Phoenix

Language-Driven Auto-Vectorisation

Jonas Sys, Elisa Gonzalez Boix, Christophe Scholliers

GHENT UNIVERSITY

VUB VRIJE UNIVERSITEIT BRUSSEL

Phoenix - Quick Recap

Phoenix: Language-Driven Auto-Vectorisation

Jonas Sys
jonas.sys@ugent.be
Ghent University and Vrije Universiteit Brussel

Elisa Gonzalez Boix
egonzale@vub.be
Vrije Universiteit Brussel

Christophe Scholliers
christophe.scholliers@ugent.be
Ghent University

1. Research Context

Modern CPUs are equipped with SIMD units, allowing a single CPU core to evaluate the same instruction on multiple data values in parallel. Even commodity hardware is able to achieve significant performance improvements by leveraging data-parallelism.

Branching control flow obscures data-parallelism in scalar code

The problem: manual vectorisation is very complex, and existing tools fail to provide guarantees for efficiently exploiting SIMD units.

2. Existing Approaches

Manual vectorisation: guaranteed performance at extreme complexity

Automatic (auto-)vectorisation: no developer burden, but also no performance guarantees

Language extensions: low developer burden, but minor guarantees to performance and correctness (data race conditions)

3. *Phoenix*

Phoenix is a data-parallel programming language, designed to allow safe, efficient vectorisation while still closely resembling scalar code by employing restricted vector loops.
All restricted vector loops are subject to these three restrictions:

Iteration independence: to eliminate loop-carried dependencies, vector loops are not allowed to modify non-loop-local variables.

Loop completion: to be able to pre-compute the loop-trip-count, vector loops cannot early exit (i.e., no break or return).

Index-free loop operation: reading masked-out elements introduces undefined behavior. *Phoenix* prohibits arbitrary indexing in vector loops.

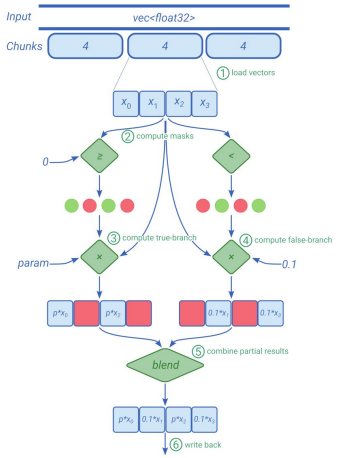
These three restrictions guarantee safe and correct vectorisability

Example: leaky scaled ReLU.

```
void leaky_relu(vec<float32> &xs, float32 param) {
  foreach x in xs {
    if (x ⩾ 0.0f) continue with param * x;
    else continue with 0.1f * x;
  }
}
```

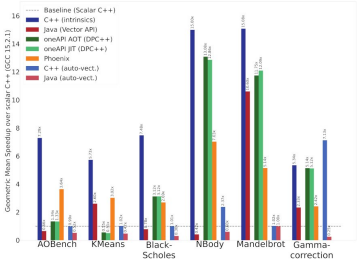
Phoenix's design eliminates compiler guesswork, guaranteeing vectorisability

4. Simplified Vectorisation Steps



5. Evaluation

We evaluated *Phoenix* through a benchmark suite with kernels from various application domains, showing that its expressivity is not hampered by its restrictions.

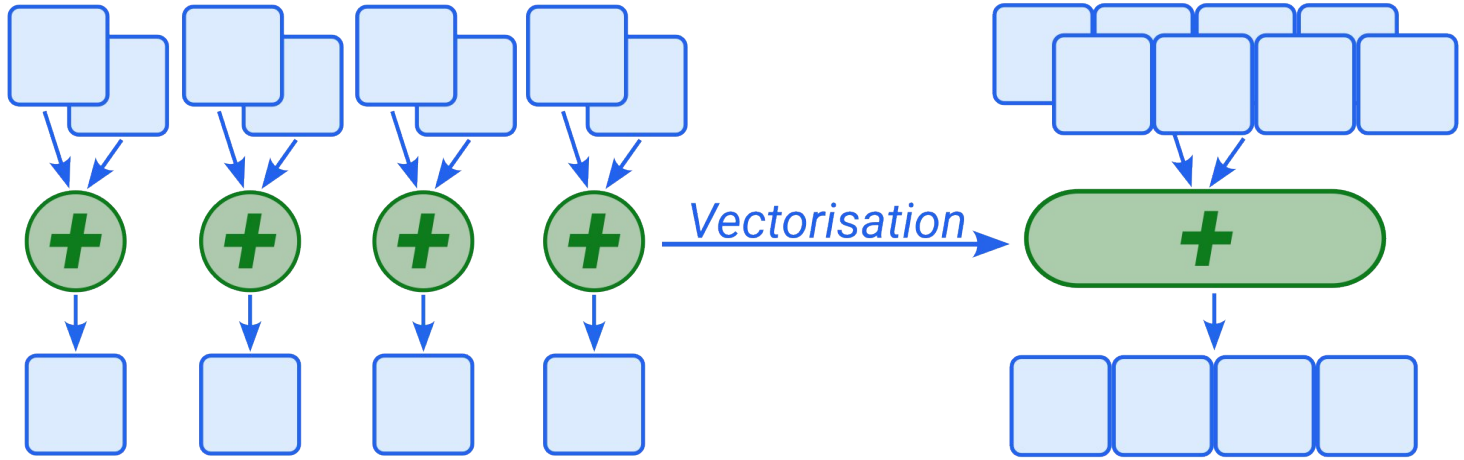


Complex control flow does not inhibit *Phoenix*'s vectoriser.

GHENT UNIVERSITY FACULTY OF SCIENCES

VUB VRIJE UNIVERSITEIT BRUSSEL

Phoenix - Quick Recap



Guaranteed performance
Complex & error-prone

Possible race-conditions

Scalar code
Not always possible

Phoenix - Quick Recap

Restricted Vector Loops

#1: Iteration Independence

#2: Loop Completion

#3: Index-Free Loop Operations

```
foreach (x, idx) in view {}
  (modify in-place)
mask (x, idx) in view {}
  (filter in-place)
project (x, idx) in view into output {}
  (modify out-of-place)
merge (x1, …, xn, idx) in (view1, …, viewn) into output {}
  (combine multiple views)
accumulate (x, idx) in view with initial into var {}
  (reduce to single value)
```

`continue`

mask out *current* output element

`continue with` e

write *current* output element

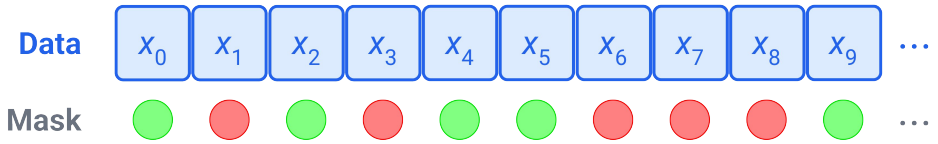
`continue[`idx`] with (`x`) ->` e

update *arbitrary* output element
executes scalar (correctness)

```
void flip_sign(vec<float32> &data) {
    foreach (x, idx) in data {
        if (x < 0.0f)
            continue with -x;
        else
            continue;
    }
}
```

Phoenix - Nuts and Bolts

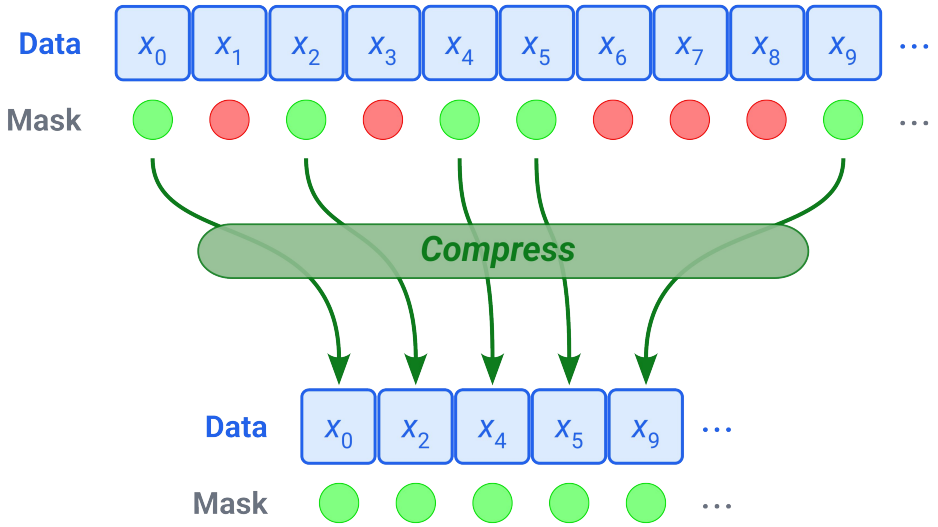
Views



Mutable: `view<T>`
Immutable: `cview<T>`
Constructed from data source: `new view<T>(vec<T> &data)`
Typical operations: `view.size()`, `view[index]`, ...
Low-level operations: `view.set(index, value)`, `view.skip(index)`, `view.keep_if(index, bool)`, `view.compress()`

Phoenix - Nuts and Bolts

Views




```
void flip_sign(vec<float32> &data) {
    foreach (x, idx) in data {
        if (x < 0.0f)
            continue with -x;
        else
            continue;
    }
}
```

type-checker

```
void flip_sign(vec<float32> &data) {
  view<float32> data_view = new view<float32>(data);
  foreach (x, idx) in data_view {
    if (x < 0.0f)
      continue with -x;
    else
      continue;
  }
  data_view.compress(); //upon destruction
}
```

Compiling *Phoenix* - Lowering

```
foreach (x, idx) in view { // view is view<float32>
  continue with 7.0f * x;
}
```


```
for (uint32 idx = 0; idx < view.size(); idx += 1) {
    if (!view.in_view(idx)) continue;
    const float32 &x = view[idx];
    float32 temp = 7.0f * x;
    view.set(idx, temp);
    continue;
}
```

Compiling *Phoenix* - Lowering

```
foreach (x, idx) in view { // view is view<float32>
  continue with 7.0f * x;
}
```


```
for (uint32 idx = 0; idx < view.size(); idx += 1) {
    if (!view.in_view(idx)) continue;
    const float32 &x = view[idx];
    float32 temp = 7.0f * x;
    view.set(idx, temp);
    continue;
}
```

Compiling *Phoenix* - Lowering

```
foreach (x, idx) in view { // view is view<float32>
  continue with 7.0f * x;
}
```


```
for (uint32 idx = 0; idx < view.size(); idx += 1) {
    if (!view.in_view(idx)) continue;
    const float32 &x = view[idx];
    float32 temp = 7.0f * x;
    view.set(idx, temp);
    continue;
}
```

Compiling *Phoenix* - Lowering

```
foreach (x, idx) in view { // view is view<float32>
  continue with 7.0f * x;
}
```


```
for (uint32 idx = 0; idx < view.size(); idx += 1) {
    if (!view.in_view(idx)) continue;
    const float32 &x = view[idx];
    float32 temp = 7.0f * x;
    view.set(idx, temp);
    continue;
}
```

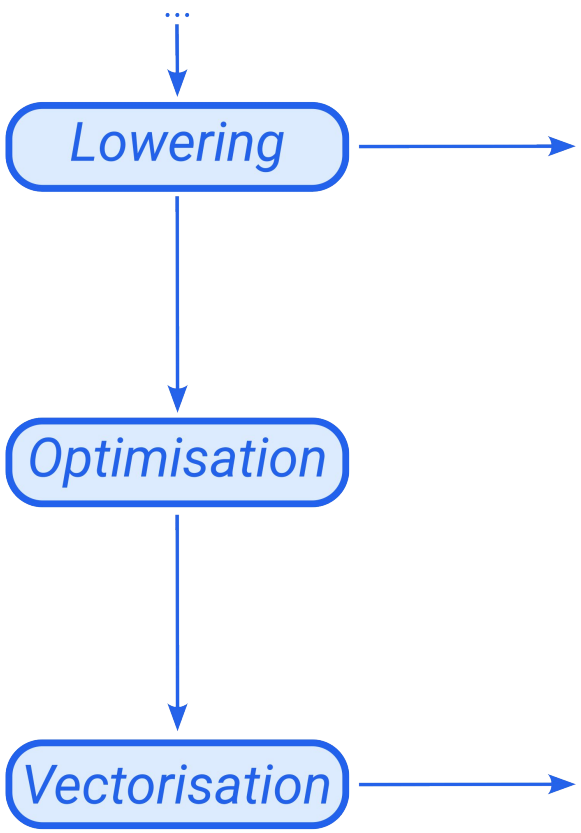
`continue` → `view.skip(idx)`

`continue with x` → `view.set(idx, x)`

`mask` loops: `continue with b` → `view.keep_if(idx, b)`

Compiling *Phoenix* - Vectorisation

```
foreach (x, idx) in view { // view is view<float32>
  continue with f(x, idx);
}
```

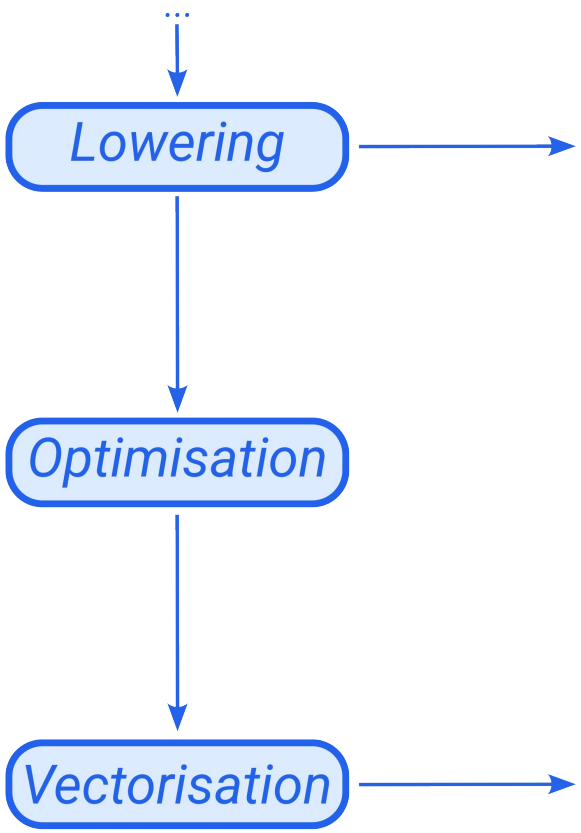



```
for (uint32 idx = 0; idx < view.size(); idx += 1) {
    if (!view.in_view(idx)) continue;
    const float32 &x = view[idx];
    float32 temp = 7.0f * x;
    view.set(idx, temp);
    continue;
}
```

```
Vec_u32 idx = Vector::of(-N, -N-1, ..., -1); // N elements
Vec_u32 offset = Vector::of(N, N, ..., N); // N elements
for (uint32_t i = 0; i < view.size(); i += N) {
    idx = Vector::add(idx, offset);
    Mask_N mask = view.mask_at(i);
    if (mask.none_set()) continue;
    Vec_f32 x = Vector::load_masked(mask, &view[i]);
    Vec_f32 const_7 = Vector::of(7.0f, ..., 7.0f);
    Vec_f32 temp = Vector::multiply(const_7, x);
    Vector::store_masked(mask, &view[i], temp);
    view.mask_enable_at(i, mask);
    continue;
}
```

Compiling *Phoenix* - Vectorisation

```
foreach (x, idx) in view { // view is view<float32>
  continue with f(x, idx);
}
```

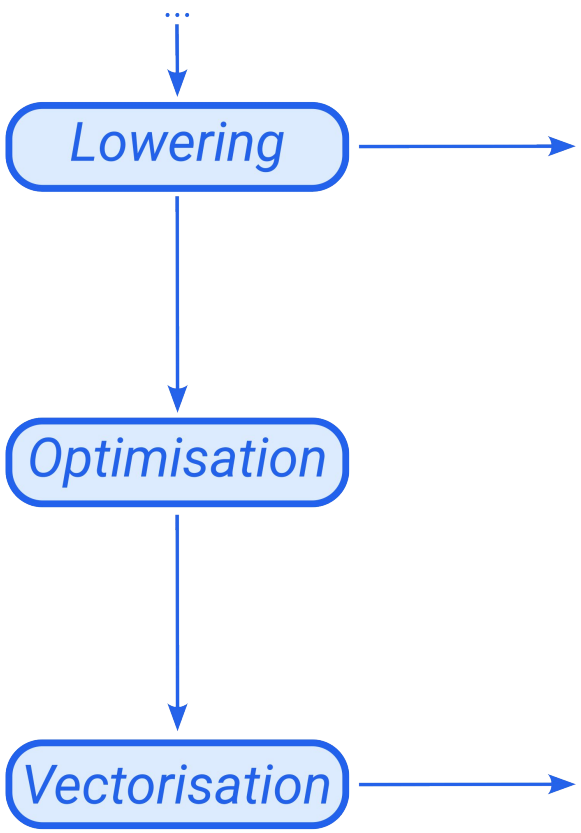



```
for (uint32 idx = 0; idx < view.size(); idx += 1) {
    if (!view.in_view(idx)) continue;
    const float32 &x = view[idx];
    float32 temp = 7.0f * x;
    view.set(idx, temp);
    continue;
}
```

```
Vec_u32 idx = Vector::of(-N, -N-1, ..., -1); // N elements
Vec_u32 offset = Vector::of(N, N, ..., N); // N elements
for (uint32_t i = 0; i < view.size(); i += N) {
    idx = Vector::add(idx, offset);
    Mask_N mask = view.mask_at(i);
    if (mask.none_set()) continue;
    Vec_f32 x = Vector::load_masked(mask, &view[i]);
    Vec_f32 const_7 = Vector::of(7.0f, ..., 7.0f);
    Vec_f32 temp = Vector::multiply(const_7, x);
    Vector::store_masked(mask, &view[i], temp);
    view.mask_enable_at(i, mask);
    continue;
}
```

Compiling *Phoenix* - Vectorisation

```
foreach (x, idx) in view { // view is view<float32>
  continue with f(x, idx);
}
```

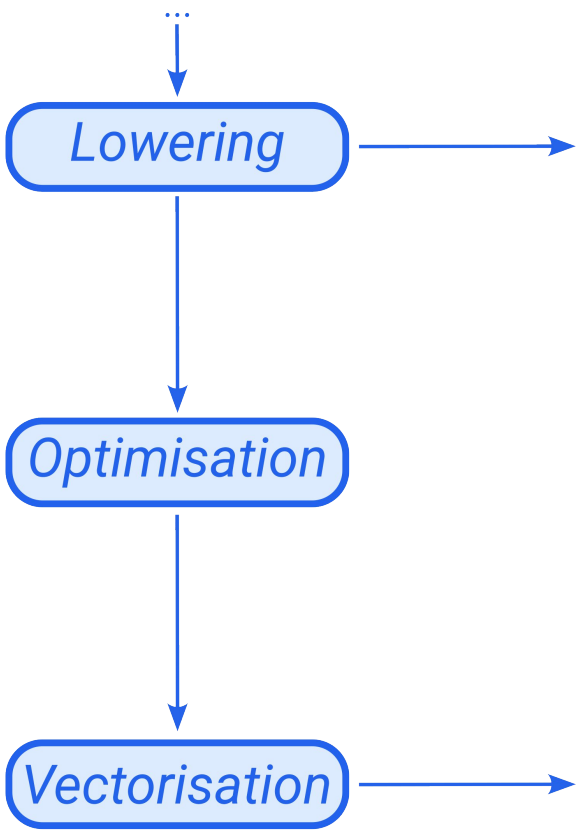



```
for (uint32 idx = 0; idx < view.size(); idx += 1) {
    if (!view.in_view(idx)) continue;
    const float32 &x = view[idx];
    float32 temp = 7.0f * x;
    view.set(idx, temp);
    continue;
}
```

```
Vec_u32 idx = Vector::of(-N, -N-1, ..., -1); // N elements
Vec_u32 offset = Vector::of(N, N, ..., N); // N elements
for (uint32_t i = 0; i < view.size(); i += N) {
    idx = Vector::add(idx, offset);
    Mask_N mask = view.mask_at(i);
    if (mask.none_set()) continue;
    Vec_f32 x = Vector::load_masked(mask, &view[i]);
    Vec_f32 const_7 = Vector::of(7.0f, ..., 7.0f);
    Vec_f32 temp = Vector::multiply(const_7, x);
    Vector::store_masked(mask, &view[i], temp);
    view.mask_enable_at(i, mask);
    continue;
}
```

Compiling *Phoenix* - Vectorisation

```
foreach (x, idx) in view { // view is view<float32>
  continue with f(x, idx);
}
```

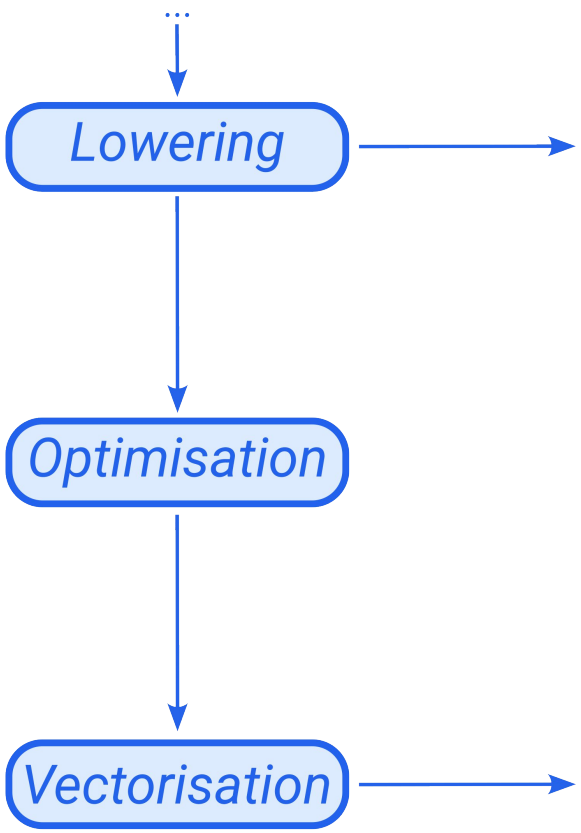



```
for (uint32 idx = 0; idx < view.size(); idx += 1) {
    if (!view.in_view(idx)) continue;
    const float32 &x = view[idx];
    float32 temp = 7.0f * x;
    view.set(idx, temp);
    continue;
}
```

```
Vec_u32 idx = Vector::of(-N, -N-1, ..., -1); // N elements
Vec_u32 offset = Vector::of(N, N, ..., N); // N elements
for (uint32_t i = 0; i < view.size(); i += N) {
    idx = Vector::add(idx, offset);
    Mask_N mask = view.mask_at(i);
    if (mask.none_set()) continue;
    Vec_f32 x = Vector::load_masked(mask, &view[i]);
    Vec_f32 const_7 = Vector::of(7.0f, ..., 7.0f);
    Vec_f32 temp = Vector::multiply(const_7, x);
    Vector::store_masked(mask, &view[i], temp);
    view.mask_enable_at(i, mask);
    continue;
}
```

Compiling *Phoenix* - Vectorisation

```
foreach (x, idx) in view { // view is view<float32>
  continue with f(x, idx);
}
```




```
for (uint32 idx = 0; idx < view.size(); idx += 1) {
    if (!view.in_view(idx)) continue;
    const float32 &x = view[idx];
    float32 temp = 7.0f * x;
    view.set(idx, temp);
    continue;
}
```

```
Vec_u32 idx = Vector::of(-N, -N-1, ..., -1); // N elements
Vec_u32 offset = Vector::of(N, N, ..., N); // N elements
for (uint32_t i = 0; i < view.size(); i += N) {
    idx = Vector::add(idx, offset);
    Mask_N mask = view.mask_at(i);
    if (mask.none_set()) continue;
    Vec_f32 x = Vector::load_masked(mask, &view[i]);
    Vec_f32 const_7 = Vector::of(7.0f, ..., 7.0f);
    Vec_f32 temp = Vector::multiply(const_7, x);
    Vector::store_masked(mask, &view[i], temp);
    view.mask_enable_at(i, mask);
    continue;
}
```

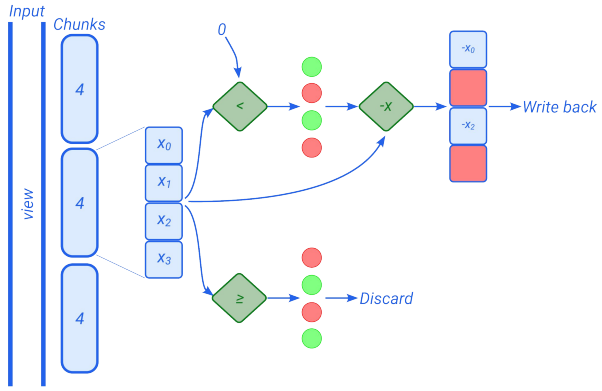
Compiling *Phoenix* - Branches

```
foreach (x, idx) in view {
    if (x < 0.0f)
        continue with -x;
    else
        continue;
}
```

Lowering

```
for (uint32 idx = 0; idx < view.size(); idx += 1) {
    if (!view.in_view(idx)) continue;
    const float32 &x = view[idx];
    if (x < 0.0f) {
        view.set(idx, -x);
        continue;
    }
    else {
        view.skip(idx);
        continue;
    }
    continue;
}
```

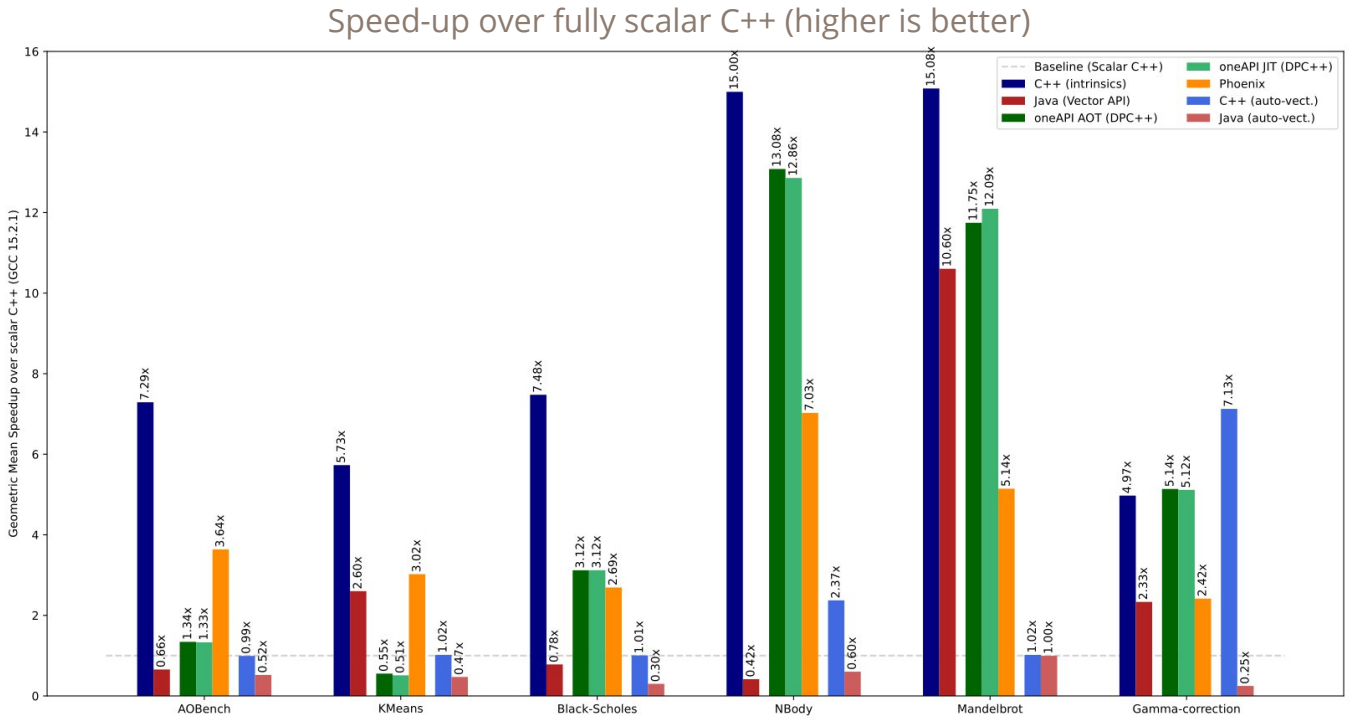
Vectorisation



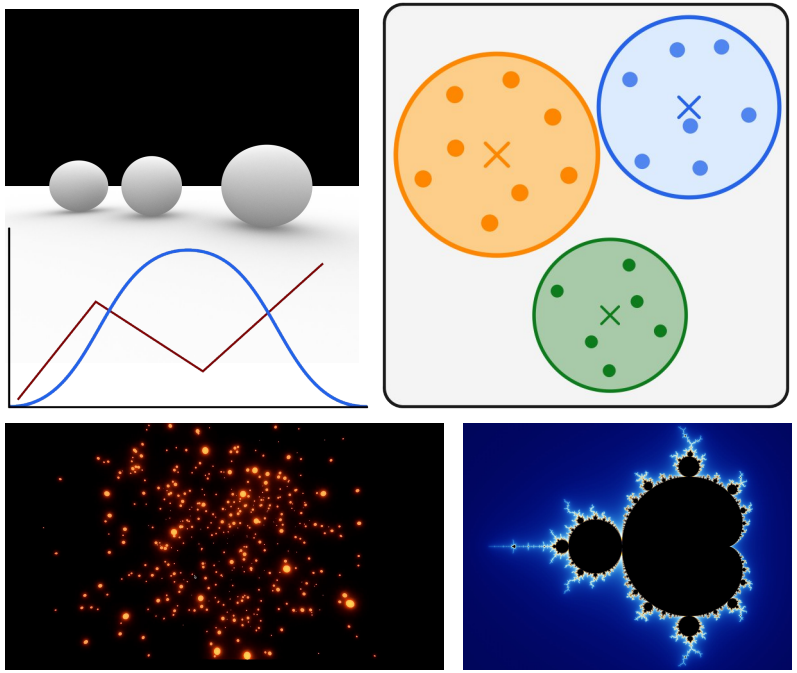

```
Vec_u32 idx = Vector::of(-N, -N-2, ..., -1); // N elements
Vec_u32 offset = Vector::of(N, N, ..., N); // N elements
for (uint32_t i = 0; i < view.size(); i += N) {
    idx = Vector::add(idx, offset);
    Mask_N mask = view.mask_at(i);
    if (mask.none_set()) continue;
    Vec_f32 x = Vector::load_masked(mask, &view[i]);

    Mask_N mask_true = Vector::lt(x, Vector::of(0.04045f, ...));
    Vec_f32 res_true = Vector::negate(x);
    Vector::store_masked(mask_true, &view[i], res_true);
    view.mask_enable_at(i, mask_true);

    Mask_N mask_false = mask & (~mask_true);
    view.mask_disable_at(i, mask_false);

    continue;
}
```

Evaluation

Execution Speed



Expressiveness



Conclusion

Jonas Sys
jonas.sys@ugent.be