Autovesk: Automatic vectorized code generation from unstructured static kernels using graph transformations

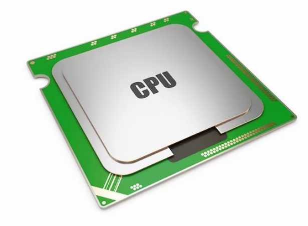
Hayfa TAYEB[1,2] Ludovic PAILLAT[3] Bérenger BRAMAS[2,3]

[1] University of Bordeaux, France
[2] Inria, France
[3] University of Strasbourg, France

18/01/2024

What is Vectorization ?

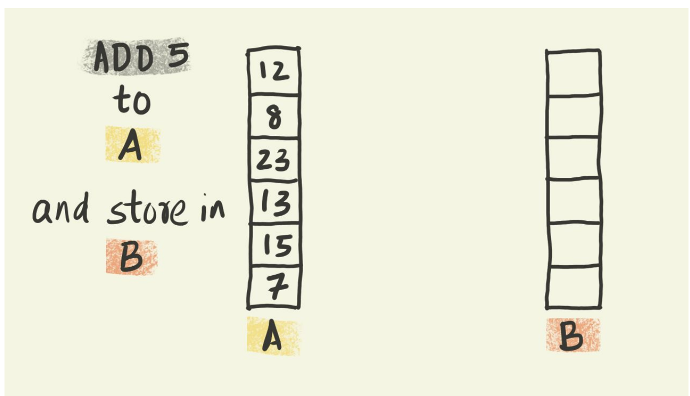
Vectorization is a feature of modern CPUs that consists of applying a single instruction to multiple data (SIMD) .



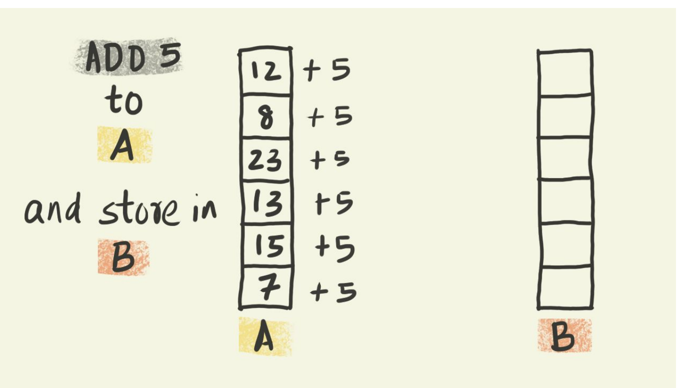
SIMD architecture (Single Instruction Multiple Data)

A computing unit executes the same instruction on different data sets.

Sequential vs. SIMD approach



6 scalar operations



1 vector operation

Why consider SIMD ?

Performance !

How to use SIMD ?

There are a number of alternatives and tools for implementing vectorization. They differ in complexity and flexibility.

Ease of Use

Compiler
Auto-vectorization (no change of code)

Compiler #pragma

Librairies

Vector intrinsic

Assembly code

Programmer

Existing auto-vectorization techniques

<table>
<tr><th colspan="2">Regular applications</th></tr>
<tr><th>Modern compilers
(GCC, Clang, Intel)</th><th>Specialized compilers</th></tr>
<tr><td>Loop vectorization: data-level parallelism</td><td rowspan="2">Polyhedral compilers: affine loops</td></tr>
<tr><td>Superword Level Parallelism (SLP): Instruction-oriented vectorization</td></tr>
</table>

<table>
<tr><th colspan="2">Irregular applications
(graph algorithms, particle simulation codes, sparse matrix codes)</th></tr>
<tr><td rowspan="2">Inspector/executor transformations: Sparse Polyhedral Framework (SPF): non-affine array accesses</td><td>Conflict masking: resolving data conflicts in SIMD vectors</td></tr>
<tr><td>Non-SIMD vector instructions: VeGen framework</td></tr>
</table>

Auto-vectorization challenges

1. Data dependencies

```
// read-after-write dependency
A[0] = 0;
for (int i=1; i<SIZE; i++)
        a[i] = a[i-1] +1;
```

2. Indirect Memory Access

```
// indirect addressing of x using index array
for (int i=0; i<SIZE; i+=2)
        b[i] += a[i] * x[index[i]];
```

3. Non-contiguous data accesses

```
// arrays accessed with stride 2
for (int i=0; i<SIZE; i+=2)
        out[i]+= a[i] * b[i];

// shifted array accesses
for (int i=0; i<SIZE; i++)
        out[i] += a[(i+2)%SIZE] * b[(i+2)%SIZE];
```

Illustration of shifted data accesses

Kernel A:

```
for(long int idx = 0 ; idx < size ; ++idx){
    outValue[idx] = inValue1[(idx+2)%size] * inValue2[(idx*2)%size];
}
```

Load inValue1

0	1	2	3	4	5

Load inValue2

0	1	2	3	4	5

Operations

*	*	*	*	*	*

Store outValue

0	1	2	3	4	5

Compiler

SPOILER !

Speedups

Execution times

10^{-1} 10^{-2} 10^{-3}

0 50 100

Test case size

1.32 1.53 1.97 4.15 1.73 4.67 0.61 11.09

INTEL-AVX512 ARM-SVE

GCC autovectorization — Autovesk

Usually

- ➔ Auto-vectorization comes for free (automatic).
- ➔ Without developer intervention, we cross our fingers that the loop will be vectorized.
- ➔ But sometimes (especially in high-performance computing applications) loops and vectorization need to be fine tuned.

Our goal ?

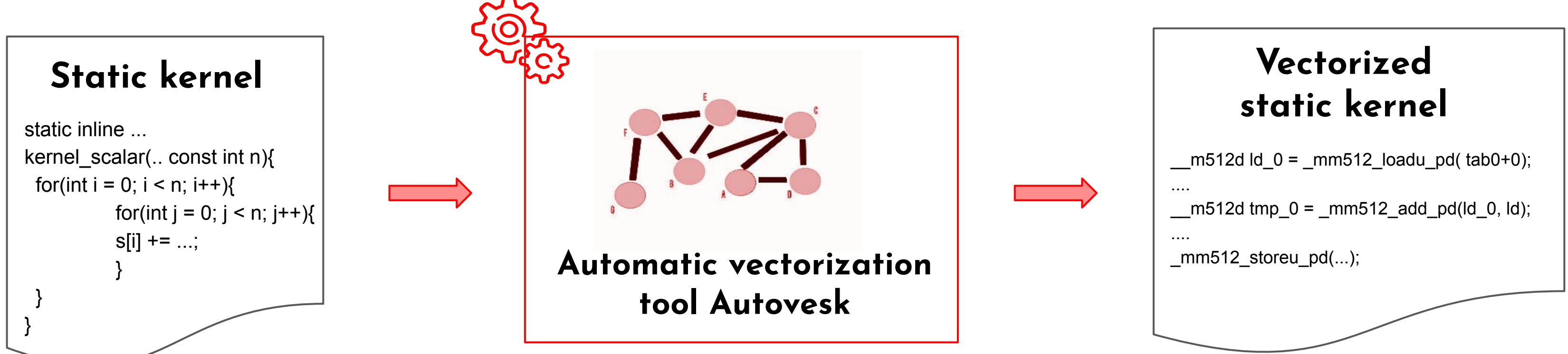
Automatically transform **scalar operations** into **vector operations**

in irregular applications with non-contiguous data access patterns,

leading to performance improvements on modern processors

Hayfa Tayeb, Ludovic Paillat, and Bérenger Bramas. 2023. Autovesk: Automatic Vectorized Code Generation from Unstructured Static Kernels Using Graph Transformations. ACM Trans. Archit. Code Optim. 21, 1, Article 4 (March 2024), 25 pages.

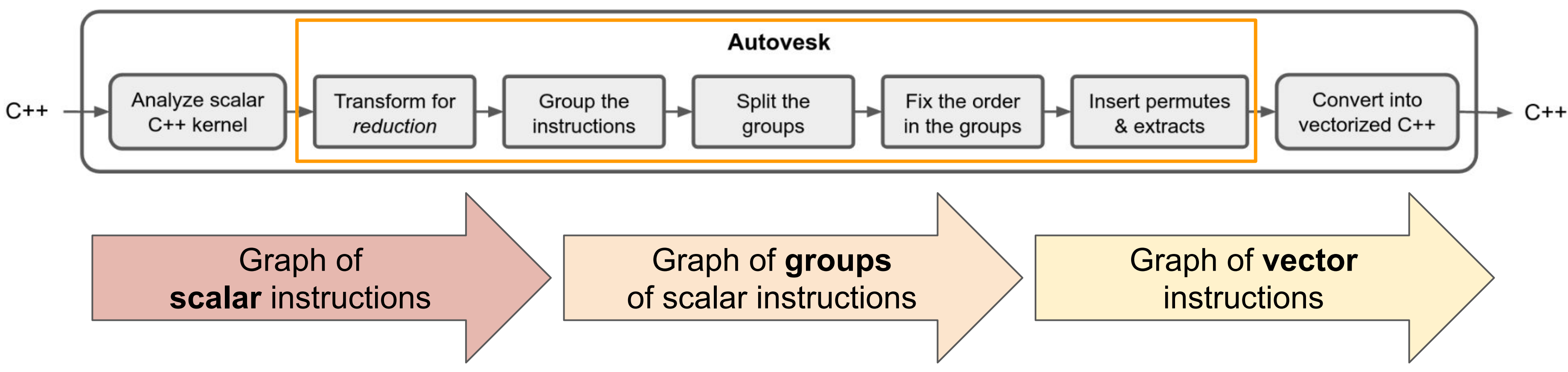
Proposed approach: Autovesk



- Build a directed acyclic graph (DAG) of scalar instructions using a custom C++ tool based on templates and operator overloading
- Use **graph transformations and heuristics** to form a graph of vector instructions
- Translate it into vectorized code through our backend

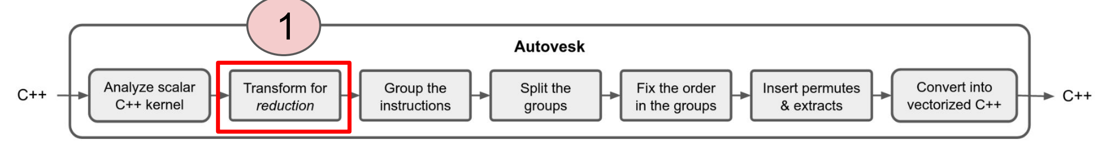
Hayfa Tayeb, Ludovic Paillat, and Bérenger Bramas. 2023. Autovesk: Automatic Vectorized Code Generation from Unstructured Static Kernels Using Graph Transformations. ACM Trans. Archit. Code Optim. 21, 1, Article 4 (March 2024), 25 pages.

Autovesk layers



The 1st and last stages are not needed if Autovesk is integrated into an existing compiler

Hayfa Tayeb, Ludovic Paillat, and Bérenger Bramas. 2023. Autovesk: Automatic Vectorized Code Generation from Unstructured Static Kernels Using Graph Transformations. ACM Trans. Archit. Code Optim. 21, 1, Article 4 (March 2024), 25 pages.



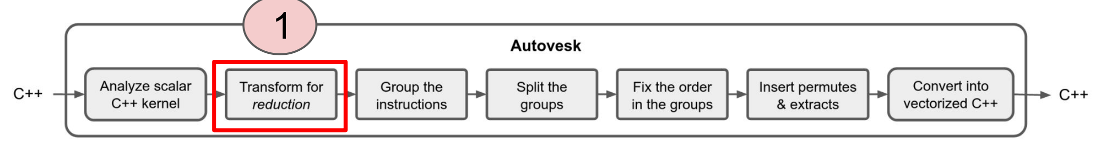
Graph of scalar instructions

Kernel B: Reduction

```
double x = 0;
for(long int idx = 0 ; idx < size ; ++idx){
    x += inValue1[idx] + inValue2[idx];
}
outValue[0] = x;
```

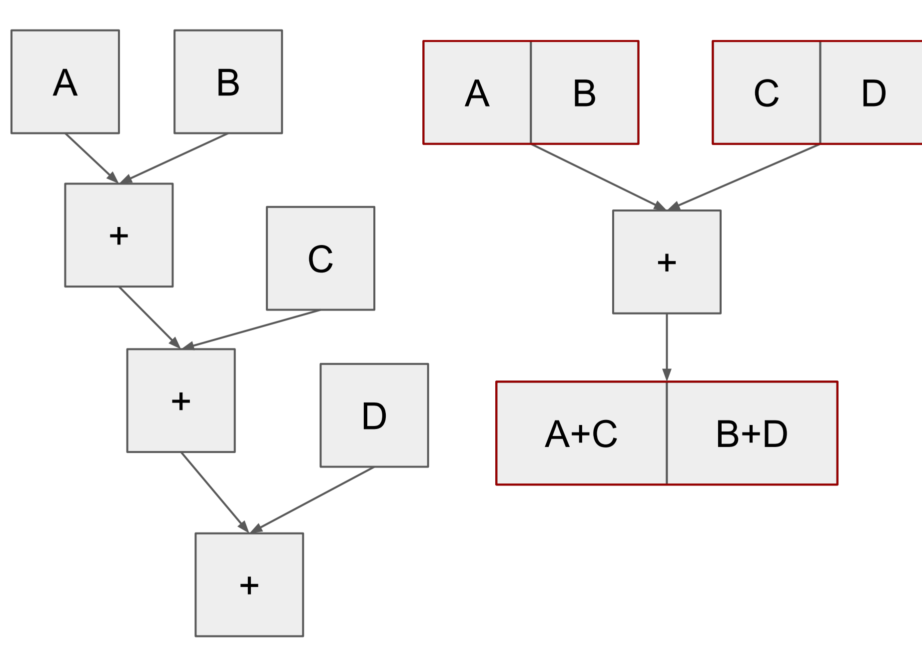
Load @0[0]
Load @1[0]
Set (float)0.000000
+
Load @0[1]
Load @1[1]
+
+
Load @0[2]
Load @1[2]
+
+
Load @0[3]
Load @1[3]
+
+
Load @0[4]
Load @1[4]
+
+
Load @0[5]
Load @1[5]
+
+
+
Store @2[0]

Hayfa Tayeb, Ludovic Paillat, and Bérenger Bramas. 2023. Autovesk: Automatic Vectorized Code Generation from Unstructured Static Kernels Using Graph Transformations. ACM Trans. Archit. Code Optim. 21, 1, Article 4 (March 2024), 25 pages.

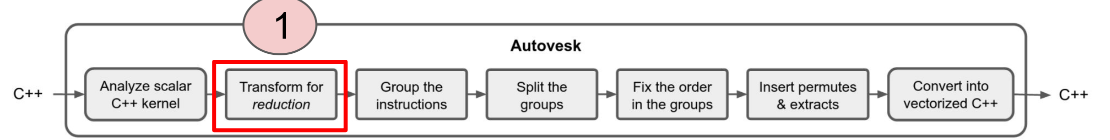


Commutative operations and reductions

- Reduction is a chain of scalar operations
- Reduction can be reordered if the operation is commutative
- Take advantage of the horizontal sum/mul (reduction) available on most CPUs

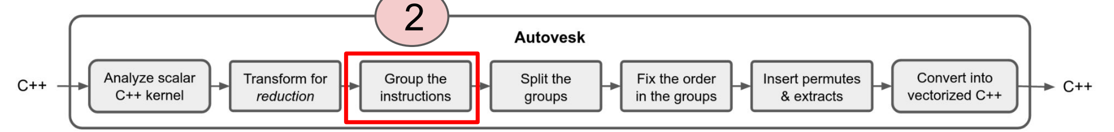


Hayfa Tayeb, Ludovic Paillat, and Bérenger Bramas. 2023. Autovesk: Automatic Vectorized Code Generation from Unstructured Static Kernels Using Graph Transformations. ACM Trans. Archit. Code Optim. 21, 1, Article 4 (March 2024), 25 pages.



Commutative operations and reductions

Hayfa Tayeb, Ludovic Paillat, and Bérenger Bramas. 2023. Autovesk: Automatic Vectorized Code Generation from Unstructured Static Kernels Using Graph Transformations. ACM Trans. Archit. Code Optim. 21, 1, Article 4 (March 2024), 25 pages.



Example: Group scalar instructions graph

Generate a meta-graph where the nodes contain lists of scalar operations that could potentially be vectorized together.

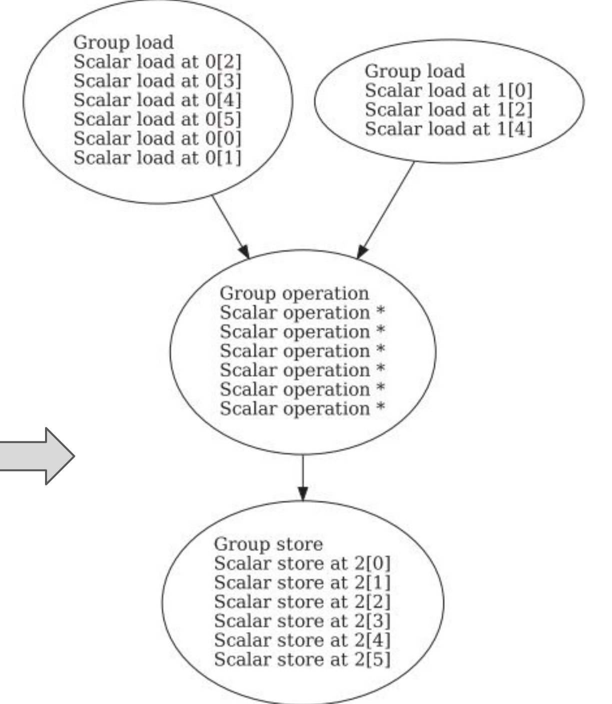
Kernel A:

```
for(long int idx = 0 ; idx < size ; ++idx){
    outValue[idx] = inValue1[(idx+2)%size] * inValue2[(idx*2)%size];
}
```

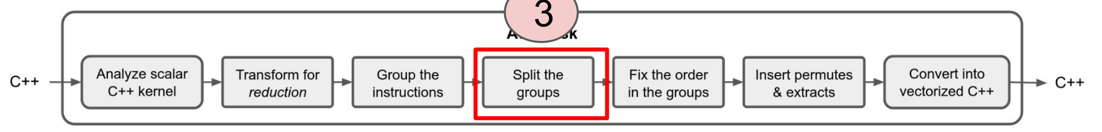
Load @0[2] Load @1[0] Load @0[5] Load @0[3] Load @1[2] Load @0[0] Load @0[4] Load @1[4] Load @0[1]

* * * * * *

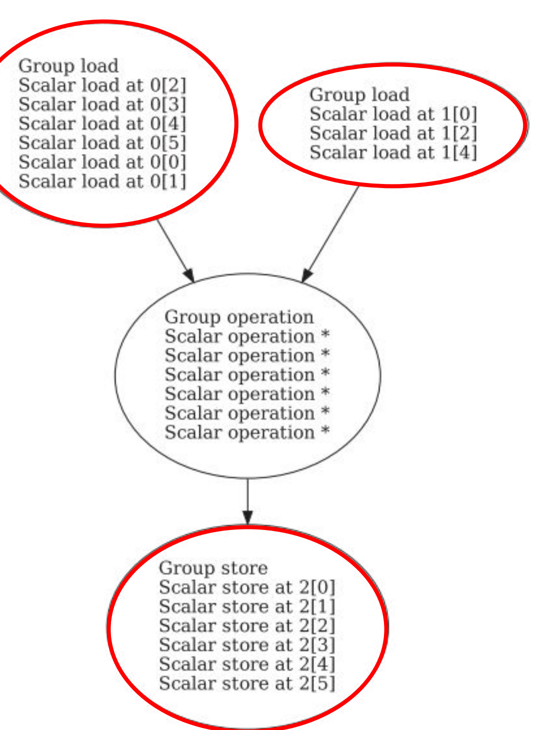
Store @2[0] Store @2[3] Store @2[1] Store @2[4] Store @2[2] Store @2[5]



Hayfa Tayeb, Ludovic Paillat, and Bérenger Bramas. 2023. Autovesk: Automatic Vectorized Code Generation from Unstructured Static Kernels Using Graph Transformations. ACM Trans. Archit. Code Optim. 21, 1, Article 4 (March 2024), 25 pages.

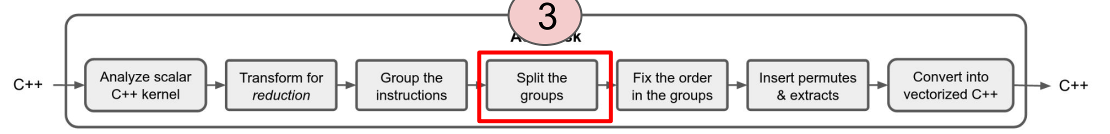


Example: Split load/store groups



Greedy strategy: Generate all combinations satisfying the 3 rules:

1. the values should be loaded in order
2. the number of memory access should be minimal
3. at most one vector should not be full



Example: Split load/store groups

Kernel A: For a vector size 4, we have (2x1x2) possibilities

Group load
Scalar load at 0[2]
Scalar load at 0[3]
Scalar load at 0[4]
Scalar load at 0[5]
Scalar load at 0[0]
Scalar load at 0[1]

Group load
Scalar load at 1[0]
Scalar load at 1[2]
Scalar load at 1[4]

Group load
Scalar load at 2[0]
Scalar load at 2[1]
Scalar load at 2[2]
Scalar load at 2[3]
Scalar load at 2[4]
Scalar load at 2[5]

Array 0

0	1	2	3		4	5

1 load *1 load*

0	1		2	3	4	5

1 load *1 load*

×

Array 1

0	2	4

1 load

×

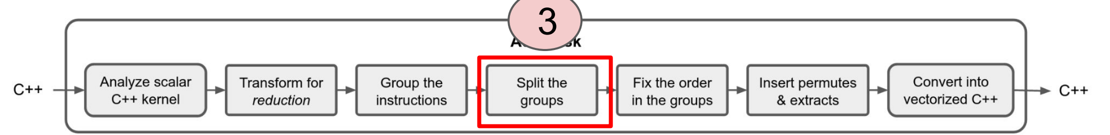
Array 2

0	1	2	3		4	5

1 store *1 store*

0	1		2	3	4	5

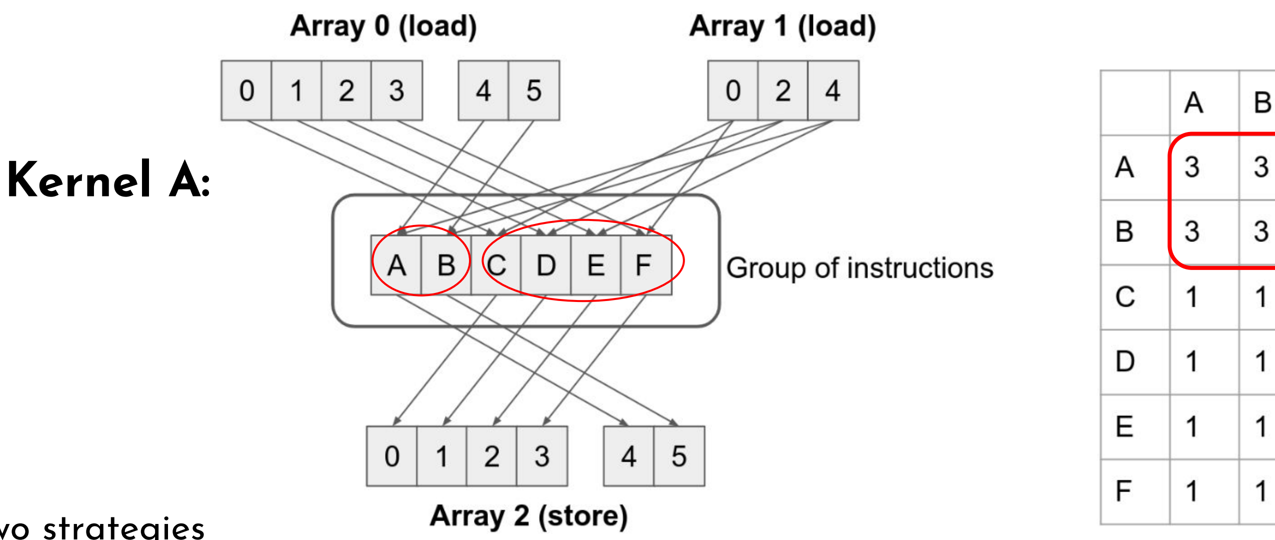
1 store *1 store*



Example: Split the operation groups

Using the Matrix of neighbours to split the group

Kernel A:

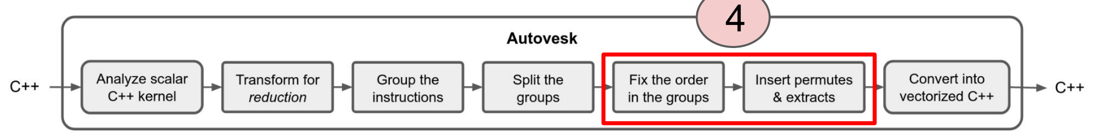


	A	B	C	D	E	F
A	3	3	1	1	1	1
B	3	3	1	1	1	1
C	1	1	3	3	3	3
D	1	1	3	3	3	3
E	1	1	3	3	3	3
F	1	1	3	3	3	3

Two strategies

(1) **clustering strategy** where we create initial sub-groups and then aggregate the elements using the best score remaining so far

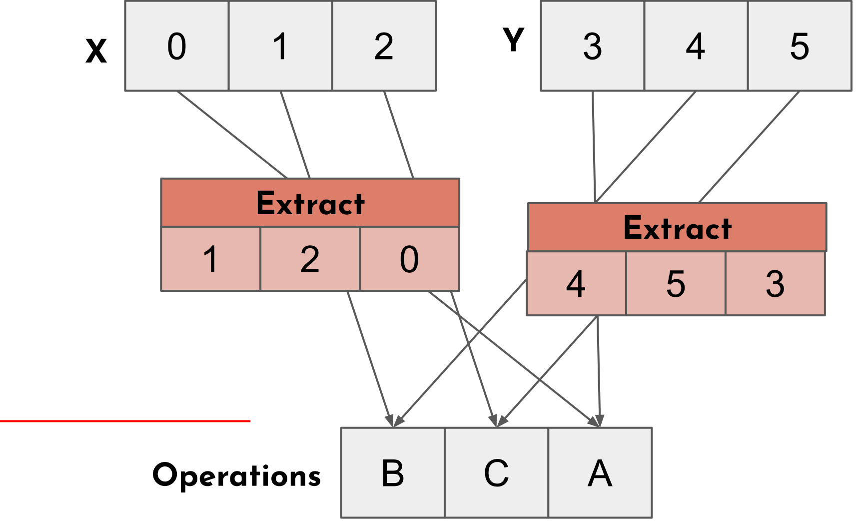
(2) **partitioning strategy** where we split the matrix at the point with the lowest score

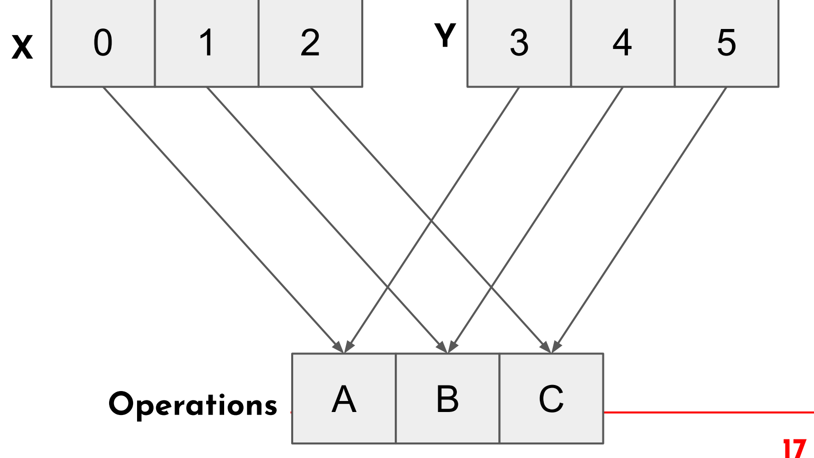


Fix the order in the operation groups

- Use extract/permute instructions, if needed, to ensure kernel consistency
- Fix the order in the operation groups to minimize adding extract/permute instructions

Example: the importance of a "good" order

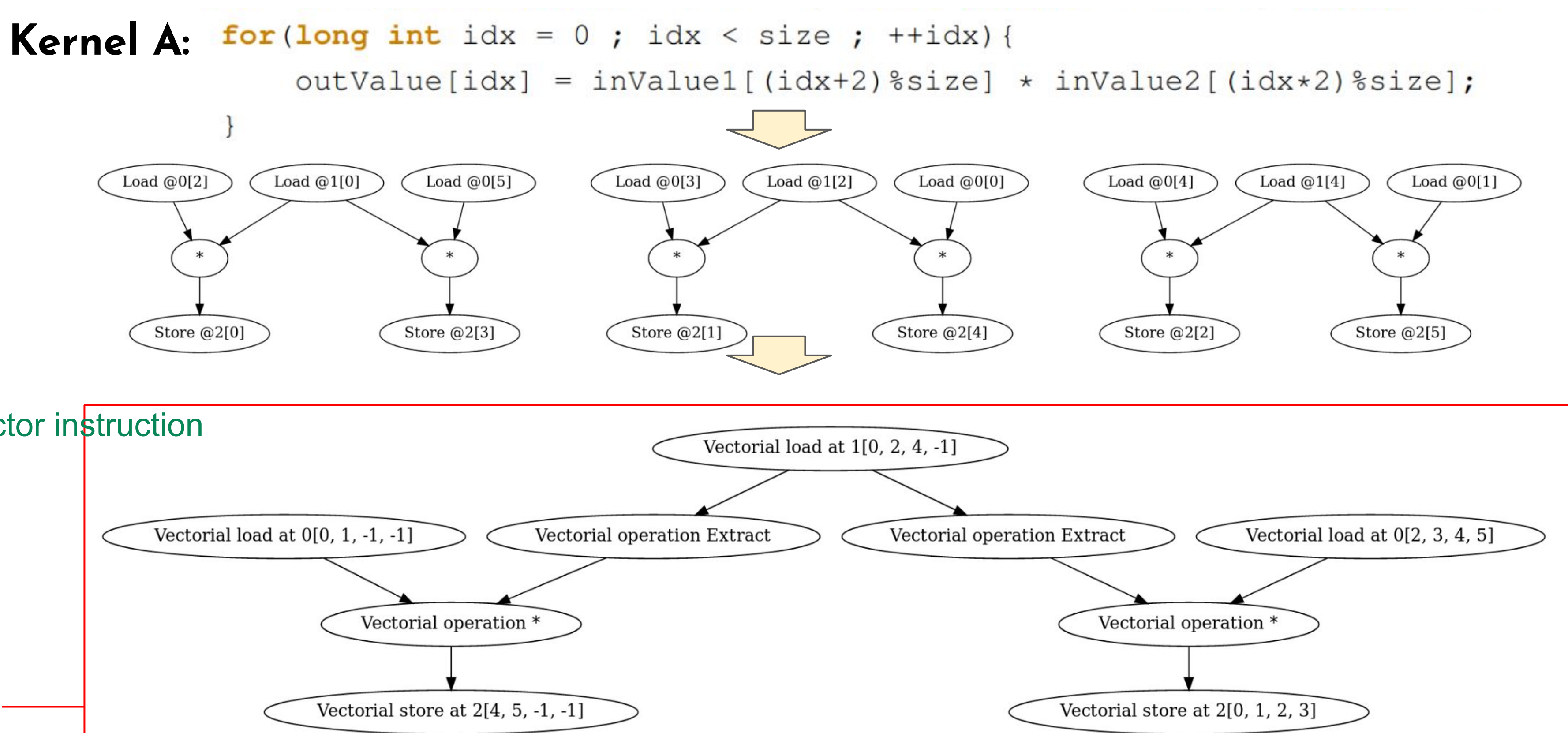




Example of vector instruction graph

Kernel A:

```
for(long int idx = 0 ; idx < size ; ++idx){
    outValue[idx] = inValue1[(idx+2)%size] * inValue2[(idx*2)%size];
}
```



Example of the vectorized code

```
m512d ld_0 = _mm512_castpd128_pd512(_mm_loadu_pd( tab1+8));
m512d ld_1 = _mm512_mask_i64gather_pd( _mm512_setzero_pd(), 15,  _mm512_set_epi64(6, 4, 2, 0),tab1, sizeof(double));
m512d tmp_0 = _mm512_mask_permutexvar_pd(_mm512_setzero_pd(), 1, _mm512_set_epi64(-1, -1, -1, 0),ld_0);
m512d tmp_1 = _mm512_mask_permutexvar_pd(_mm512_setzero_pd(), 14, _mm512_set_epi64(2, 1, 0, -1),ld_1);
m512d tmp_2 = _mm512_mask_permutexvar_pd(_mm512_setzero_pd(), 2, _mm512_set_epi64(-1, -1, 0, -1),ld_0);
m512d tmp_3 = _mm512_mask_permutexvar_pd(_mm512_setzero_pd(), 1, _mm512_set_epi64(-1, -1, -1, 3),ld_1);
m512d ld_2 = _mm512_loadu_pd( tab0+2);
m512d tmp_4 = _mm512_or_pd(tmp_1, tmp_0);
m512d ld_3 = _mm512_loadu_pd( tab0+6);
m512d tmp_5 = _mm512_or_pd(tmp_3, tmp_2);
m512d ld_4 = _mm512_castpd256_pd512(_mm256_loadu_pd( tab0+0));
m512d tmp_6 = _mm512_mul_pd(ld_3, tmp_4);
m512d tmp_7 = _mm512_mul_pd(ld_2, ld_1);
m512d tmp_8 = _mm512_mul_pd(ld_4, tmp_5);
_mm256_storeu_pd( tab2+8, _mm512_castpd512_pd256( tmp_8));
_mm512_storeu_pd( tab2+4, tmp_6);
_mm512_storeu_pd( tab2+0, tmp_7);
```

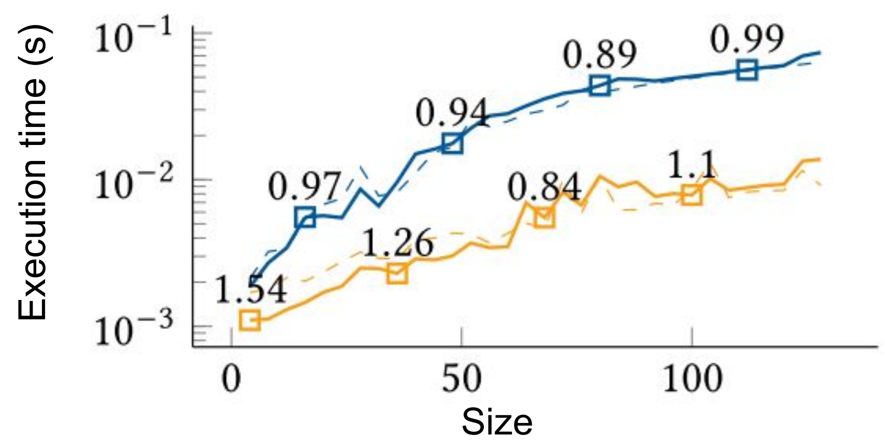
Kernel A with VECTOR SIZE = 4

Speedup vs Gcc: Intel-AVX512 and ARM-SVE

Regular kernels

- GNU compiler 10.2.0
- 512-bit AVX
- SIMD vector size 8 double floating-point values

Results Kernel vector addition
dest[i] = src0[i] + src1[i]



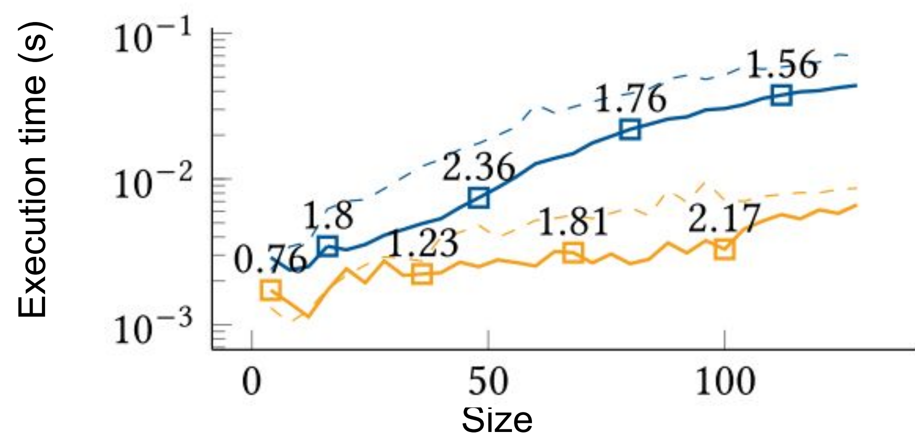
Hayfa Tayeb, Ludovic Paillat, and Bérenger Bramas. 2023. Autovesk: Automatic Vectorized Code Generation from Unstructured Static Kernels Using Graph Transformations. ACM Trans. Archit. Code Optim. 21, 1, Article 4 (March 2024), 25 pages.

Speedup vs Gcc: Intel-AVX512 and ARM-SVE

Regular kernels

- GNU compiler 10.2.0
- 512-bit AVX
- SIMD vector size 8 double floating-point values

Results Kernel B (reduction)
dest += src0[i] * src1[i]

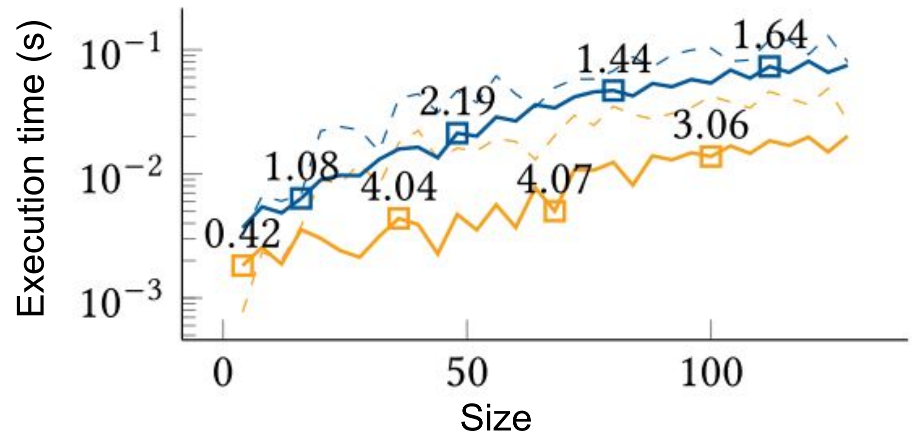


Hayfa Tayeb, Ludovic Paillat, and Bérenger Bramas. 2023. Autovesk: Automatic Vectorized Code Generation from Unstructured Static Kernels Using Graph Transformations. ACM Trans. Archit. Code Optim. 21, 1, Article 4 (March 2024), 25 pages.

Speedup vs Gcc: Intel-AVX512 and ARM-SVE
Irregular kernels

- GNU compiler 10.2.0
- 512-bit AVX
- SIMD vector size 8 double floating-point values

Results Kernel (reduction)
dest += src0[random(i)] * src1[i]



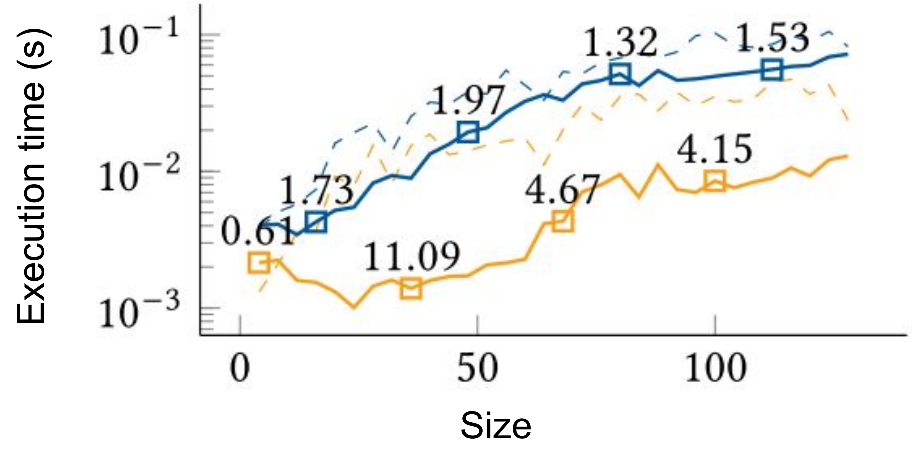
Hayfa Tayeb, Ludovic Paillat, and Bérenger Bramas. 2023. Autovesk: Automatic Vectorized Code Generation from Unstructured Static Kernels Using Graph Transformations. ACM Trans. Archit. Code Optim. 21, 1, Article 4 (March 2024), 25 pages.

Speedup vs Gcc: Intel-AVX512 and ARM-SVE

Irregular kernels

- GNU compiler 10.2.0
- 512-bit AVX
- SIMD vector size 8 double floating-point values

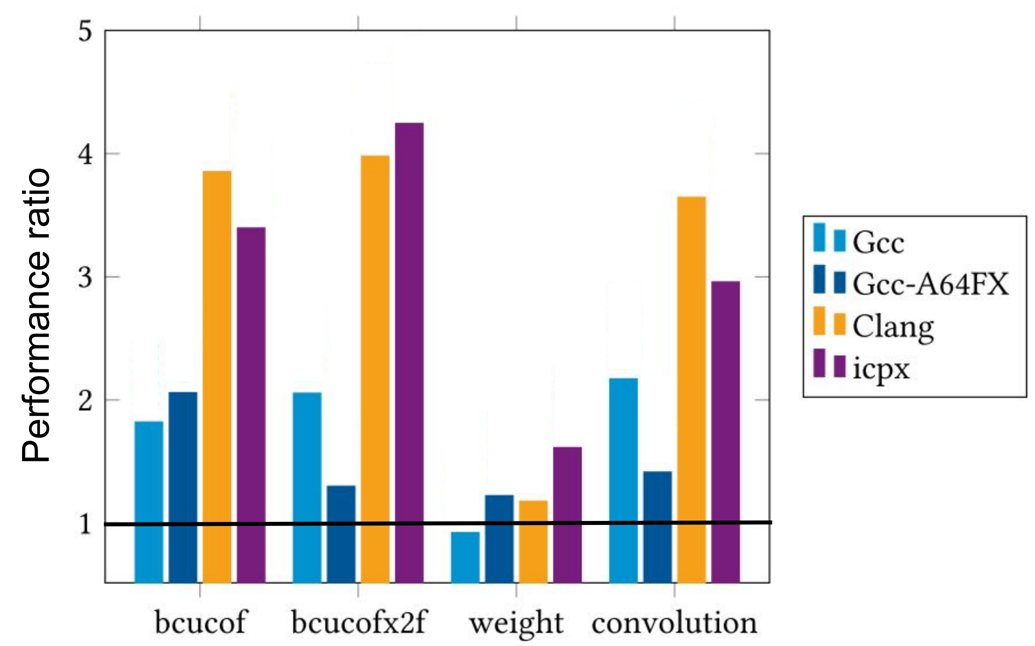
Results Kernel A
dest[i] += src0[shift(i)] * src1[shift(i)]



Hayfa Tayeb, Ludovic Paillat, and Bérenger Bramas. 2023. Autovesk: Automatic Vectorized Code Generation from Unstructured Static Kernels Using Graph Transformations. ACM Trans. Archit. Code Optim. 21, 1, Article 4 (March 2024), 25 pages.

Performance of Autovesk
Kernels from real applications

- **bcucof** routine computes a table for bi-cubic interpolation
- **bcucofx2f**: Two consecutive calls to bcucof on different data
- **weight** routine computes differentiation matrices for pseudo-spectral collocation
- **convolution**: Discrete convolution



Hayfa Tayeb, Ludovic Paillat, and Bérenger Bramas. 2023. Autovesk: Automatic Vectorized Code Generation from Unstructured Static Kernels Using Graph Transformations. ACM Trans. Archit. Code Optim. 21, 1, Article 4 (March 2024), 25 pages.

Perspectives

- Support branches (dynamic)

- Transform any loop into a repetition of a static kernel
  - And optimize the static kernel with Autovesk
  - Similar to unrolling

Hayfa Tayeb, Ludovic Paillat, and Bérenger Bramas. 2023. Autovesk: Automatic Vectorized Code Generation from Unstructured Static Kernels Using Graph Transformations. ACM Trans. Archit. Code Optim. 21, 1, Article 4 (March 2024), 25 pages.

Thanks!
Questions?