General SIMD Vectorization

Florian Wende

Zuse Institute Berlin
Vectorization is nothing totally new!

Not so long ago, codes have been heavily optimized for vector processing.

Today: revival of vector processing as a 3rd layer of parallelism.
SIMD vectorization today

CPU vendors increasingly draw on SIMD vectorization to push the single-chip compute performance towards multiple TFLOPs

<table>
<thead>
<tr>
<th>SIMD vector lanes</th>
<th>potential gain over scalar execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSE[2,3,4]</td>
<td>2x</td>
</tr>
<tr>
<td>AVX[2]</td>
<td>4x</td>
</tr>
<tr>
<td>KNCNI, AVX-512</td>
<td>8x</td>
</tr>
</tbody>
</table>

Single-Instruction Multiple-Data (SIMD): 
$n$ lanes execute in lockstep on multiple words, e.g., 8 “ADD”s at once
SIMD vectorization today: why is it relevant?

Today’s hardware: one or more SIMD execution unit(s) per CPU core
- up to 112 SIMD lanes on Intel Xeon CPUs
- up to 1152 SIMD lanes on Intel Xeon Phi
SIMD vectorization today: why is it relevant?

Today’s hardware: one or more SIMD execution unit(s) per CPU core
- up to 112 SIMD lanes on Intel Xeon CPUs
- up to 1152 SIMD lanes on Intel Xeon Phi

Amdahl’s law for SIMD
- AVX-512: approx. 80% of your code must vectorize to get 50% peak

Intel Xeon Phi 7290 (KNL): 72 CPU cores, AVX-512
- $\text{peak}_{\text{SIMD,32-bit}} = 5990 \text{ GFLOPs} = (72 \cdot 2 \cdot 16 \cdot 2 \cdot 1.3)$
- $\text{peak}_{\text{no-SIMD,32-bit}} = 375 \text{ GFLOPs}$
SIMD vectorization

The basic idea is to exploit data-level parallelism: process multiple computer words at once by applying the same operation

- SIMD vector instruction set + word size defines the number of “parallel” executions
- “execution streams” are mapped to SIMD lanes
- program state is shared across SIMD lanes

```c
for (int i = 0; i < n; ++i)  
y[i] = log(x[i]);

for (int i = 0; i < n; i += VL)  
  for (int ii = 0; ii < VL; ++ii)  
    y[i + ii] = log(x[i + ii]);
```
The basic idea is to exploit data-level parallelism: process multiple computer words at once by applying the same operation

- SIMD vector instruction set + word size defines the number of “parallel” executions
- “execution streams” are mapped to SIMD lanes
- program state is shared across SIMD lanes

```c
for (int i = 0; i < n; ++i)
    y[i] = log(x[i]);

for (int i = 0; i < n; i += VL)
    for (int ii = 0; ii < VL; ++ii)
        y[i + ii] = log(x[i + ii]);

for (int i = 0; i < n; i += VL)
    y[i] = simd_log(x[i]);
```
The basic idea is to exploit data-level parallelism: process multiple computer words at once by applying the same operation

- SIMD vector instruction set + word size defines the number of “parallel” executions
- “execution streams” are mapped to SIMD lanes
- program state is shared across SIMD lanes

```
for (int i = 0; i < n; i += VL)
  for (int ii = 0; ii < VL; ++ii)
    y[i + ii] = log(x[i + ii]);

for (int i = 0; i < n; i += VL)
  y[i] = simd_log(x[i]);
```

**SIMD processing ≠ vector processing**  
(parallelism vs. pipelining)
SIMD vectorization: what can be vectorized?

Candidates: loops and code blocks applying the same operation to different data!

However, data dependences might prevent vectorization

- write-after-read (anti dependence)
- write-after-write (output dependence)
- read-after-write (flow dependence)

```
  do i = 8, N
  a(i) = a(i + X) + C
```

SIMD execution with vector length (VL) 2

```
  a(8) = a(8 + X) + C
  a(9) = a(9 + X) + C
  a(10) = a(10 + X) + C
  a(11) = a(11 + X) + C
  a(12) = a(12 + X) + C
  a(13) = a(13 + X) + C
  a(14) = a(14 + X) + C
  a(15) = a(15 + X) + C
  a(16) = a(16 + X) + C
  ..
```

https://www.nersc.gov/users/computational-systems/edison/programming/vectorization
SIMD vectorization: what can be vectorized?

Candidates: loops and code blocks applying the same operation to different data!

However, data dependences might prevent vectorization

- write-after-read (anti dependence)
- write-after-write (output dependence)
- read-after-write (flow dependence)

```
do i = 8, N
  a(i) = a(i + X) + C
```

SIMD execution with vector length (VL) 2

<table>
<thead>
<tr>
<th>i</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>a(8) = a(8 + X) + C</td>
</tr>
<tr>
<td>9</td>
<td>a(9) = a(9 + X) + C</td>
</tr>
<tr>
<td>10</td>
<td>a(10) = a(10 + X) + C</td>
</tr>
<tr>
<td>11</td>
<td>a(11) = a(11 + X) + C</td>
</tr>
<tr>
<td>12</td>
<td>a(12) = a(12 + X) + C</td>
</tr>
<tr>
<td>13</td>
<td>a(13) = a(13 + X) + C</td>
</tr>
<tr>
<td>14</td>
<td>a(14) = a(14 + X) + C</td>
</tr>
<tr>
<td>15</td>
<td>a(15) = a(15 + X) + C</td>
</tr>
<tr>
<td>16</td>
<td>a(16) = a(16 + X) + C</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Read-after-write dependence

```
X = -1
```

[https://www.nersc.gov/users/computational-systems/edison/programming/vectorization](https://www.nersc.gov/users/computational-systems/edison/programming/vectorization)
SIMD vectorization: what can be vectorized?

Candidates: loops and code blocks applying the same operation to different data!

However, data dependences might prevent vectorization

- write-after-read (anti dependence)
- write-after-write (output dependence)
- read-after-write (flow dependence)

do i = 8, N
a(i) = a(i + X) + C

pseudo vector version: VL 2

is it vectorizable?

do i = 8, N, 2
{a(i), a(i + 1)} = {a(i + X), a(i + X + 1)} + {C, C}

case 1: X < 0 and |X| < VL
read-after-write dependence
not/partially vectorizable

case 2: X > 0 and X < VL
write-after-read dependence
vectorizable

case 3: |X| ≥ VL
no real dependence
vectorizable

https://www.nersc.gov/users/computational-systems/edison/programming/vectorization
SIMD vectorization: what can be vectorized?

Candidates: loops and code blocks applying the same operation to different data!

However, data dependences might prevent vectorization

- write-after-read (anti dependence)
- write-after-write (output dependence)
- read-after-write (flow dependence)

\[
\begin{align*}
\text{do } & i = 8, N \\
\quad & a(i) = a(i + X) + C
\end{align*}
\]

pseudo vector version: VL 2

\[
\begin{align*}
\text{do } & i = 8, N, 2 \\
\quad & \{a(i), a(i + 1)\} = \{a(i + X), a(i + X + 1)\} + \{C, C\}
\end{align*}
\]

\[
\begin{array}{|c|c|c|c|}
\hline
n & a(n) input & a(n) scalar & a(n) vector \\
\hline
7 & 2 & 2 & 2 \\
8 & 5 & 3 & 3 \\
9 & 4 & 4 & 6 \\
10 & 2 & 5 & 7 \\
11 & 9 & 6 & 3 \\
\vdots & \vdots & \vdots & \vdots \\
\hline
\end{array}
\]

https://www.nersc.gov/users/computational-systems/edison/programming/vectorization
SIMD vectorization: how to?

Different levels to address SIMD vectorization in your code

- leave it to the compiler entirely: auto-vectorizer
- explicit vectorization via compiler directives

- SIMD intrinsics
- (inline) assembly

Vector Modules & Libraries

C/C++

Fortran

high-level

low-level
## SIMD vectorization: how to?

<table>
<thead>
<tr>
<th>Means / Techniques</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Auto-vectorization</strong></td>
</tr>
<tr>
<td>just compile and leave everything to the compiler (usually with <code>-O3</code>)</td>
</tr>
<tr>
<td><strong>Explicit vectorization</strong></td>
</tr>
<tr>
<td>use compiler directives (e.g., <code>#pragma..</code>) to give hints to the compiler</td>
</tr>
<tr>
<td><strong>Manual vectorization</strong></td>
</tr>
<tr>
<td>do it yourself (e.g., SIMD intrinsics, assembly)</td>
</tr>
</tbody>
</table>
SIMD vectorization: how to?

<table>
<thead>
<tr>
<th>Method</th>
<th>Code Invasiveness</th>
<th>Required Skills</th>
<th>Prospect of Success</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auto-vectorization</td>
<td>none</td>
<td>none</td>
<td>low</td>
</tr>
<tr>
<td>Explicit vectorization</td>
<td>low</td>
<td>medium</td>
<td>depends</td>
</tr>
<tr>
<td>Manual vectorization</td>
<td>high</td>
<td>expert</td>
<td>high</td>
</tr>
</tbody>
</table>
Explicit SIMD vectorization

Use compiler directives to give hints to the compiler

- vendor-specific

  #pragma ivdep             Intel
  #pragma GCC ivdep        GNU
  #pragma _CRI ivdep       Cray
  #pragma simd             Intel
  #pragma vector           Intel

  ..
  for (int i = 0; i < n; ++i) {
    ..
  }

  not necessarily the same meaning
Explicit SIMD vectorization

Use compiler directives to give hints to the compiler

- **vendor-specific**
  - `#pragma ivdep` (Intel)
  - `#pragma GCC ivdep` (GNU)
  - `#pragma _CRI ivdep` (Cray)
  - `#pragma simd` (Intel)
  - `#pragma vector` (Intel)

  ```c
  for (int i = 0; i < n; ++i) {
    ...
  }
  ```

  **do not use them any longer, unless you have serious reasons!**

- **OpenMP 4 SIMD constructs**
  - expressiveness, portability, unified approach, languages: C/C++/Fortran
  - → separate talk!
SIMD vectorization with intrinsics (C/C++ only)

If loop structures are (very) complex, explicit vectorization might not result in effective SIMD code: be more specific with SIMD intrinsics!

- means to explicitly tell the compiler which SIMD instruction to use
- its use might interfere with other compiler optimizations
- you definitely should know what you are doing
SIMD vectorization with intrinsics (C/C++ only)

If loop structures are (very) complex, explicit vectorization might not result in effective SIMD code: be more specific with SIMD intrinsics!

- means to explicitly tell the compiler which SIMD instruction to use
- its use might interfere with other compiler optimizations
- you definitely should know what you are doing

```c
#include <immintrin.h>

for (int i = 0; i < n; ++i)
    if (x[i] > 1.0)
        y[i] = log(x[i]);
    else
        y[i] = 1.0;

for (int i = 0; i < n; i += 8) {
    __m512d vx = _mm512_loadu_pd(&x[i]);
    __m512d one = _mm512_set1_pd(1.0);
    __mmask8 m = _mm512_cmp_pd_mask(vx, one, _CMP_GT_OS);
    __m512d vy = _mm512_mask_log_pd(one, m, vx);
    _mm512_storeu_pd(&y[i], vy);
}
```
SIMD vectorization with intrinsics (C/C++ only)

If loop structures are (very) complex, explicit vectorization might not result in effective SIMD code: be more specific with SIMD intrinsics!

- means to explicitly tell the compiler which SIMD instruction to use
- its use might interfere with other compiler optimizations
- you definitely should know what you are doing

```c
for (int i = 0; i < n; ++i)
  if (x[i] > 1.0)
    y[i] = log(x[i]);
else
  y[i] = 1.0;

#include <immintrin.h>
for (int i = 0; i < n; i += 8) {
  __m512d vx = _mm512_loadu_pd(&x[i]);
  __m512d one = _mm512_set1_pd(1.0);
  __mmask8 m = _mm512_cmp_pd_mask(vx, one, _CMP_GT_OS);
  __m512d vy = _mm512_mask_log_pd(one, m, vx);
  _mm512_storeu_pd(&y[i], vy);
}
```

- expressiveness
- you cannot be more explicit
- SIMD performance guaranteed in most cases

- hard to read/maintain
- platform specific
- good luck with bug fixing 😊
SIMD vectorization with intrinsics (C/C++ only)

Multiple SIMD instruction sets, one code!

// simd.h
#include <immintrin.h>

#if defined(__AVX512F__)
    #define SIMD_512
    #include <simd_macros_512.h>
#elif defined(__AVX__)
    #define SIMD_256
    #include <simd_macros_256.h>
#elif defined(..)
    ..
#endif

// simd_macros_512.h
#define vecd __m512d
#define vecf __m512
..
#define vec_add_pd(x1, x2) _mm512_add_pd(x1, x2)
..

// simd_macros_256.h
#define vecd __m256d
#define vecf __m256
..
#define vec_add_pd(x1, x2) _mm256_add_pd(x1, x2)
..
C++ SIMD classes

Most implementations wrap SIMD intrinsics to achieve
- code portability
- more comfortable coding
- increased readability of the code

```cpp
for (int i = 0; i < n; ++i)
    y[i] = 2.0 * x[i];

vec_real_t* vx = (vec_real_t*) x[0];
vec_real_t* vy = (vec_real_t*) y[0];
for (int i = 0; i < n; i += vec_real_t::size)
    vy[i] = 2.0 * vx[i];
```
C++ SIMD classes

Most implementations wrap SIMD intrinsics to achieve
- code portability
- more comfortable coding
- increased readability of the code

Different implementations, e.g.

VC

Agner Fog’s vector class library (VCL)
http://www.agner.org/optimize/#vectorclass

UMESIMD
https://github.com/edanor/umesimd