

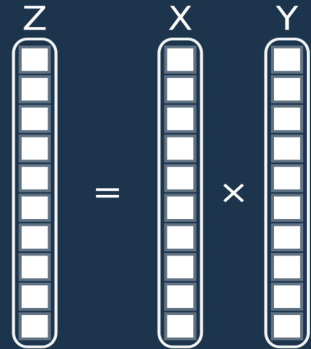
Introduction :

C++ Algorithms

Pierre Aubert

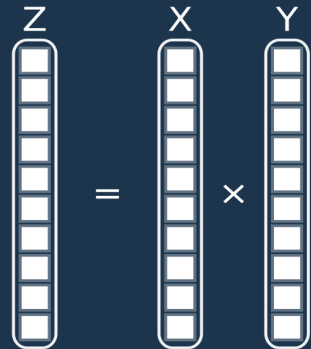


Example : Hadamard Product



Example : Hadamard Product

Element Wise
Operation

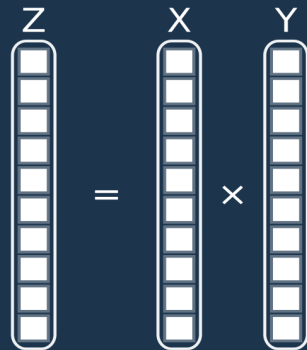


Example : Hadamard Product

C++

```
for(long unsigned int i(0lu); i < nbElement; ++i){  
    »    tabResult[i] = tabX[i]*tabY[i];  
}
```

Element Wise
Operation



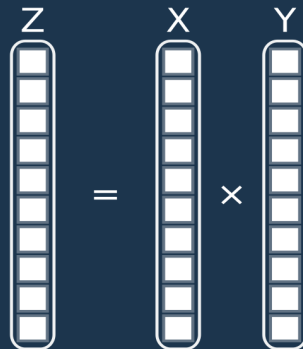
Example : Hadamard Product

C++

```
for(long unsigned int i(0lu); i < nbElement; ++i){
    > tabResult[i] = tabX[i]*tabY[i];
}
```

Explicit order
not necessary

Element Wise
Operation



Example : Hadamard Product

C++

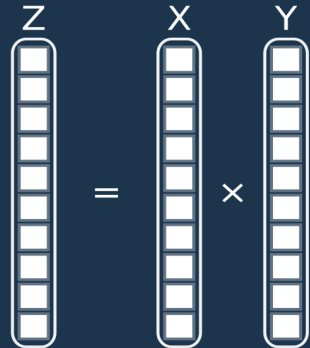
```
for(long unsigned int i(0lu); i < nbElement; ++i){
    > tabResult[i] = tabX[i]*tabY[i];
}
```

Explicit order
not necessary

Element Wise
Operation

C++ Algorithm : `std::transform`

```
std::transform(std::begin(tabX), std::end(tabX),
    > std::begin(tabY), std::begin(tabRes),
    > [](float xi, float yi){ return xi * yi; });
```



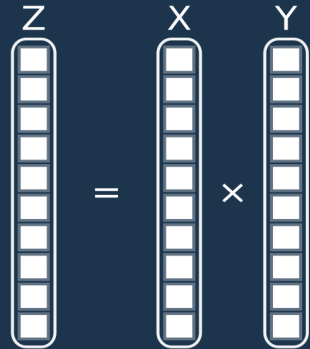
Example : Hadamard Product

C++

```
for(long unsigned int i(0lu); i < nbElement; ++i){
    >     tabResult[i] = tabX[i]*tabY[i];
}
```

Explicit order
not necessary

Element Wise
Operation



C++ Algorithm : `std::transform`

```
std::transform(std::begin(tabX), std::end(tabX),
    >     std::begin(tabY), std::begin(tabRes),
    >     [](float xi, float yi){ return xi * yi; });
```

C++ 17 / C++ 20

```
std::transform(std::execution::par_unseq,
    >     std::begin(tabX), std::end(tabX),
    >     std::begin(tabY), std::begin(tabRes),
    >     [](float xi, float yi){ return xi * yi; });
```

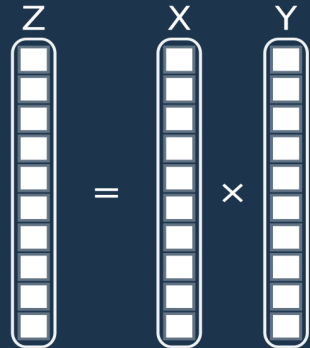
Example : Hadamard Product

C++

```
for(long unsigned int i(0lu); i < nbElement; ++i){
    >     tabResult[i] = tabX[i]*tabY[i];
}
```

Explicit order
not necessary

Element Wise
Operation



C++ Algorithm : `std::transform`

```
std::transform(std::begin(tabX), std::end(tabX),
    >     std::begin(tabY), std::begin(tabRes),
    >     [](float xi, float yi){ return xi * yi; });
```

C++ 17 / C++ 20 Execution Policy

```
std::transform(std::execution::par_unseq
    >     std::begin(tabX), std::end(tabX),
    >     std::begin(tabY), std::begin(tabRes),
    >     [](float xi, float yi){ return xi * yi; });
```

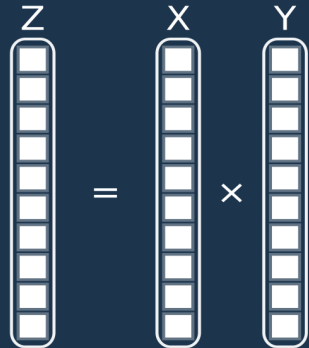
Example : Hadamard Product

C++

```
for(long unsigned int i(0lu); i < nbElement; ++i){
    > tabResult[i] = tabX[i]*tabY[i];
}
```

Explicit order
not necessary

Element Wise
Operation



C++ Algorithm : `std::transform`

```
std::transform(std::begin(tabX), std::end(tabX),
    > std::begin(tabY), std::begin(tabRes),
    > [](float xi, float yi){ return xi * yi; });
```

C++ 17 / C++ 20

Execution Policy

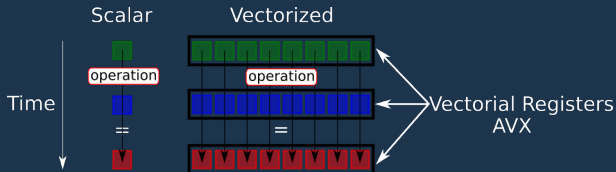
- seq
- unseq
- par
- par_unseq

```
std::transform(std::execution::par_unseq,
    > std::begin(tabX), std::end(tabX),
    > std::begin(tabY), std::begin(tabRes),
    > [](float xi, float yi){ return xi * yi; });
```

What is vectorization ?

The idea is to compute several elements at the same time.

Architecture	Instruction Set	CPU	Nb float Computed at the same time
SSE4	2006	2007	4
AVX	2008	2011	8
AVX 512	2013	2016	16



LINUX : `cat /proc/cpuinfo | grep avx` MAC : `sysctl -a | grep machdep.cpu | grep AVX`

What is vectorization ?

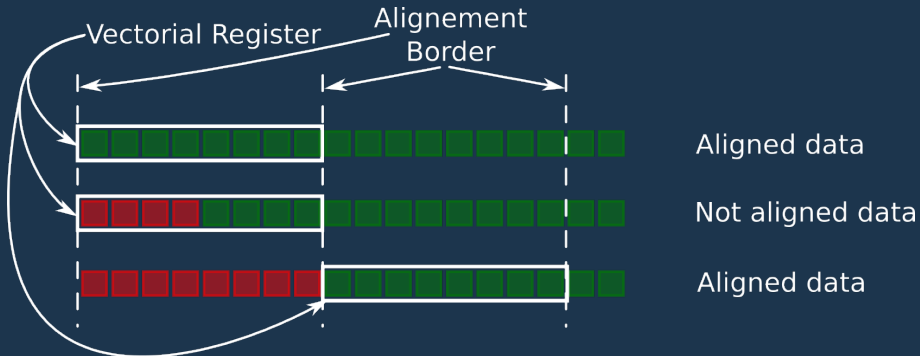
CPU has to read several elements at the same time.

- ▶ Data **contiguosness** :
 - ▶ All data to be used have to be adjacent with others.
 - ▶ Always the case with pointers but be careful with your applications.



What is vectorization ?

- ▶ Data alignment :
 - ▶ All data have to be aligned on vectorial registers size.
 - ▶ Change **new** or **malloc** to **memalign**, **posix_memalign**, or **std::aligned_malloc** since **C++17**



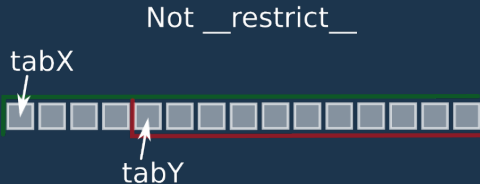
What do we have to do with the code ?

- ▶ The `__restrict__` keyword :
 - ▶ Specify to the compiler there is no overhead between pointers

```
float* tabResult,  
const float* tabX,  
const float* tabY,
```

⇒

```
float* __restrict__ tabResult,  
const float* __restrict__ tabX,  
const float* __restrict__ tabY,
```



What do we have to do with the code ?

- ▶ The `__builtin_assume_aligned` function (or `std::assume_aligned` since C++17) :
 - ▶ Specify to the compiler pointers are aligned
 - ▶ If this is not true, you will get a **Segmentation Fault**.
 - ▶ Here **VECTOR_ALIGNMENT** = 32 (for `float` in **AVX** or **AVX2** extensions).

```
const float* tabX = (const float*)__builtin_assume_aligned(ptabX, VECTOR_ALIGNMENT);
const float* tabY = (const float*)__builtin_assume_aligned(ptabY, VECTOR_ALIGNMENT);
float* tabResult = (float*)__builtin_assume_aligned(ptabResult, VECTOR_ALIGNMENT);
```

Definition in the `CMakeLists.txt` :

```
set(VECTOR_ALIGNMENT 32)
add_definitions(-DVECTOR_ALIGNMENT=${VECTOR_ALIGNMENT})
```

- ▶ The Compilation Options become :
 - ▶ **-O3 -ftree-vectorize -march=native -mtune=native -mavx2**
- ▶ **-ftree-vectorize**
 - ▶ Activate the vectorization
- ▶ **-march=native**
 - ▶ Target only the host CPU architecture for binary
- ▶ **-mtune=native**
 - ▶ Target only the host CPU architecture for optimization
- ▶ **-mavx2**
 - ▶ Vectorize with AVX2 extention (not needed with **g++ 11** or **clang++ 14**)

Modifications Summary

- ▶ Data alignment :
 - ▶ All the data to be aligned on vectorial registers size.
 - ▶ Change **new** or **malloc** to **memalign** or **posix_memalign**

You can use **asterics_malloc** to have LINUX/MAC compatibility (in **evaluateHadamardProduct**):

```
(float*)asterics_malloc(sizeof(float)*nbElement);
```

The **__restrict__** keyword (arguments of **hadamard_product** function):

```
float* __restrict__ tabResult,  
const float* __restrict__ tabX,  
const float* __restrict__ tabY,
```

The **__builtin_assume_aligned** function call (in **hadamard_product** function):

```
const float* tabX = (const float*)__builtin_assume_aligned(ptabX, VECTOR_ALIGNMENT);  
const float* tabY = (const float*)__builtin_assume_aligned(ptabY, VECTOR_ALIGNMENT);  
float* tabResult = (float*)__builtin_assume_aligned(ptabResult, VECTOR_ALIGNMENT);
```

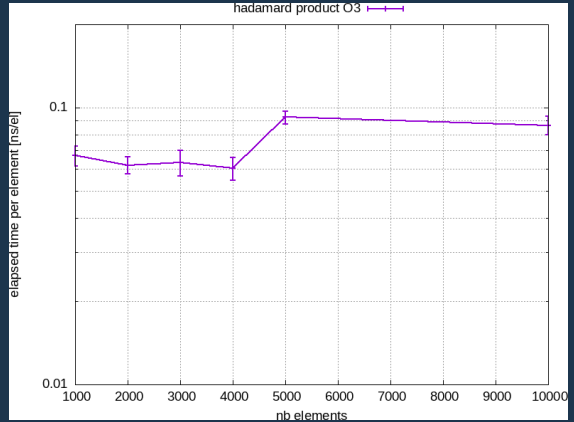
- ▶ The Compilation Options become :
 - ▶ **-O3 -ftree-vectorize -march=native -mtune=native -mavx2**

```
void hadamard_product(float* __restrict__ ptabResult, const float* __restrict__ ptabX, const float* __restrict__ ptabY, long unsigned int nbElement){
    const float* tabX = (const float*)__builtin_assume_aligned(ptabX, VECTOR_ALIGNMENT);
    const float* tabY = (const float*)__builtin_assume_aligned(ptabY, VECTOR_ALIGNMENT);
    float* tabResult = (float*)__builtin_assume_aligned(ptabResult, VECTOR_ALIGNMENT);

    for(long unsigned int i(0lu); i < nbElement; ++i){
        tabResult[i] = tabX[i]*tabY[i];
    }
}
```

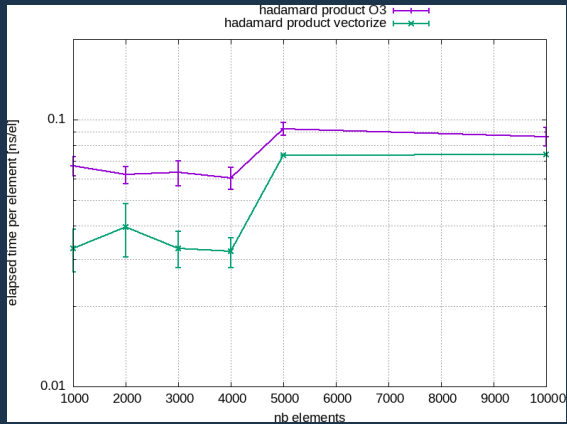
Performance : unseq ?

G++ 11



Performance : unseq ?

G++ 11

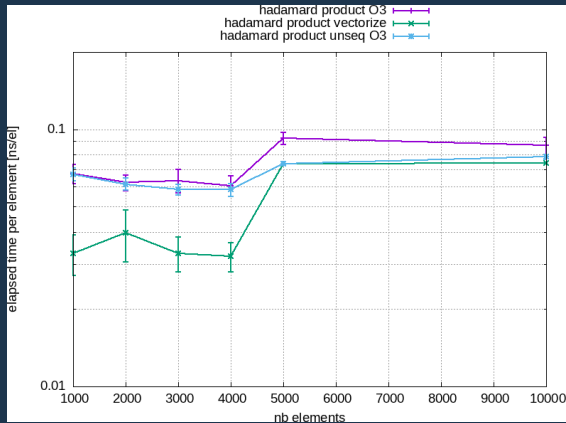


Performance : unseq ?

`std::execution::unseq`

- ▶ Only `-O3` : some improvement

G++ 11

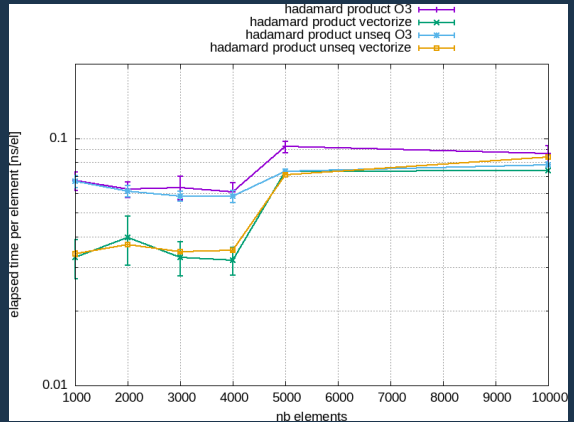


Performance : unseq ?

std::execution::unseq

- ▶ Only **-O3** : some improvement
- ▶ Add **-ftree-vectorize -march=native -mtune=native (-mavx2)**

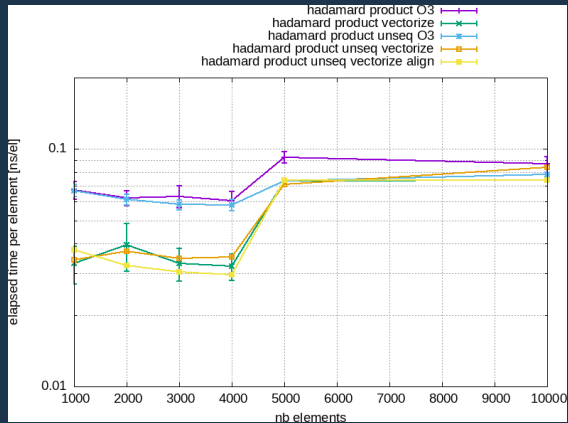
G++ 11



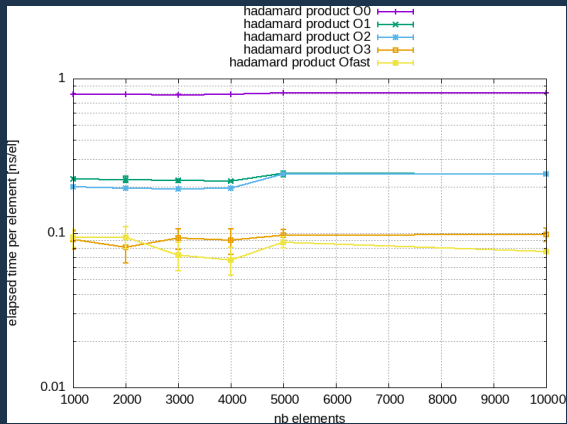
std::execution::unseq

- ▶ Only **-O3** : some improvement
- ▶ Add **-ftree-vectorize -march=native -mtune=native (-mavx2)**
- ▶ On **aligned** data : very efficient

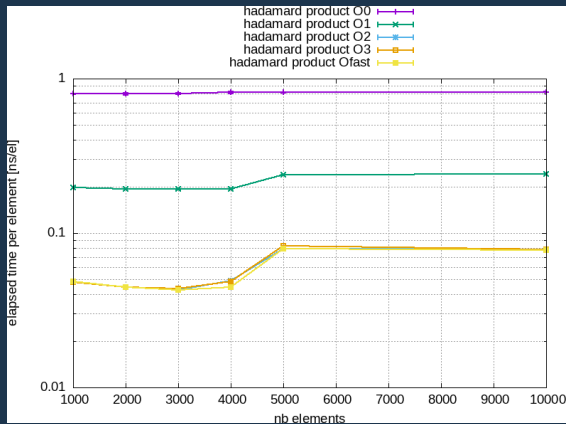
G++ 11



G++ 11

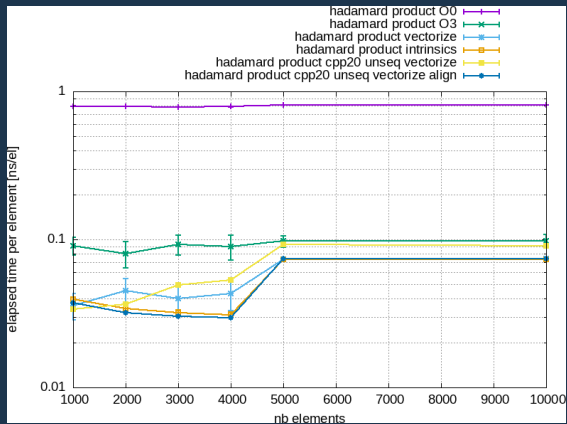


CLang++ 14

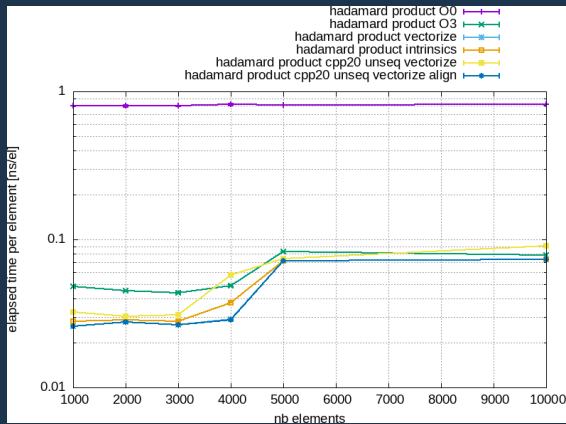


The Hadamard product : Vectorization

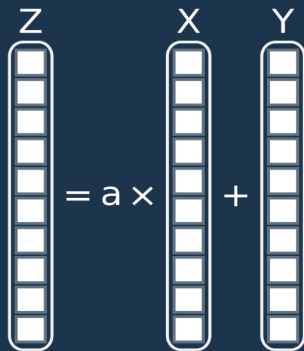
G++ 11



CLang++ 14

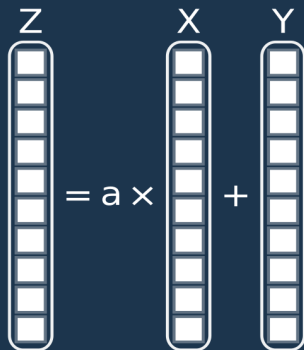


Example : Saxpy



Example : Saxpy

Element Wise
Operation

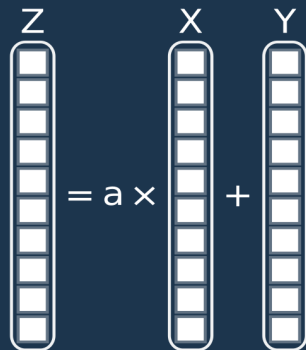


Example : Saxpy

C++

```
for(long unsigned int i(0lu); i < nbElement; ++i){  
    »   tabResult[i] = a*tabX[i] + tabY[i];  
}
```

Element Wise
Operation



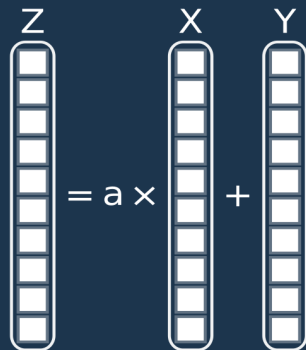
Example : Saxpy

C++

```
for(long unsigned int i(0lu); i < nbElement; ++i){
    »   tabResult[i] = a*tabX[i] + tabY[i];
}
```

Explicit order
not necessary

Element Wise
Operation



Example : Saxpy

C++

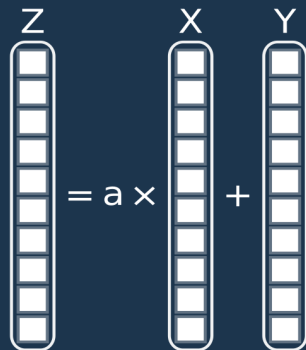
```
for(long unsigned int i(0lu); i < nbElement; ++i){
    >> tabResult[i] = a*tabX[i] + tabY[i];
}
```

Explicit order
not necessary

Element Wise
Operation

C++ Algorithm : `std::transform`

```
std::transform(std::begin(tabX), std::end(tabX),
    >> std::begin(tabY), std::begin(tabResult),
    >> [=](float xi, float yi){ return a*xi + yi; });
```



Example : Saxpy

C++

```
for(long unsigned int i(0lu); i < nbElement; ++i){
    » tabResult[i] = a*tabX[i] + tabY[i];
}
```

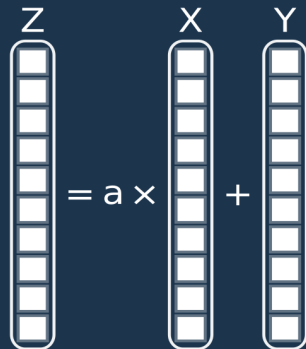
Explicit order
not necessary

C++ Algorithm : `std::transform`

```
std::transform(std::begin(tabX), std::end(tabX),
    » std::begin(tabY), std::begin(tabResult),
    » [=](float xi, float yi){ return a*xi + yi; });
```

Catches Extra
variables by copy

Element Wise
Operation



Example : Saxpy

C++

```
for(long unsigned int i(0lu); i < nbElement; ++i){
    » tabResult[i] = a*tabX[i] + tabY[i];
}
```

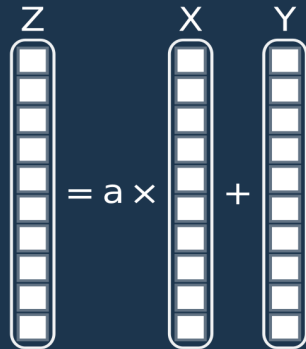
Explicit order
not necessary

C++ Algorithm : `std::transform`

```
std::transform(std::begin(tabX), std::end(tabX),
    » std::begin(tabY), std::begin(tabResult),
    » [=](float xi, float yi){ return a*xi + yi; });
```

Catches Extra
variables by copy

Element Wise
Operation



C++ 17 / C++ 20

```
std::transform(std::execution::par_unseq,
    » std::begin(tabX), std::end(tabX),
    » std::begin(tabY), std::begin(tabResult),
    » [=](float xi, float yi){ return a*xi + yi; });
```

Example : Saxpy

C++

```
for(long unsigned int i(0lu); i < nbElement; ++i){
    » tabResult[i] = a*tabX[i] + tabY[i];
}
```

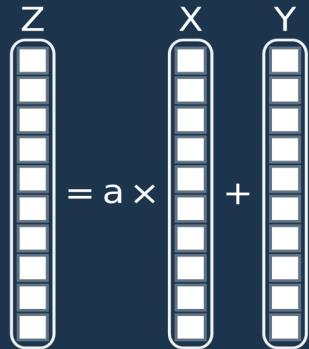
Explicit order
not necessary

C++ Algorithm : `std::transform`

```
std::transform(std::begin(tabX), std::end(tabX),
    » std::begin(tabY), std::begin(tabResult),
    » [=](float xi, float yi){ return a*xi + yi; });
```

Catches Extra
variables by copy

Element Wise
Operation



C++ 17 / C++ 20

Execution Policy

```
std::transform(std::execution::par_unseq,
    » std::begin(tabX), std::end(tabX),
    » std::begin(tabY), std::begin(tabResult),
    » [=](float xi, float yi){ return a*xi + yi; });
```

Example : Saxpy

C++

```
for(long unsigned int i(0lu); i < nbElement; ++i){
    » tabResult[i] = a*tabX[i] + tabY[i];
}
```

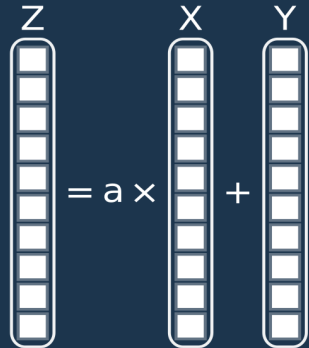
Explicit order
not necessary

C++ Algorithm : `std::transform`

```
std::transform(std::begin(tabX), std::end(tabX),
    » std::begin(tabY), std::begin(tabResult),
    » [=](float xi, float yi){ return a*xi + yi; });
```

Catches Extra
variables by copy

Element Wise
Operation



C++ 17 / C++ 20

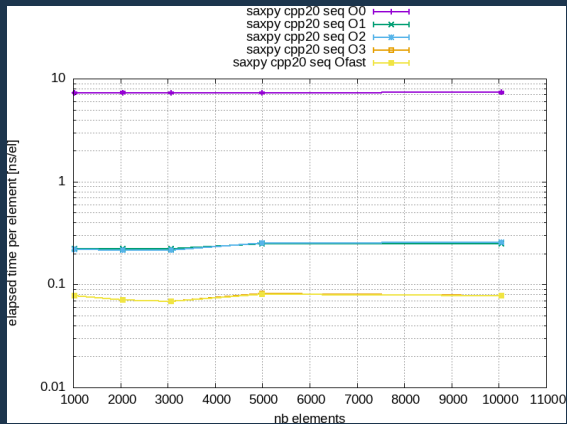
Execution Policy

```
std::transform(std::execution::par_unseq,
    » std::begin(tabX), std::end(tabX),
    » std::begin(tabY), std::begin(tabResult),
    » [=](float xi, float yi){ return a*xi + yi; });
```

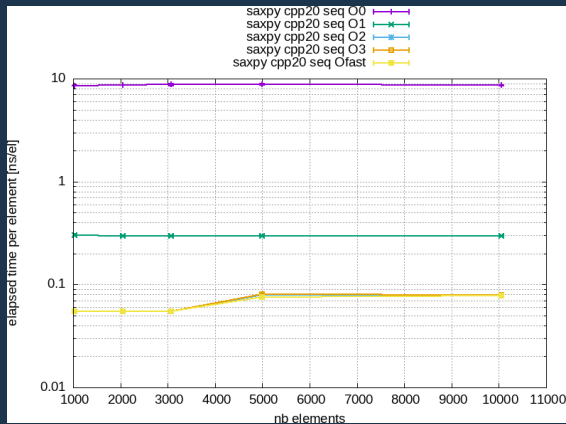
- seq
- unseq
- par
- par_unseq

Saxpy : Basic Options

G++ 11

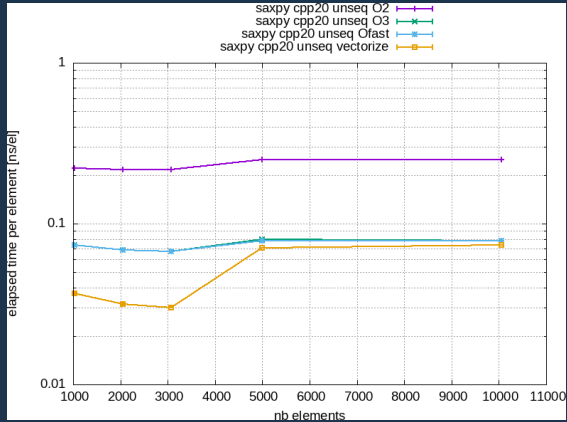


CLang++ 14

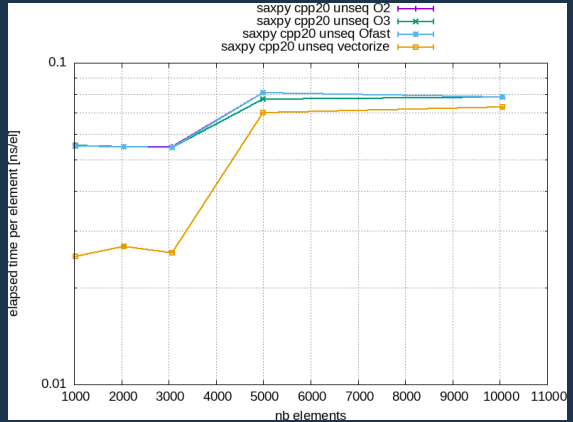


Saxpy : Vectorization

G++ 11



CLang++ 14



Example : Reduction



Example : Reduction

C++

```
float res(0.0f);  
for(long unsigned int i(0lu); i < nbElement; ++i){  
    >> res += tabValue[i];  
}  
return res;
```



Example : Reduction

C++

```
float res(0.0f);
for(long unsigned int i(0lu); i < nbElement; ++i){
    >>     res += tabValue[i];
}
return res;
```

Explicit order
not necessary
every time



Example : Reduction

C++

```
float res(0.0f);
for(long unsigned int i(0lu); i < nbElement; ++i){
    >>     res += tabValue[i];
}
return res;
```

Explicit order
not necessary
every time

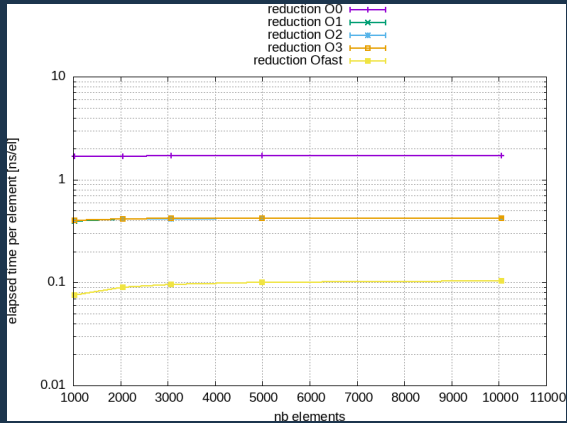
C++ 17 / C++ 20

```
return std::reduce(std::execution::par_unseq,
    >>     std::begin(vecX), std::end(vecX),
    >>     0.0f, std::plus{});
```

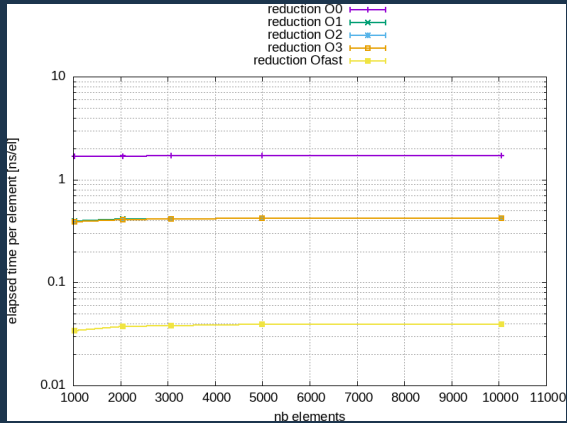


Reduction : Basic Options

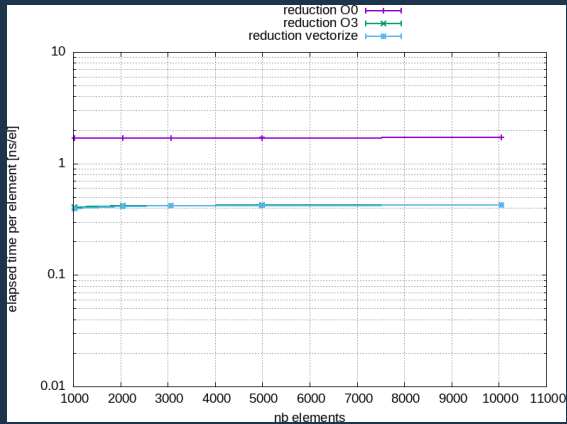
G++ 11



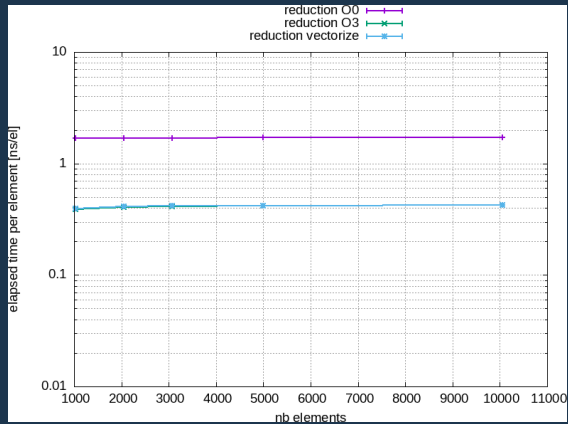
CLang++ 14



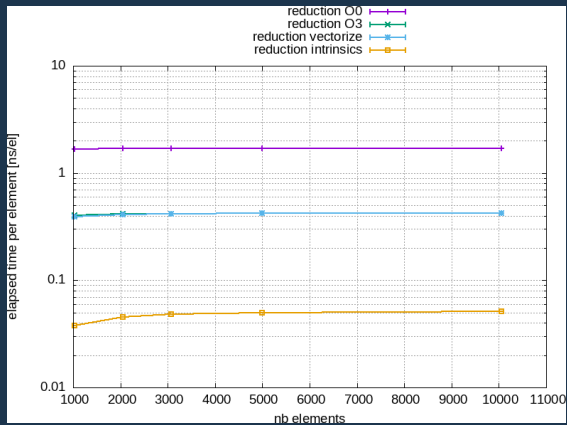
G++ 11



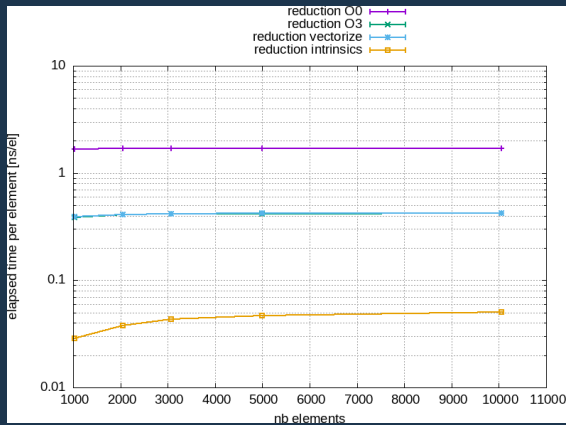
CLang++ 14



G++ 11

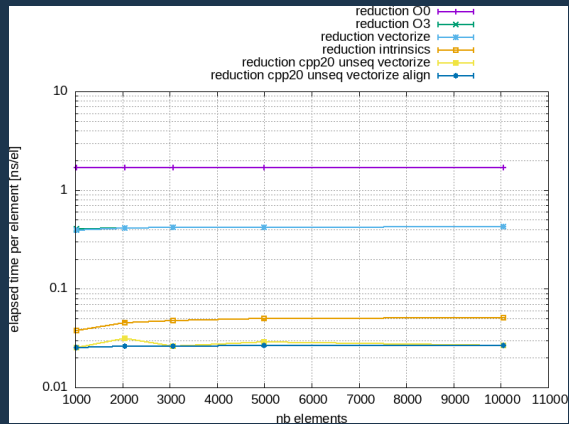


CLang++ 14

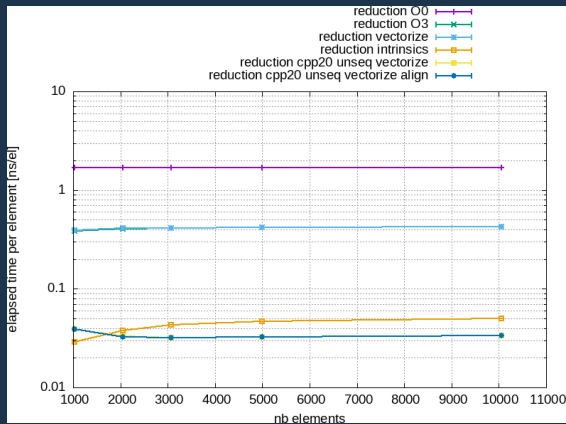


Reduction : Vectorization

G++ 11

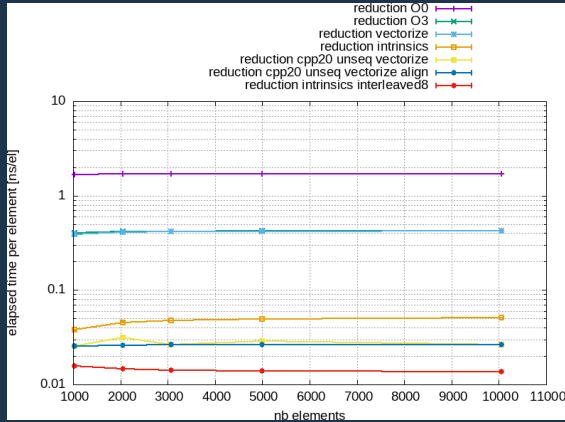


CLang++ 14

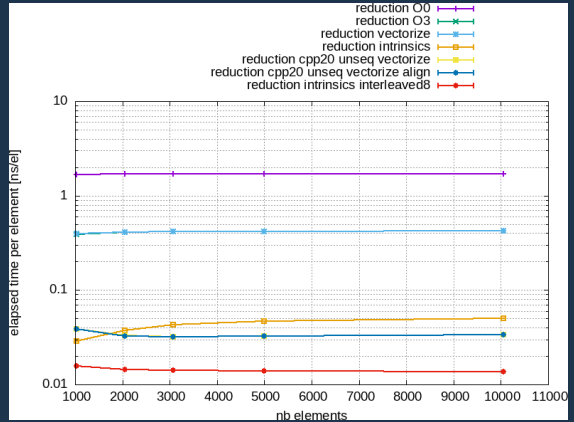


Reduction : Vectorization

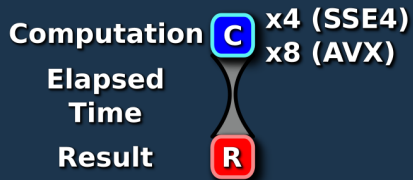
G++ 11



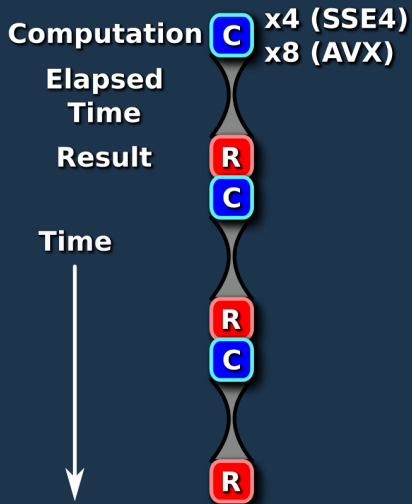
CLang++ 14





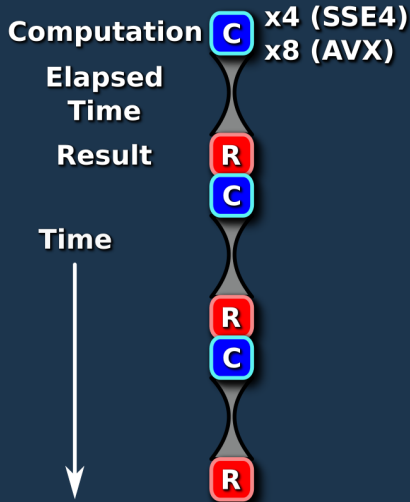


1 accumulator

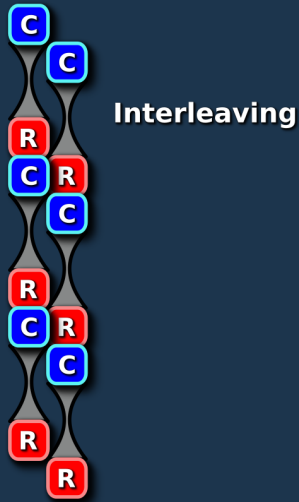


Reduction optimisation

1 accumulator

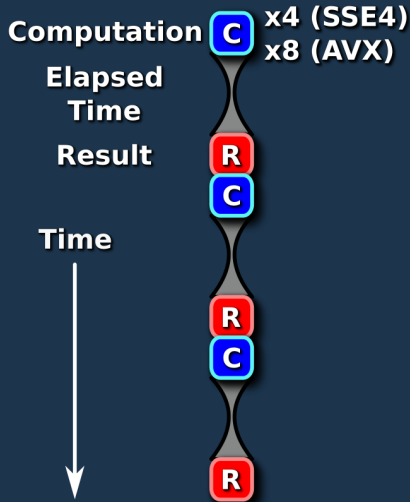


2 accumulators

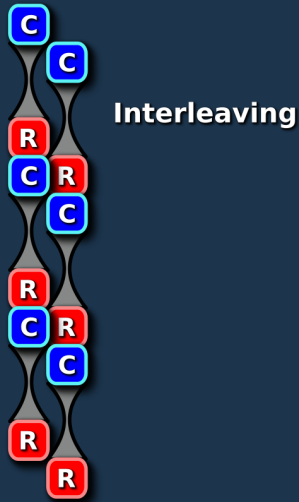


Reduction optimisation

1 accumulator



2 accumulators

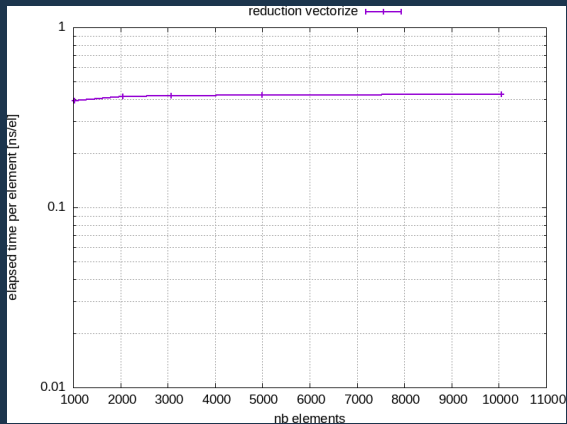


4 accumulators

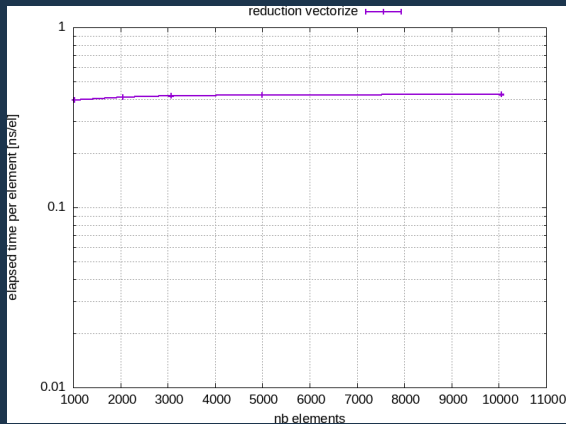


Reduction : Intrinsic Interleaved

G++ 11

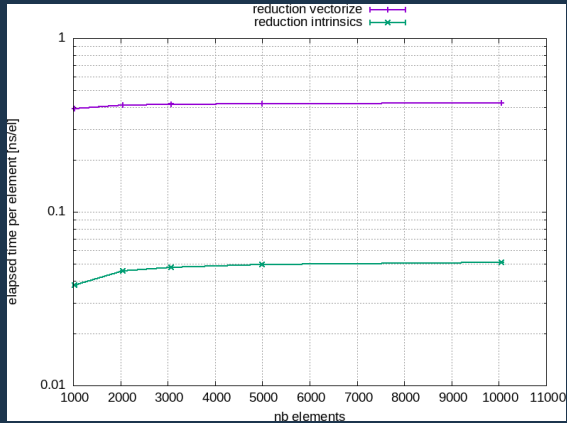


CLang++ 14

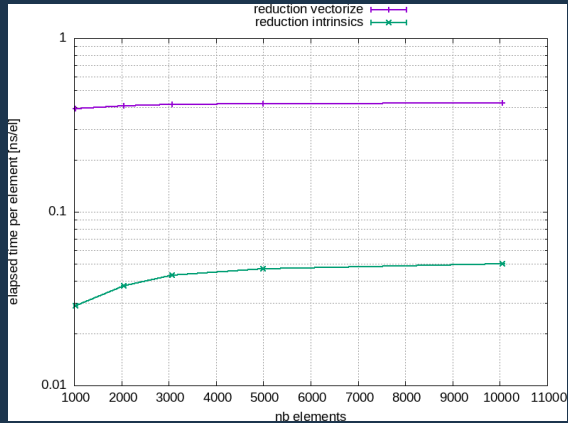


Reduction : Intrinsic Interleaved

G++ 11

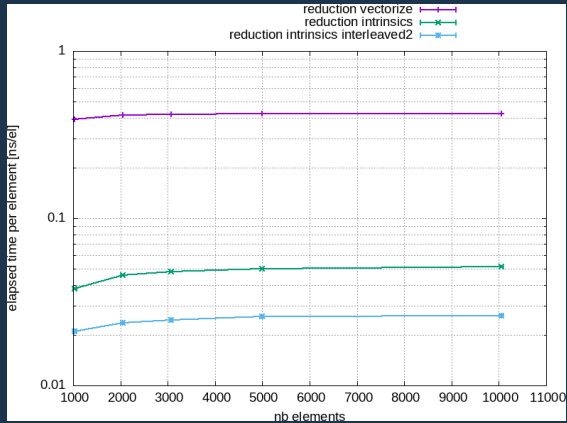


CLang++ 14

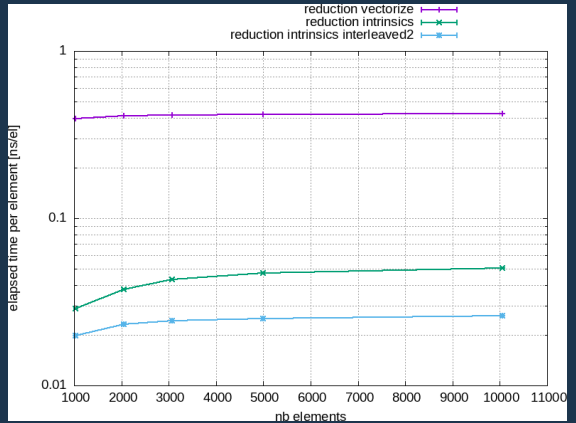


Reduction : Intrinsic Interleaved

G++ 11

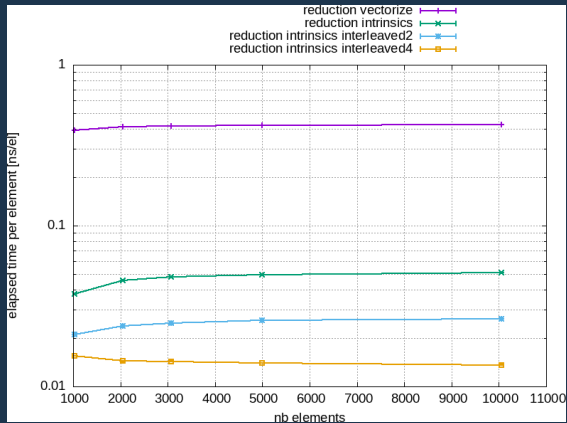


CLang++ 14

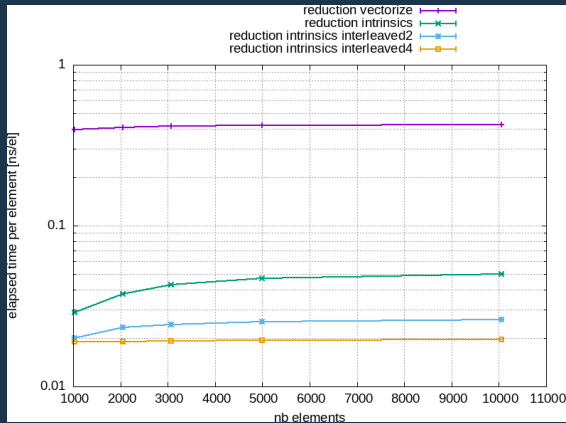


Reduction : Intrinsic Interleaved

G++ 11

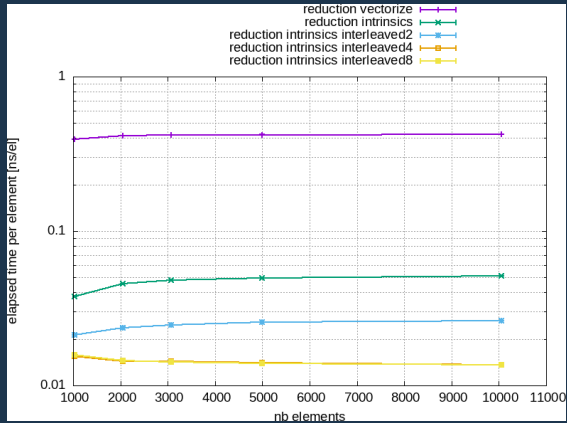


CLang++ 14

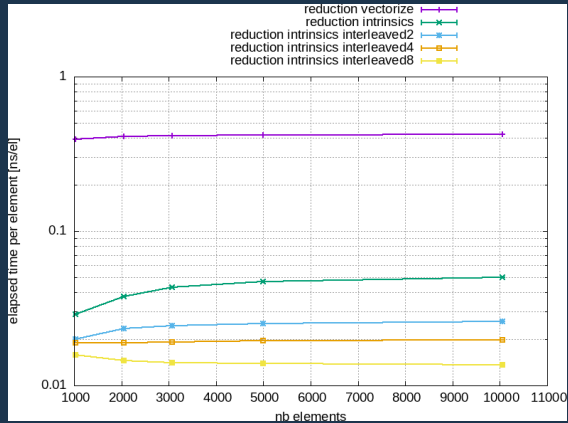


Reduction : Intrinsic Interleaved

G++ 11



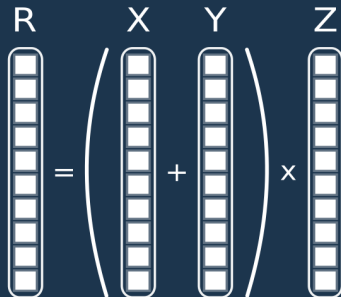
CLang++ 14



Triadic : $z = x + y$

Triadic : $z = x + y$

Quadriadic Computation

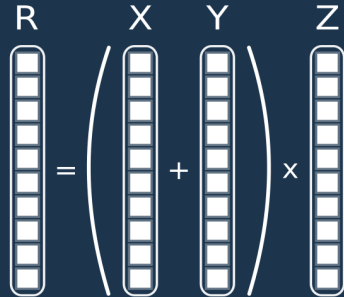


Triadic : $z = x + y$

Classic C++

```
for(long unsigned int i(0lu); i < nbElement; ++i){
    >> tabRes[i] = (tabX[i] + tabY[i])*tabZ[i];
}
```

Quadriadic Computation



Triadic : $z = x + y$

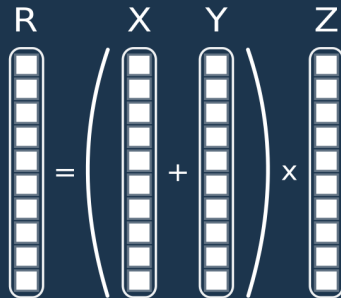
Classic C++

```
for(long unsigned int i(0lu); i < nbElement; ++i){
    >>     tabRes[i] = (tabX[i] + tabY[i])*tabZ[i];
}
```

C++ 17 / 20 / 23

```
std::transform(std::execution::par_unseq,
    >>     std::begin(vecIndex), std::end(vecIndex),
    >>     std::begin(vecX), std::begin(vecRes),
    >>     [=](int i, float x){
    >>         >>         return (x + vecY[i]) * vecZ[i];
    >>     });
```

Quadriadic Computation



Triadic : $z = x + y$

Classic C++

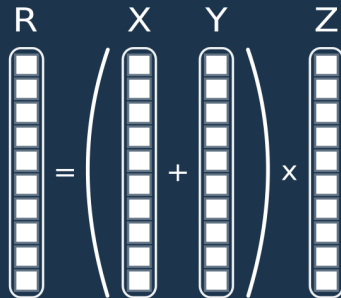
```
for(long unsigned int i(0lu); i < nbElement; ++i){
    >> tabRes[i] = (tabX[i] + tabY[i])*tabZ[i];
}
```

C++ 17 / 20 / 23

```
std::transform(std::execution::par_unseq,
    >> std::begin(vecIndex), std::end(vecIndex),
    >> std::begin(vecX), std::begin(vecRes),
    >> [=](int i, float x){
    >>     >> return (x + vecY[i]) * vecZ[i];
    >> });
```

vecY, vecZ have to be std::vector

Quadriadic Computation



std::transform : triadic

Triadic : $z = x + y$

Classic C++

```
for(long unsigned int i(0lu); i < nbElement; ++i){
    >> tabRes[i] = (tabX[i] + tabY[i])*tabZ[i];
}
```

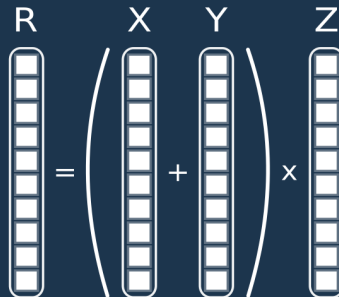
C++ 17 / 20 / 23

```
std::transform(std::execution::par_unseq,
    >> std::begin(vecIndex), std::end(vecIndex),
    >> std::begin(vecX), std::begin(vecRes),
    >> [=](int i, float x){
    >>     >> return (x + vecY[i]) * vecZ[i];
    >> });
```

vecY, vecZ have to be std::vector

Fully Vectorized

Quadriadic Computation



std::transform : triadic

Triadic : $z = x + y$

Classic C++

```
for(long unsigned int i(0lu); i < nbElement; ++i){
    >> tabRes[i] = (tabX[i] + tabY[i])*tabZ[i];
}
```

C++ 17 / 20 / 23

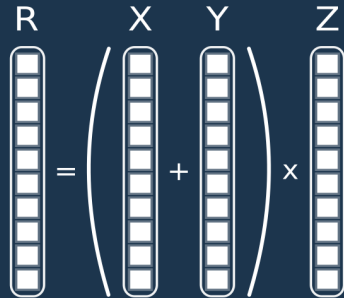
```
std::transform(std::execution::par_unseq,
    >> std::begin(vecIndex), std::end(vecIndex),
    >> std::begin(vecX), std::begin(vecRes),
    >> [=](int i, float x){
    >>     >> return (x + vecY[i]) * vecZ[i];
    >> });
```

vecY, vecZ have to be std::vector

Fully Vectorized

Needs extra index table

Quadriadic Computation



std::transform : triadic

Triadic : $z = x + y$

Classic C++

```
for(long unsigned int i(0lu); i < nbElement; ++i){
    >> tabRes[i] = (tabX[i] + tabY[i])*tabZ[i];
}
```

C++ 17 / 20 / 23

```
std::transform(std::execution::par_unseq,
    >> std::begin(vecIndex), std::end(vecIndex),
    >> std::begin(vecX), std::begin(vecRes),
    >> [=](int i, float x){
    >> >> return (x + vecY[i]) * vecZ[i];
    >> });
```

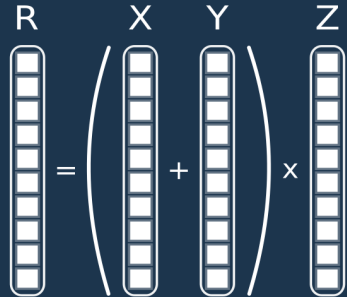
vecY, vecZ have to be std::vector

Fully Vectorized

Needs extra index table

Not vectorized
with **std::for_each**

Quadriadic Computation



std::transform : triadic

Triadic : $z = x + y$

Classic C++

```
for(long unsigned int i(0lu); i < nbElement; ++i){
    >> tabRes[i] = (tabX[i] + tabY[i])*tabZ[i];
}
```

C++ 17 / 20 / 23

```
std::transform(std::execution::par_unseq,
    >> std::begin(vecIndex), std::end(vecIndex),
    >> std::begin(vecX), std::begin(vecRes),
    >> [=](int i, float x){
    >> >> return (x + vecY[i]) * vecZ[i];
    >> });
```

vecY, vecZ have to be std::vector

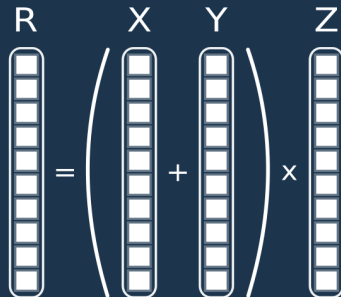
Fully Vectorized

Needs extra index table

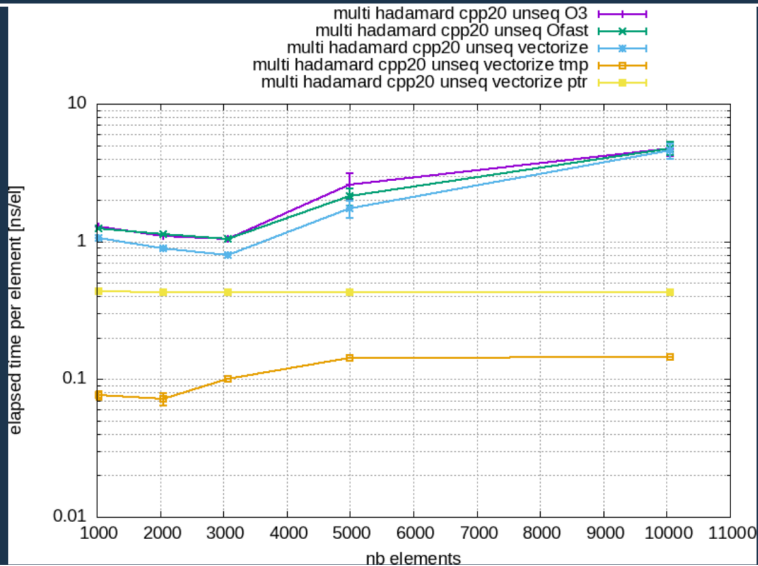
Not vectorized
with **std::for_each**

Not vectorized
with pointers **vecX, vecY**

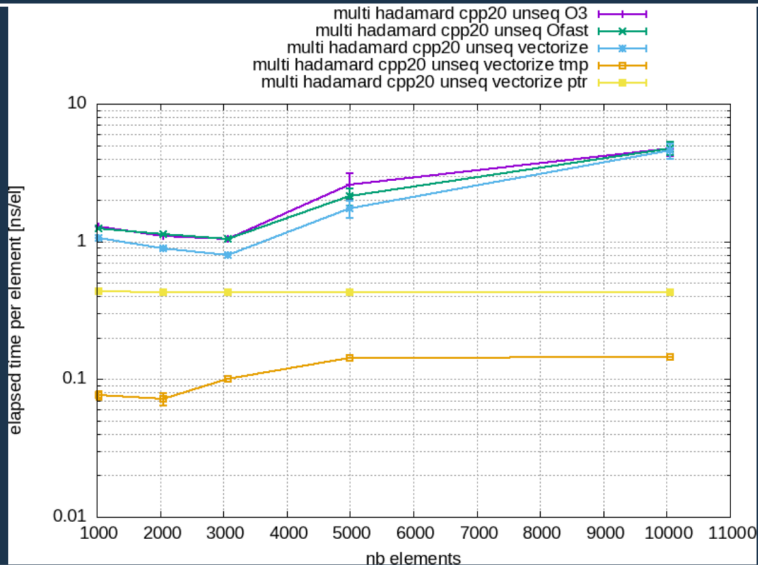
Quadriadic Computation



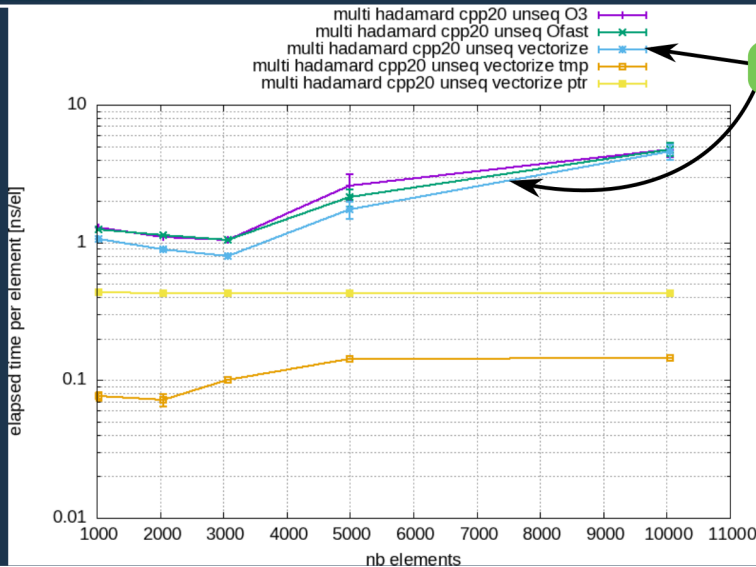
std::transform : (X + Y) x Z



std::transform : (X + Y) x Z

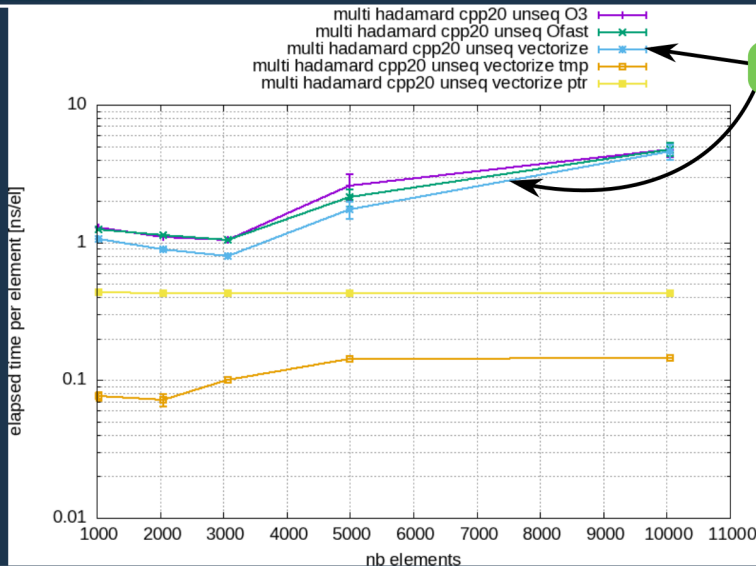


std::transform : (X + Y) x Z



Vectorised

std::transform : (X + Y) x Z

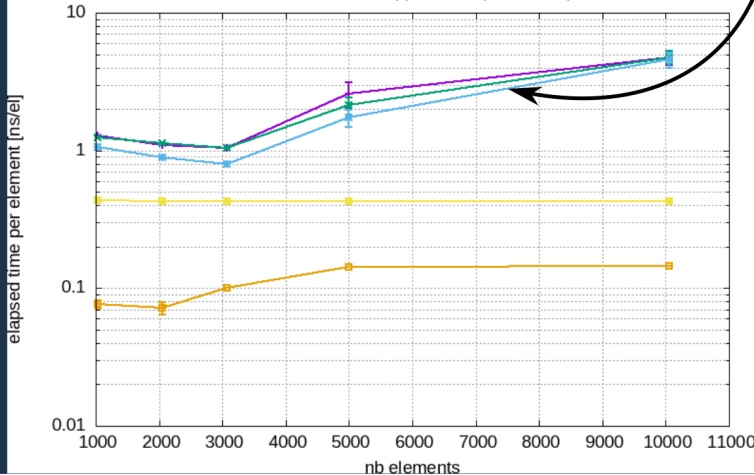


Vectorised

But too much **branching** and **data copy**

std::transform : (X + Y) x Z

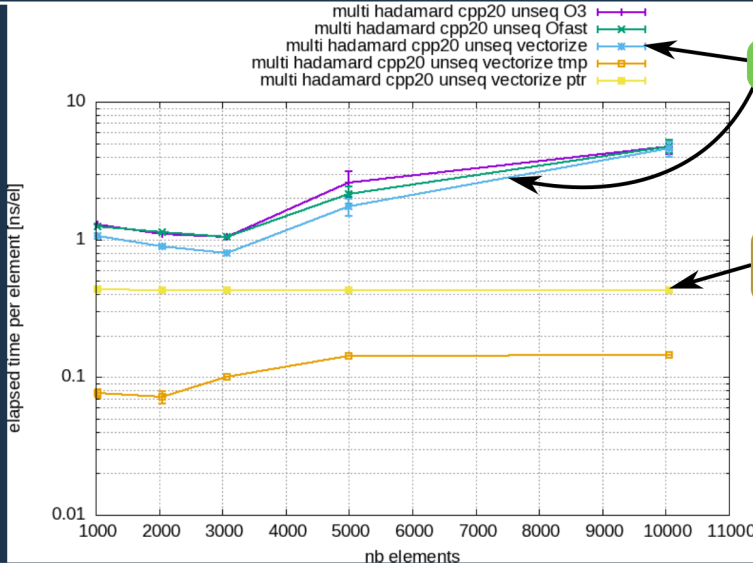
multi hadamard cpp20 unseq O3
 multi hadamard cpp20 unseq Ofast
 multi hadamard cpp20 unseq vectorize
 multi hadamard cpp20 unseq vectorize tmp
 multi hadamard cpp20 unseq vectorize ptr



Vectorised

But too much **branching** and **data copy**

std::transform : (X + Y) x Z

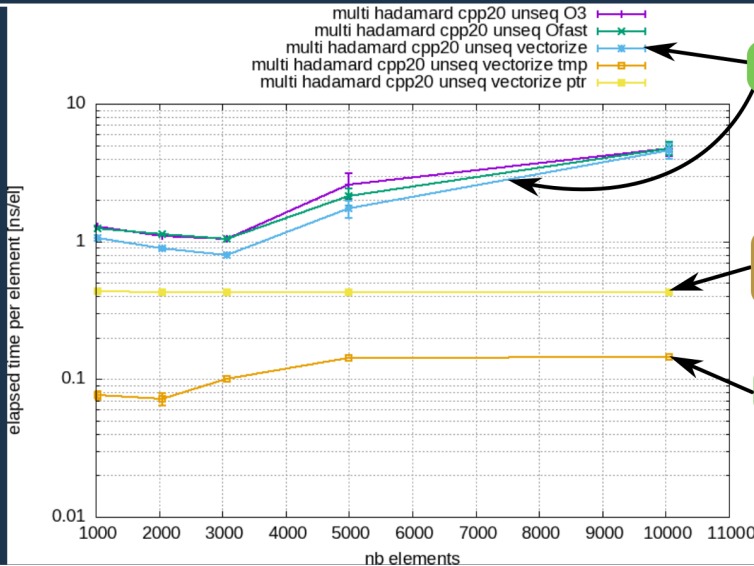


Vectorised

But too much **branching**
and **data copy**

Not Vectorized but
no **data copy**

std::transform : (X + Y) x Z



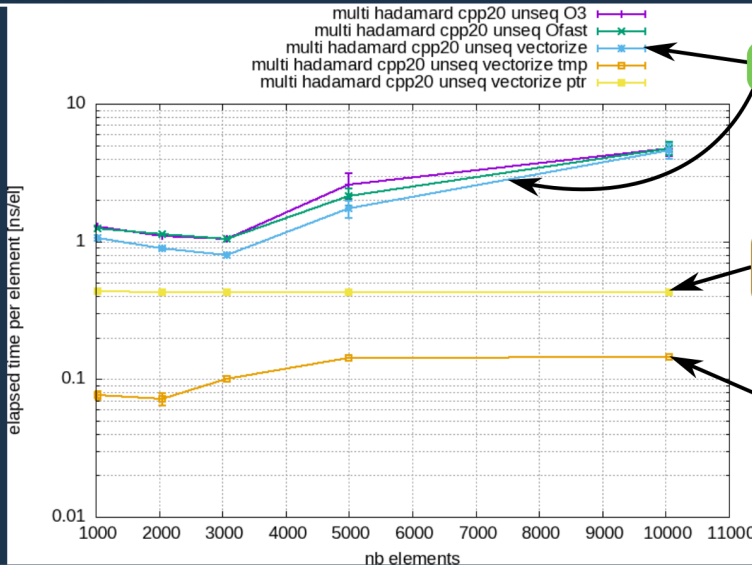
Vectorised

But too much **branching** and **data copy**

Not Vectorized but no **data copy**

Vectorised

std::transform : (X + Y) x Z



Vectorised

But too much **branching** and **data copy**

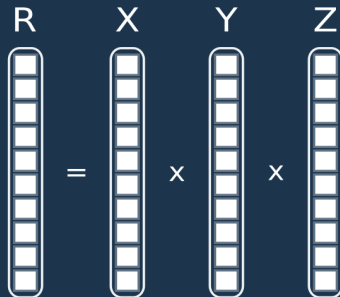
Not Vectorized but no **data copy**

Vectorised

But use **temporary vecXY**



Quadriadic Computation



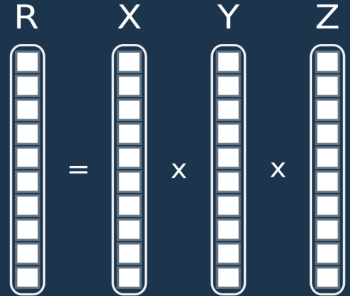
std::views::transform

```

auto vXY = std::views::transform(std::views::zip(vecX, vecY),
    > >      [](auto tuple){
    > > >      auto & [x, y] = tuple;
    > > >      return x*y;
    > >      }
);
std::transform(std::execution::par_unseq,
    >      std::begin(vXY), std::end(vXY), std::begin(vecZ),
    >      std::begin(vecRes),
    >      [](float xy, float z){
    > >      return xy *z;
    > >      }
);

```

Quadriadic Computation



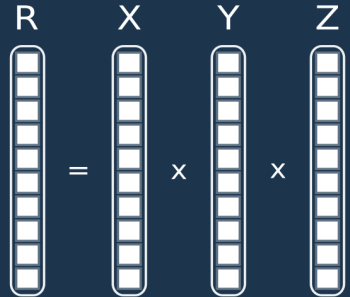
std::views::transform

```

auto vXY = std::views::transform(std::views::zip(vecX, vecY, vecZ),
> > > [](auto tuple){
> > > > auto & [x, y, z] = tuple;
> > > > return x*y*z;
> > > }
);
std::transform(EXECUTION_POLICY,
> std::begin(vXY), std::end(vXY),
> std::begin(vecRes),
> [](float res){
> > > return res;
> > }
);

```

Quadriadic Computation



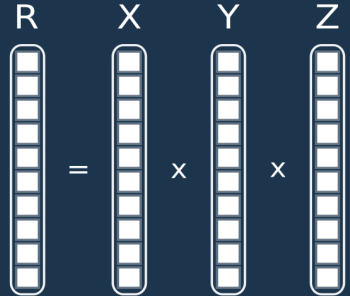
std::views::transform

```

auto vXY = std::views::transform(std::views::zip(vecX, vecY, vecZ),
> > > [](auto tuple){
> > > > auto & [x, y, z] = tuple;
> > > > return x*y*z;
> > > }
);
std::transform(EXECUTION_POLICY,
> std::begin(vXY), std::end(vXY),
> std::begin(vecRes),
> [](float res){
> > > return res;
> > }
);
    
```

No extra table needed

Quadriadic Computation



std::views::transform

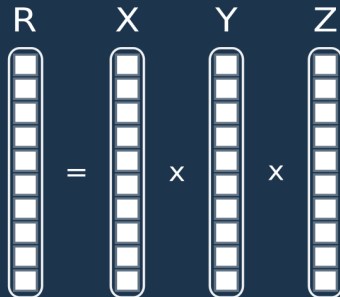
```

auto vXY = std::views::transform(std::views::zip(vecX, vecY, vecZ),
> >      [](auto tuple){
> > >      auto & [x, y, z] = tuple;
> > >      return x*y*z;
> >      }
);
std::transform(EXECUTION_POLICY,
>      std::begin(vXY), std::end(vXY),
>      std::begin(vecRes),
>      [](float res){
> >      return res;
> >      }
);
    
```

No extra table needed

Not vectorized yet
because of **std::views::zip**

Quadriadic Computation



std::views::transform

```

auto vXY = std::views::transform(std::views::zip(vecX, vecY, vecZ),
> >      [](auto tuple){
> > >      auto & [x, y, z] = tuple;
> > >      return x*y*z;
> > >      }
);
std::transform(EXECUTION_POLICY,
>      std::begin(vXY), std::end(vXY),
>      std::begin(vecRes),
>      [](float res){
> >      return res;
> >      }
);

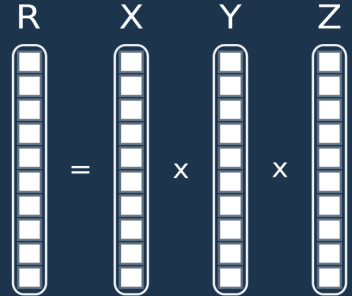
```

No extra table needed

Use **std::tuple**
Contiguous elements

Not vectorized yet
because of **std::views::zip**

Quadriadic Computation



std::views::transform

```

auto vXY = std::views::transform(std::views::zip(vecX, vecY, vecZ),
> > > [](auto tuple){
> > >     auto & [x, y, z] = tuple;
> > >     return x*y*z;
> > > }
);
std::transform(EXECUTION_POLICY,
> std::begin(vXY), std::end(vXY),
> std::begin(vecRes),
> [](float res){
> >     return res;
> > }
);

```

No extra table needed

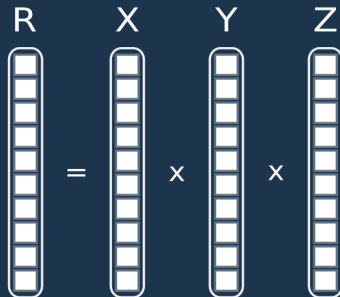
Not vectorized yet
because of **std::views::zip**

Use **std::tuple**
Contiguous elements

Not Vectorisable



Quadriadic Computation



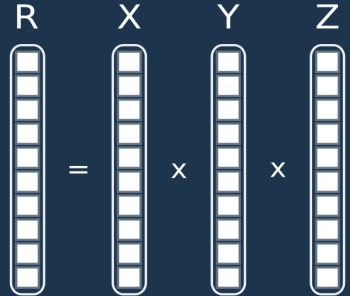
std::views::transform

```

auto vXY = std::views::transform(std::views::zip(vecX, vecY, vecZ),
> > > [](auto tuple){
> > >     auto & [x, y, z] = tuple;
> > >     return x*y*z;
> > > }
);
std::transform(EXECUTION_POLICY,
> std::begin(vXY), std::end(vXY),
> std::begin(vecRes),
> [](float res){
> >     return res;
> > }
);

```

Quadriadic Computation



Use **std::tuple**
Contiguous elements

Not vectorized yet
because of **std::views::zip**

No extra table needed

Not Vectorisable



Vectorisable

