often just one
- There are attempts to hide multitude of SIMD flavours behind some generic API, may be worth to check out.
- Compilers can **autovectorize** loops — they do similar transformation as we did to `vector_add`

SIMD for programmers

- Currently C/C++ do not directly support SIMD data types
- ...exception is a C extension **ispc** (ispc.github.io)
- ...Swift, Rust and C# have builtin support; Java – depends on JIT
- SIMD instructions are available for programmers as so called **intrinsic** functions (in our examples `vector_foo`)
- Intrinsic functions are translated by compilers into a predefined sequence of CPU instructions, often just one
- There are attempts to hide multitude of SIMD flavours behind some generic API, may be worth to check out.
- Compilers can **autovectorize** loops — they do similar transformation as we did to `vector_add`
- Autovectorization is not as smart as human, but is decent

Signal mixing in practise

This is actual C++ code for signal mixing which uses Intel intrinsics functions for AVX2 extension (full list on Intrinsics Guide)

```
#include <immintrin.h>

void vector_lerp(float* a, float* b, float* c, size_t N, float p) {
    __m256 Vp = _mm256_set1_ps(p);
    __m256 Vq = _mm256_set1_ps(1 - p);
    for (size_t i=0; i < N; i += 8) {
        __m256 Va  = _mm256_loadu_ps(a + i);
        __m256 Vb  = _mm256_loadu_ps(b + i);
        __m256 Vt0 = _mm256_mul_ps(Va, Vp);
        __m256 Vt1 = _mm256_mul_ps(Vb, Vq);
        __m256 Vc  = _mm256_mul_ps(Vt0, Vt1);
        _mm256_storeu_ps(c + i, Vc);
    }

    for (size_t i=(N / 8) * 8; i < N; i++)
        c[i] = (a[i] * p) + (b[i] * (1 - p));
}
```

Signal mixing in practise

This is actual C++ code for signal mixing which uses Intel intrinsics functions for AVX2 extension (full list on Intrinsics Guide)

```cpp
#include <immintrin.h>

void vector_lerp(float* a, float* b, float* c, size_t N, float p) {
    __m256 Vp = _mm256_set1_ps(p);
    __m256 Vq = _mm256_set1_ps(1 - p);
    for (size_t i=0; i < N; i += 8) {
        __m256 Va  = _mm256_loadu_ps(a + i);
        __m256 Vb  = _mm256_loadu_ps(b + i);
        __m256 Vt0 = _mm256_mul_ps(Va, Vp);
        __m256 Vt1 = _mm256_mul_ps(Vb, Vq);
        __m256 Vc  = _mm256_mul_ps(Vt0, Vt1);
        _mm256_storeu_ps(c + i, Vc);
    }

    for (size_t i=(N / 8) * 8; i < N; i++)
        c[i] = (a[i] * p) + (b[i] * (1 - p));
}
```

It compiles! `gcc -mavx2 -c vector-lerp-avx2.cpp`

Takeaways

Takeaways

- SIMD instructions are ubiquitous

Takeaways

- SIMD instructions are ubiquitous
- SIMD can give significant speedup

Takeaways

- SIMD instructions are ubiquitous
- SIMD can give significant speedup
- . . . but doesn't make a program magically faster

Takeaways

- SIMD instructions are ubiquitous
- SIMD can give significant speedup
- . . . but doesn't make a program magically faster
- SIMD instructions are quite hard to master by programmers

Takeaways

- SIMD instructions are ubiquitous
- SIMD can give significant speedup
- ...but doesn't make a program magically faster
- SIMD instructions are quite hard to master by programmers
- ...our mental model is different

Takeaways

- SIMD instructions are ubiquitous
- SIMD can give significant speedup
- ...but doesn't make a program magically faster
- SIMD instructions are quite hard to master by programmers
- ...our mental model is different
- ...training helps

Takeaways

- SIMD instructions are ubiquitous
- SIMD can give significant speedup
- . . . but doesn't make a program magically faster
- SIMD instructions are quite hard to master by programmers
- . . . our mental model is different
- . . . training helps
- Luckily compilers are getting better in autovectorization

Takeaways

- SIMD instructions are ubiquitous
- SIMD can give significant speedup
- . . . but doesn't make a program magically faster
- SIMD instructions are quite hard to master by programmers
- . . . our mental model is different
- . . . training helps
- Luckily compilers are getting better in autovectorization
- More and more software libraries use SIMD instructions, we can benefit from it without changing our code