Accurate and Efficiently Vectorized Sums and Dot Products in Julia

Julia Paris Meetup

October 17th, 2019

François Févotte – TriScale innov
Chris Elrod – Baylor University

Summation & dot product

Summation & dot product

Algorithm 1 Naive summation

```
$\sigma \leftarrow 0$
for $i = 1 \dots n$ do
    $\sigma \leftarrow \sigma + x_i$
end for
return $\sigma$
```

Algorithm 2 Naive dot product

```
$\sigma \leftarrow 0$
for $i = 1 \dots n$ do
    $\sigma \leftarrow \sigma + x_i\, y_i$
end for
return $\sigma$
```

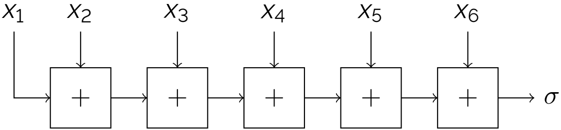
Accuracy

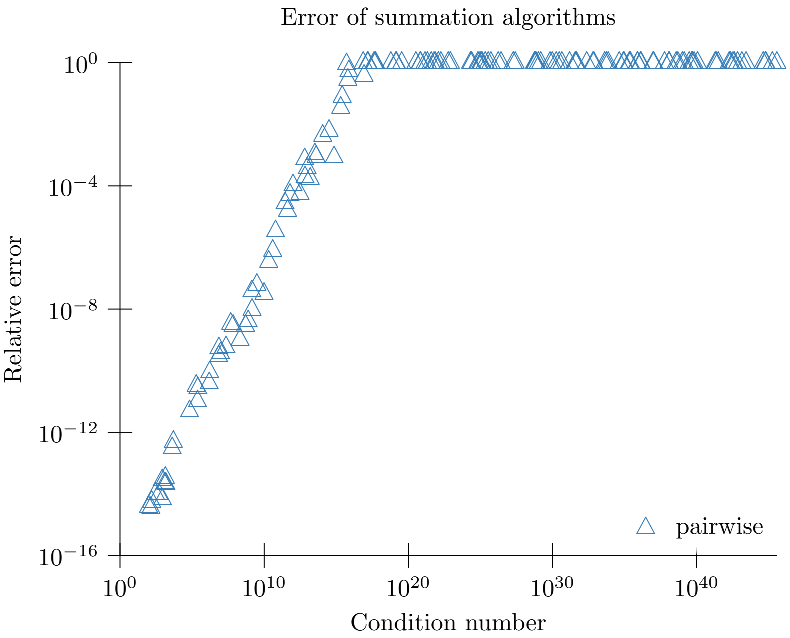
Accuracy of summation algorithms

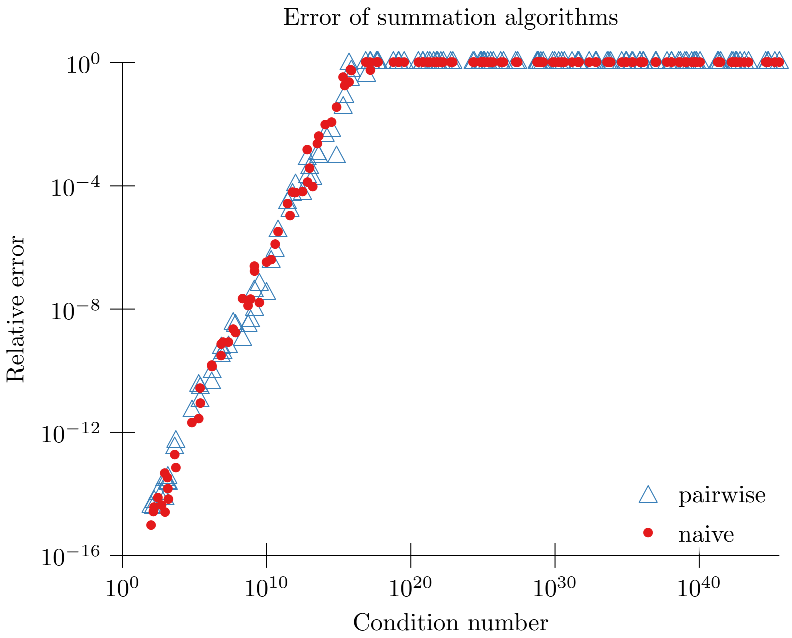
Naive summation

Algorithm 3 Naive summation

```
$\sigma \leftarrow 0$
for $i = 1 \dots n$ do
    $\sigma \leftarrow \sigma + x_i$
end for
return $\sigma$
```




Error of summation algorithms
Relative error
10^{0}
10^{-4}
10^{-8}
10^{-12}
10^{-16}
10^{0}
10^{10}
10^{20}
10^{30}
10^{40}
Condition number
△ pairwise


Error of summation algorithms
Relative error
Condition number
10^{0}
10^{-4}
10^{-8}
10^{-12}
10^{-16}
10^{0}
10^{10}
10^{20}
10^{30}
10^{40}
△ pairwise
● naive

Accuracy of summation algorithms

Compensated algorithms

- For some classical algorithms, there exist "compensated" variants :
  - summation
  - dot product
  - polynomial evaluation
  - ...

- these "compensated algorithms" are based on Error Free Transforms (EFTs) :

$$x \underset{\text{EFT}}{\circ} y = (r, \delta) \qquad \Big(\forall \circ \in \{+, -, \times\}\Big)$$

such that

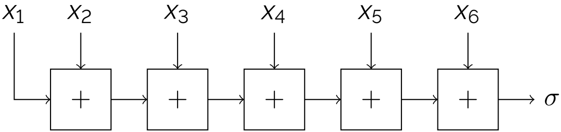
$$r = \lfloor x \circ y \rceil$$
$$r + \delta = x \circ y$$

Accuracy of summation algorithms

Compensated summation

Algorithm 4 Naive summation

```
$\sigma \leftarrow 0$
for $i = 1 \dots n$ do
    $\sigma \leftarrow \sigma + x_i$
end for
return $\sigma$
```

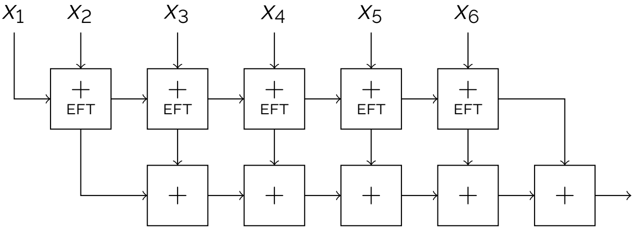


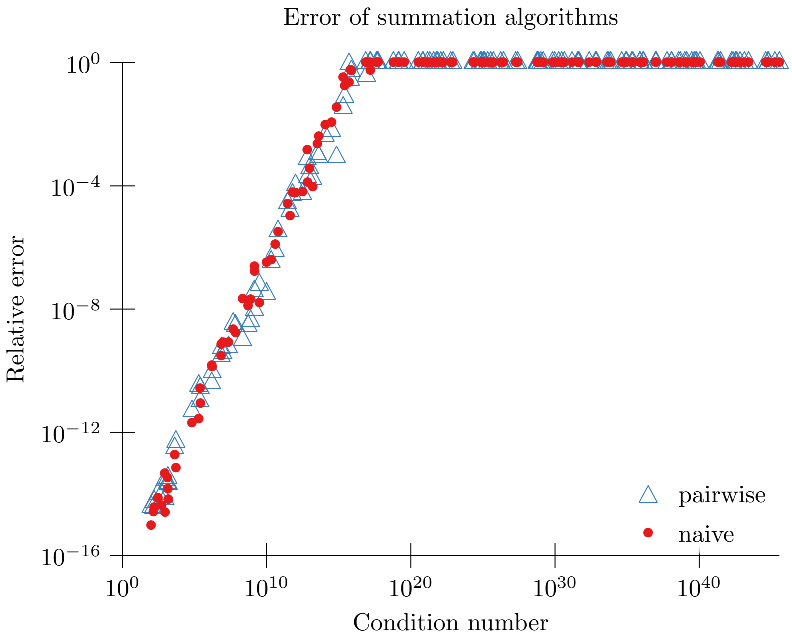
Accuracy of summation algorithms

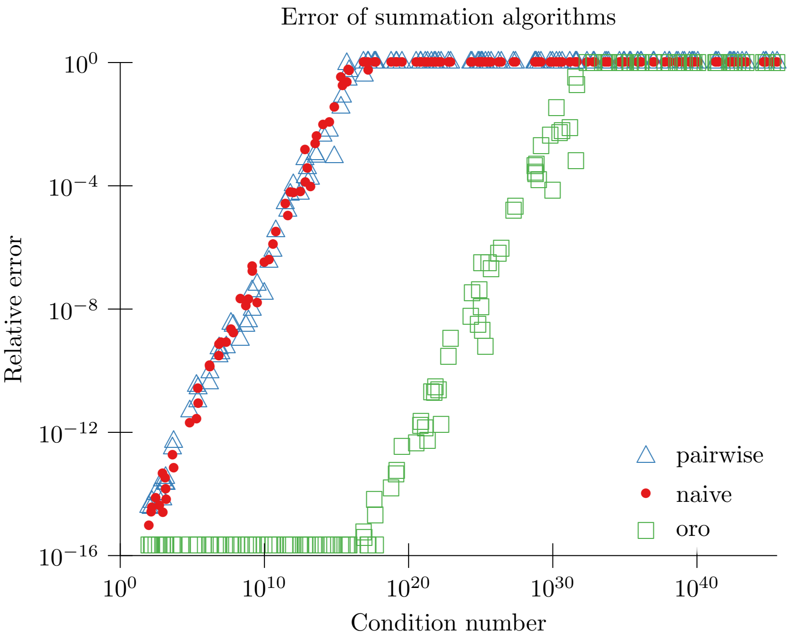
Compensated summation

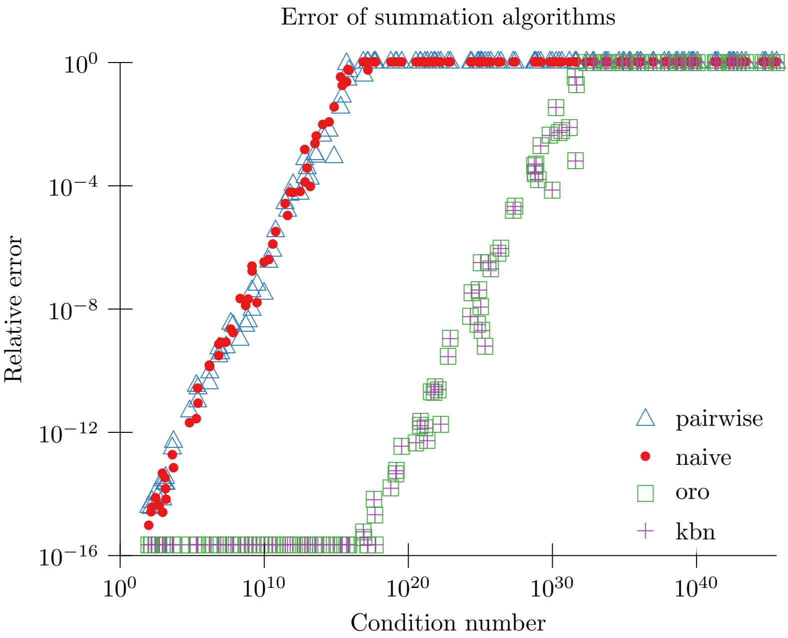
Algorithm 7 Compensated summation

```
$\sigma \leftarrow 0$
$\pi \leftarrow 0$
for $i = 1 \dots n$ do
    $(\sigma, e) \leftarrow \sigma \underset{\text{EFT}}{+} x_i$
    $\pi \leftarrow \pi + e$
end for
return $\sigma + \pi$
```




Error of summation algorithms
Relative error
10^0
10^{-4}
10^{-8}
10^{-12}
10^{-16}
Condition number
10^0
10^{10}
10^{20}
10^{30}
10^{40}
△ pairwise
● naive


Error of summation algorithms
Relative error
10^0
10^{-4}
10^{-8}
10^{-12}
10^{-16}
Condition number
10^0
10^{10}
10^{20}
10^{30}
10^{40}
pairwise
naive
oro


Error of summation algorithms
Relative error
10^{0}
10^{-4}
10^{-8}
10^{-12}
10^{-16}
Condition number
10^{0}
10^{10}
10^{20}
10^{30}
10^{40}
pairwise
naive
oro
kbn

Performance & Vectorization (SIMD)

Performance of summation algorithms

Vectorization (SIMD)

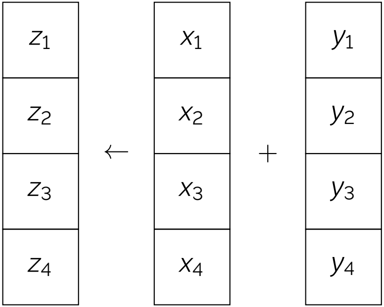
- Scalar operation

$$\boxed{z_1} \leftarrow \boxed{x_1} + \boxed{y_1}$$

Performance of summation algorithms

Vectorization (SIMD)

- Vector operation (Single Instruction Multiple Data)

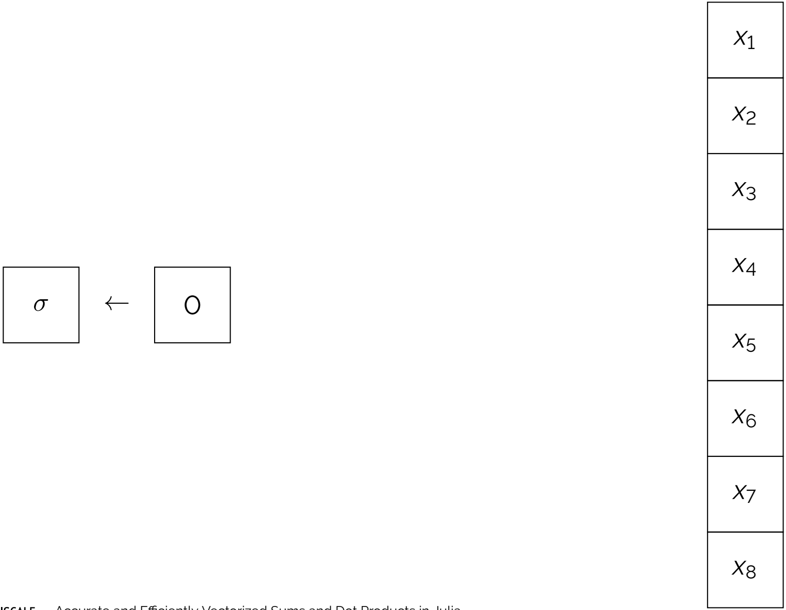


$$\begin{array}{|c|}\hline z_1\\\hline z_2\\\hline z_3\\\hline z_4\\\hline\end{array} \leftarrow \begin{array}{|c|}\hline x_1\\\hline x_2\\\hline x_3\\\hline x_4\\\hline\end{array} + \begin{array}{|c|}\hline y_1\\\hline y_2\\\hline y_3\\\hline y_4\\\hline\end{array}$$

- Hardware generations :
  - SSE
  - AVX
  - AVX-2
  - AVX-512

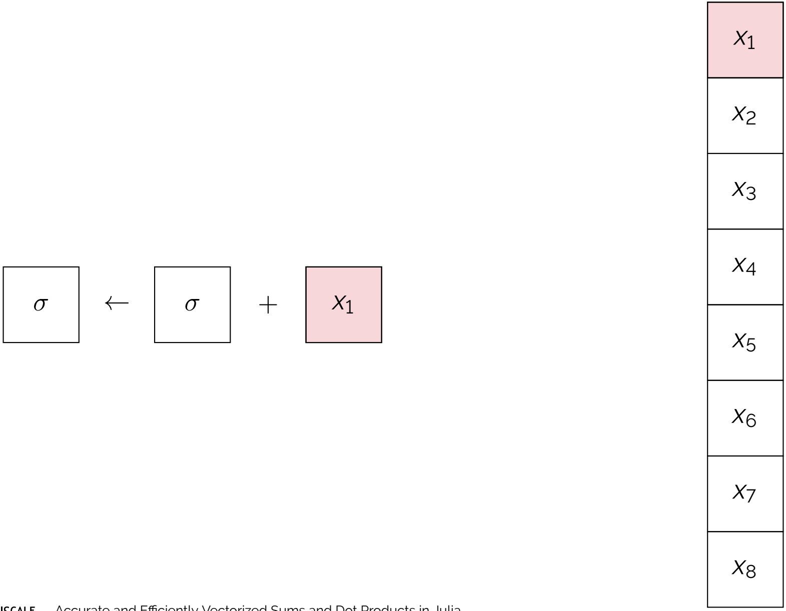
Performance of summation algorithms

Naive summation, scalar version



Performance of summation algorithms

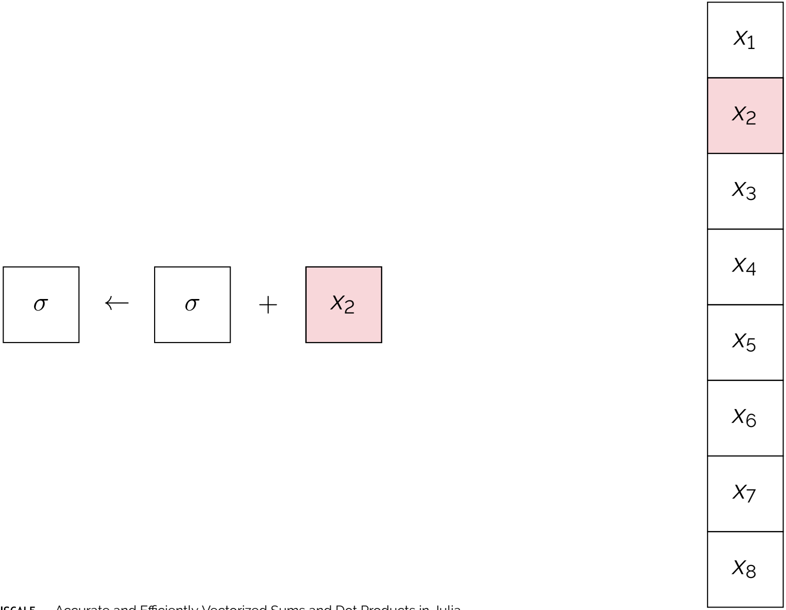
Naive summation, scalar version



$$\sigma \leftarrow \sigma + x_1$$

Performance of summation algorithms

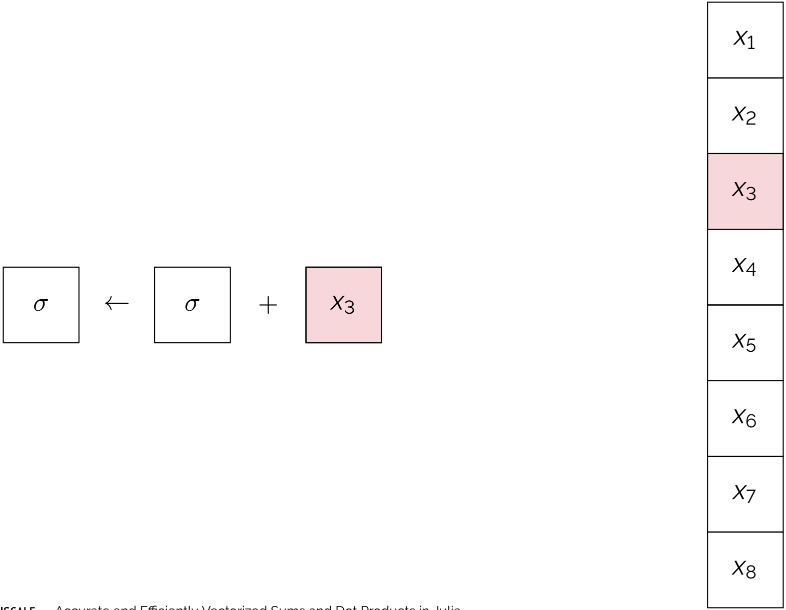
Naive summation, scalar version



$\sigma \leftarrow \sigma + x_2$

Performance of summation algorithms

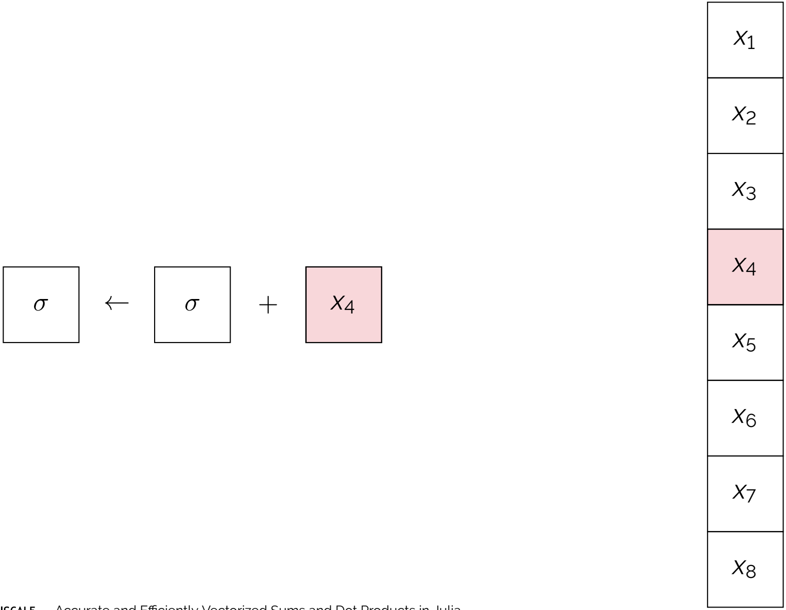
Naive summation, scalar version



$$\sigma \leftarrow \sigma + x_3$$

Performance of summation algorithms

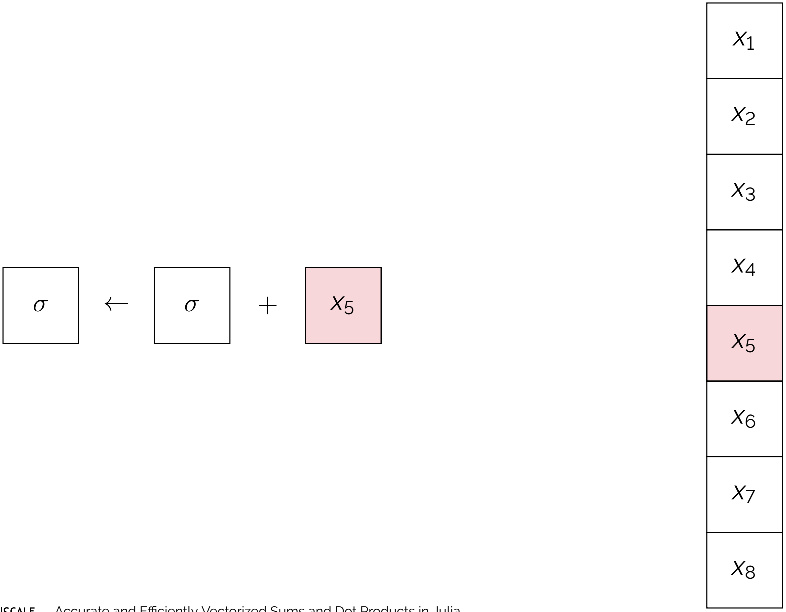
Naive summation, scalar version



$$\sigma \leftarrow \sigma + x_4$$

Performance of summation algorithms

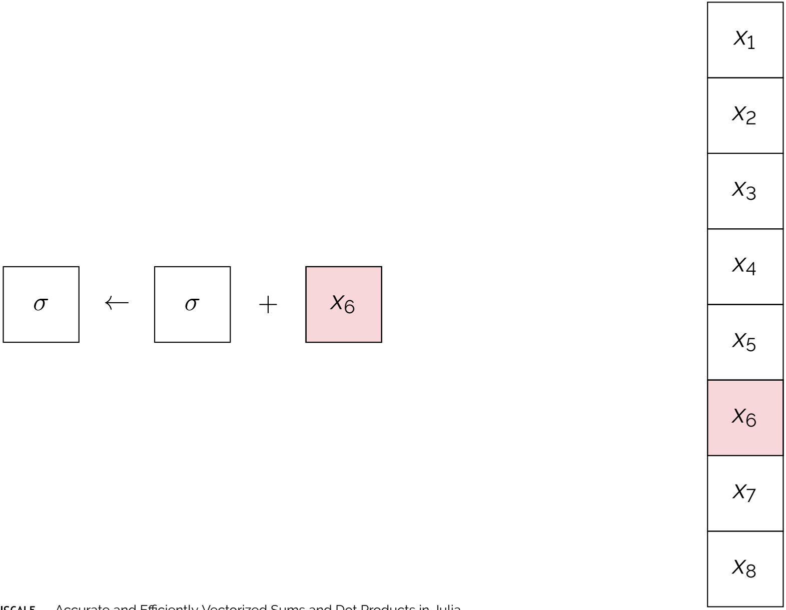
Naive summation, scalar version



$$\sigma \leftarrow \sigma + x_5$$

Performance of summation algorithms

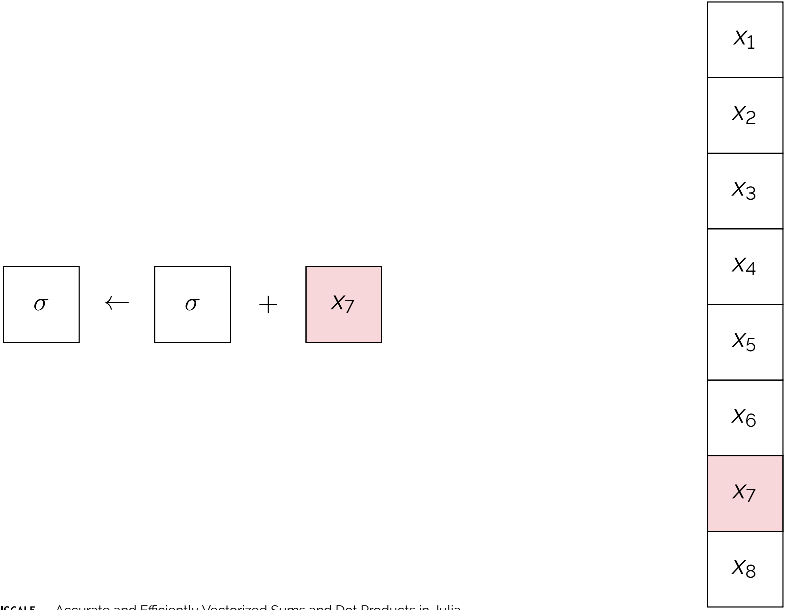
Naive summation, scalar version



$\sigma \leftarrow \sigma + x_6$

Performance of summation algorithms

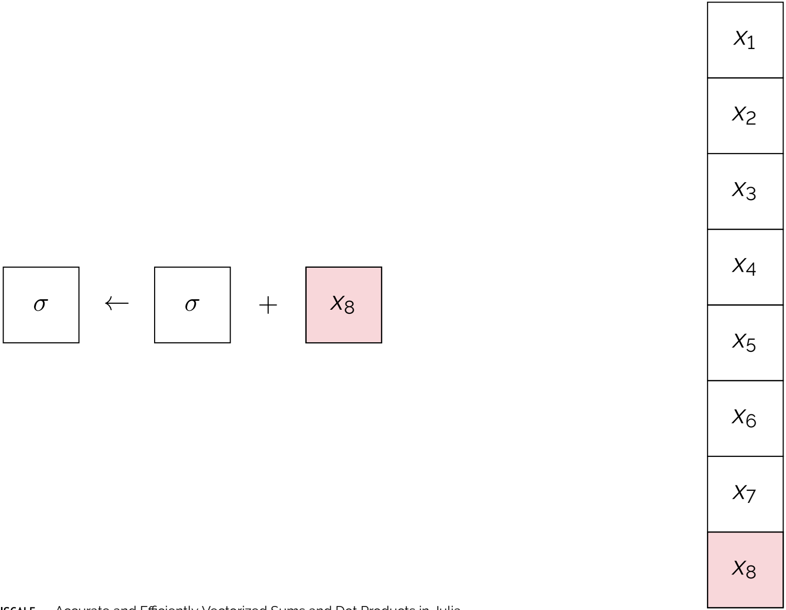
Naive summation, scalar version



$$\boxed{\sigma} \leftarrow \boxed{\sigma} + \boxed{x_7}$$

Performance of summation algorithms

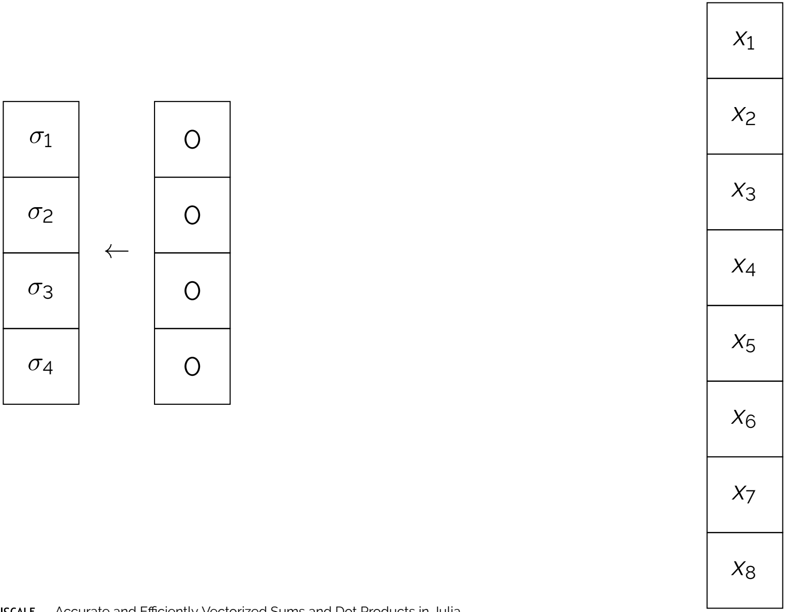
Naive summation, scalar version



$$\sigma \leftarrow \sigma + x_8$$

Performance of summation algorithms

Naive summation, vectorized version

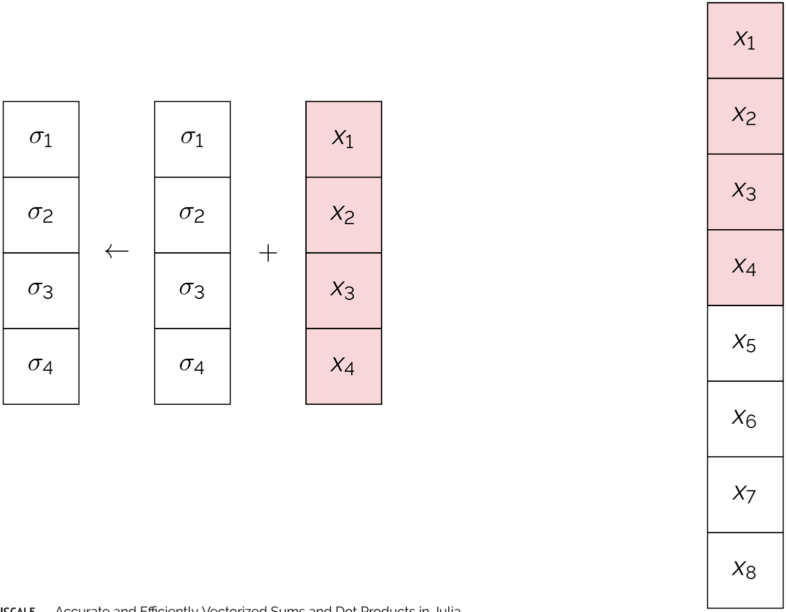


$$\begin{pmatrix}\sigma_1\\ \sigma_2\\ \sigma_3\\ \sigma_4\end{pmatrix} \leftarrow \begin{pmatrix}0\\ 0\\ 0\\ 0\end{pmatrix}$$

$$\begin{pmatrix}x_1\\ x_2\\ x_3\\ x_4\\ x_5\\ x_6\\ x_7\\ x_8\end{pmatrix}$$

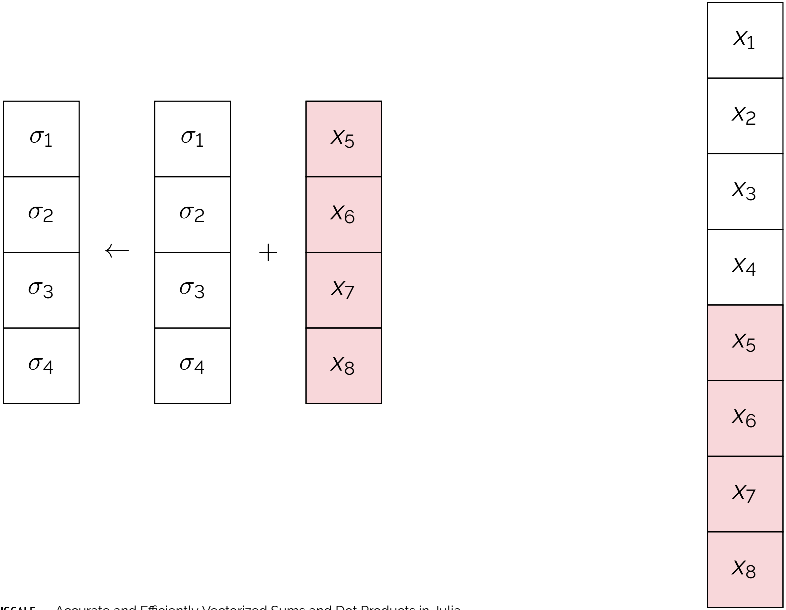
Performance of summation algorithms

Naive summation, vectorized version



Performance of summation algorithms

Naive summation, vectorized version

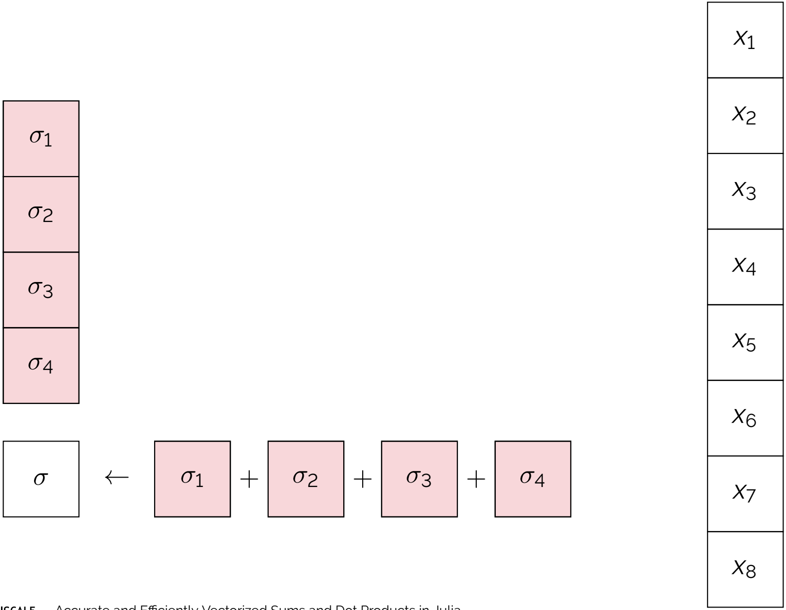


$$\begin{bmatrix}\sigma_1\\ \sigma_2\\ \sigma_3\\ \sigma_4\end{bmatrix} \leftarrow \begin{bmatrix}\sigma_1\\ \sigma_2\\ \sigma_3\\ \sigma_4\end{bmatrix} + \begin{bmatrix}x_5\\ x_6\\ x_7\\ x_8\end{bmatrix}$$

$$\begin{bmatrix}x_1\\ x_2\\ x_3\\ x_4\\ x_5\\ x_6\\ x_7\\ x_8\end{bmatrix}$$

Performance of summation algorithms

Naive summation, vectorized version



$$\boxed{\sigma} \leftarrow \boxed{\sigma_1} + \boxed{\sigma_2} + \boxed{\sigma_3} + \boxed{\sigma_4}$$

Performance of summation algorithms

Vectorized summation

Algorithm 8 Naive summation

```
{Initialization :}
$a \leftarrow 0$

{Loop on vector elements :}
for $i \in 1 : N$ do
    $a \leftarrow a \oplus x_i$
end for

return $a$
```

- Naive

variant	ns/el.	speedup
scalar	1.35	

Performance of summation algorithms

Vectorized summation

Algorithm 12 Vectorized summation

```
{Initialization :}
$\mathbf{a} \leftarrow 0$

{Loop on full packs :}
for $j \in 1 : \lfloor \frac{N}{W} \rfloor$ do
    $i \leftarrow W(j-1)+1$
    $\mathbf{p} \leftarrow (x_i, x_{i+1}, x_{i+2}, x_{i+3})$
    $\mathbf{a} \leftarrow \mathbf{a} \oplus \mathbf{p}$
end for

{Reduction of SIMD accumulator :}
$a \leftarrow \mathtt{vsum}(\mathbf{a})$

{Loop on remaining elements :}
for $j \in W \lfloor \frac{N}{W} \rfloor + 1 : N$ do
    $a \leftarrow a \oplus x_i$
end for

return $a$
```

- Naive

variant	ns/el.	speedup
scalar	1.35	
vector	0.17	×8

Performance of summation algorithms

Vectorized summation

Algorithm 16 Unrolled vectorized summation

```
{Initialization :}
$\mathbf{a}_1 \leftarrow 0$
$\mathbf{a}_2 \leftarrow 0$

{Loop on full packs, unrolled twice :}
for $j \in 1 : \lfloor \frac{N}{2W} \rfloor$ do
    $i_1 \leftarrow 2\,W(j-1)+1$
    $\mathbf{p}_1 \leftarrow (x_{i_1}, x_{i_1+1}, x_{i_1+2}, x_{i_1+3})$
    $\mathbf{a}_1 \leftarrow \mathbf{a}_1 \oplus \mathbf{p}_1$

    $i_2 \leftarrow 2\,W(j-1)+W+1$
    $\mathbf{p}_2 \leftarrow (x_{i_2}, x_{i_2+1}, x_{i_2+2}, x_{i_2+3})$
    $\mathbf{a}_2 \leftarrow \mathbf{a}_2 \oplus \mathbf{p}_2$
end for

{Loop on remaining full packs :}
for $j \in 2\lfloor \frac{N}{2W} \rfloor + 1 : \lfloor \frac{N}{W} \rfloor$ do
    $i \leftarrow W(j-1)+1$
    $\mathbf{p} \leftarrow (x_i, x_{i+1}, x_{i+2}, x_{i+3})$
    $\mathbf{a}_1 \leftarrow \mathbf{a}_1 \oplus \mathbf{p}$
end for

{Reduction of SIMD accumulators :}
$\mathbf{a}_1 \leftarrow \mathbf{a}_1 \oplus \mathbf{a}_2$
$a \leftarrow \mathtt{vsum}(\mathbf{a}_1)$

{Loop on remaining elements :}
for $j \in W\lfloor \frac{N}{W} \rfloor + 1 : N$ do
    $a \leftarrow a \oplus x_i$
end for

return $a$
```

- Naive

variant	ns/el.	speedup
scalar	1.35	
vector	0.17	×8

- Unrolled

variant	ns/el.	speedup
scalar	0.35	
vector	0.09	×3.8

Implementation

`AccurateArithmetic.jl`

Implementation

Why Julia ?

Algorithm

- Summation
- Dot product

Compensation

- Naive
- Compensated (ORO)
- Compensated (KBN)

Vectorization

- Scalar
- Vectorized
- Vectorized + unrolled

Julia implementation

Why Julia? Separation of concerns : multiple dispatch, splatting...

```
function sum(x)
  acc = zero(eltype(x))
  for e in x
    acc += e
  end
  return acc
end
```

Julia implementation

Why Julia ? Separation of concerns : multiple dispatch, splatting…

```
sum(x) = sum_(x, NaiveAcc)


function sum_(x, accType)
  acc = zero(accType)
  for e in x
    add!(acc, e)
  end
  return value(acc)
end
```

Julia implementation

```
sum(x, y) = acc_((x,),  NaiveSum)
dot(x, y) = acc_((x,y), NaiveDot)

function acc_(operands, accType)
  acc = zero(accType)
  for e in zip(operands...)
    add!(acc, e)
  end
  return value(acc)
end
```

Implementation

Why Julia ?

Algorithm

- Summation
- Dot product

Compensation

- Naive
- Compensated (ORO)
- Compensated (KBN)

Vectorization

- Scalar
- Vectorized
- Vectorized + unrolled

```
sum1(x) = acc_((x,), NaiveSum)
```

Implementation

Why Julia ?

Algorithm

- Summation
- Dot product

Compensation

- Naive
- Compensated (ORO)
- Compensated (KBN)

Vectorization

- Scalar
- Vectorized
- Vectorized + unrolled

```
sum2(x) = acc_((x,), CompSum{two_sum})
```

Implementation

Why Julia ?

Algorithm

- Summation
- Dot product

Compensation

- Naive
- Compensated (ORO)
- Compensated (KBN)

Vectorization

- Scalar
- Vectorized
- Vectorized + unrolled

```
dot1(x, y) = \
    acc_((x, y), CompDot{two_sum})
```

Implementation

Why Julia ?

Algorithm

- Summation
- Dot product

Compensation

- Naive
- Compensated (ORO)
- Compensated (KBN)

Vectorization

- Scalar
- Vectorized
- Vectorized + unrolled

```
dot2(x, y) = \
    acc_vec((x, y), CompDot{two_sum}, Val(3))
```

Julia implementation

Algorithm 17 `two_sum` error-free transform

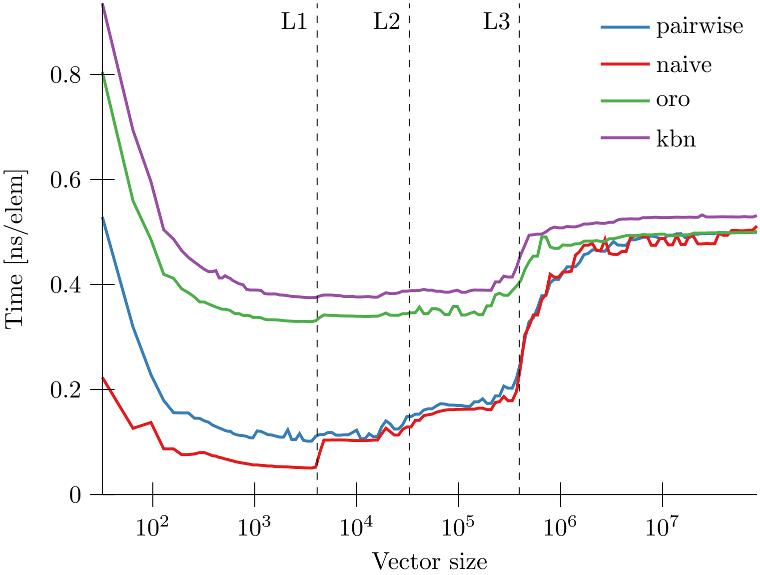
```
Require: $(a, b) \in \mathbb{F}^2$
Ensure: $x = a \oplus b$ and $x + e = a + b$
  $x \leftarrow a \oplus b$
  $y \leftarrow x \ominus a$
  $e \leftarrow (a \ominus (x \ominus y)) \oplus (b \ominus y)$
```

```julia
function two_sum(a::T, b::T) where {T}
    SIMDops.@explicit

    x = a + b
    y = x - a
    e = (a - (x - y)) + (b - y)
    return x, e
end
```

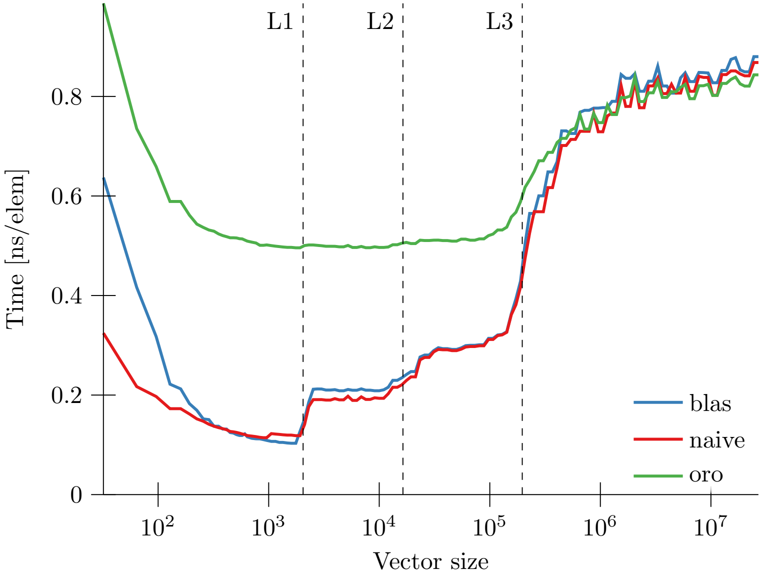
Julia implementation

Why Julia ? Because it's fast !



Julia implementation

Why Julia? Because it's fast!



Conclusions

- Julia solves the 2-language problem :
    - (relatively) easy to implement (complex) algorithms
    - good performance (on par with OpenBLAS)

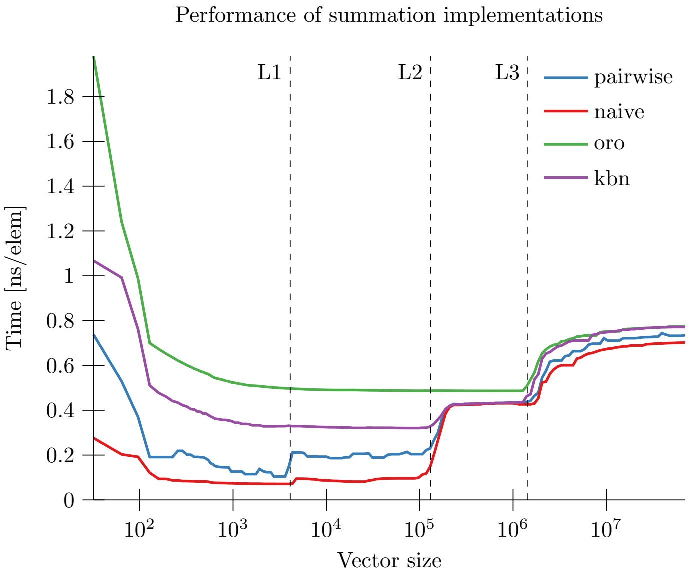
- Browse the sources of AccurateArithmetic.jl to see how this is really done
    - `www.github.com/JuliaMath/AccurateArithmetic.jl`
    - `hal.archives-ouvertes.fr/hal-02265534`

Thanks !

Questions ?

Julia implementation

Why Julia ? Because it's fast !



Julia implementation

Why Julia ? Because it's fast !