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

Hayfa TAYEB¹ ², Ludovic PAILLAT³, Bérenger BRAMAS² 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.
# Existing auto-vectorization techniques

## Regular applications

<table>
<thead>
<tr>
<th>Modern compilers (GCC, Clang, Intel)</th>
<th>Specialized compilers</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Loop vectorization:</strong> data-level parallelism</td>
<td>Polyhedral compilers: affine loops</td>
</tr>
<tr>
<td><strong>Superword Level Parallelism (SLP):</strong> Instruction-oriented vectorization</td>
<td></td>
</tr>
</tbody>
</table>

## Irregular applications

*Irregular applications* (graph algorithms, particle simulation codes, sparse matrix codes)

<table>
<thead>
<tr>
<th>Conflict masking: resolving data conflicts in SIMD vectors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inspector/executor transformations: Sparse Polyhedral Framework (SPF): non-affine array accesses</td>
</tr>
<tr>
<td>Non-SIMD vector instructions: VeGen framework</td>
</tr>
</tbody>
</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:  
```c
for(long int idx = 0 ; idx < size ; ++idx){
    outValue[idx] = inValue1[(idx+2)%size] * inValue2[(idx*2)%size];
}
```

Load `inValue1`  
Load `inValue2`  
Operations  
Store `outValue`  

SPOILER!  

Execution times  
Speedups  

- 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
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

Static kernel

```cpp
static inline ...
kernel_scalar(.. const int n){
    for(int i = 0; i < n; i++){
        for(int j = 0; j < n; j++){
            s[i] += ...;
        }
    }
}
```

Automatic vectorization tool Autovesk

Vectorized static kernel

```cpp
__m512d ld_0 = _mm512_loadu_pd( tab0+0);
...
__m512d tmp_0 = _mm512_add_pd(ld_0, ld);
...
_mm512_storeu_pd(...);
```
The 1st and last stages are not needed if Autovesk is integrated into an existing compiler.
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;
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
Commutative operations and reductions
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:

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

**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

Array 0

0 1 2 3 4 5
1 load 1 load
1 load 1 load

Array 1

0 1 2 4
1 load

Array 2

0 1 2 3 4 5
1 store 1 store
1 store 1 store
Example: Split the operation groups
Using the Matrix of neighbours to split the group

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];
}

Vectorial load at 1[0, 2, 4, -1]
- Vectorial load at 0[0, 1, -1, -1]
- Vectorial operation Extract
- Vectorial operation *
- Vectorial store at 2[4, 5, -1, -1]

Vectorial load at 0[2, 3, 4, 5]
- Vectorial operation *
- Vectorial store at 2[0, 1, 2, 3]
Example of the vectorized code

```
例示的向量化代码

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_castpd256_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]
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]
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]
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)]

--- GCC autovectorization —- Autovesk  INTEL-AVX512  ARM-SVE

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
Perspectives

- Support branches (dynamic)
- Transform any loop into a repetition of a static kernel
  - And optimize the static kernel with Autovesk
  - Similar to unrolling
Thanks!

Questions?