SIMD Vectorization with High-Level Vectors

Florian Wende

Zuse Institute Berlin
Part I – SIMD Vectorization
SIMD Vectorization

Single-Instruction Multiple-Data execution model

- means to gain the arithmetic throughput with only a bit more logic on the chip
- one instruction applied to VL (vector length) computer words at once
  - shared program counter: control flow can spoil the effectiveness of SIMD

```c
for (i = 0; i < N; ++i)
  if (p[i]) y[i] = log(x[i]);
  else y[i] = exp(x[i]);
```

```c
for (i = 0; i < N; i += 8)
  if (m ← p[i]) y[i] = vlog_mask(y[i], m, x[i]);
  else y[i] = vexp_mask(y[i], ~m, x[i]);
```

4 times faster execution with SIMD
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*

\[
\text{peak}_{\text{SIMD,32-bit}} = 5990 \text{ GFLOPs} \quad (= 72 \cdot 2 \cdot 16 \cdot 2 \cdot 1.3)
\]

\[
\text{peak}_{\text{no-SIMD,32-bit}} = 375 \text{ GFLOPs}
\]
How to approach SIMD?

- use optimized libraries
- user code optimization – loops .. functions .. control flow .. data layout ..
  - auto-vectorization
  - explicit vectorization through (OpenMP) compiler directives
  - manual vectorization
SIMD Vectorization

How to approach SIMD?

- use optimized libraries
- user code optimization – loops .. functions .. control flow .. data layout ..
  - auto-vectorization
  - explicit vectorization through (OpenMP) compiler directives
  - manual vectorization

Florian Wende (wende@zib.de)                                  SIMD Vectorization with High-Level Vectors
How to approach SIMD?

- use optimized libraries
- user code optimization – loops .. functions .. control flow .. data layout ..
  - auto-vectorization
  - explicit vectorization through (OpenMP) compiler directives
  - manual vectorization

Florian Wende (wende@zib.de)  SIMD Vectorization with High-Level Vectors
SIMD Vectorization with OpenMP

Quasi-standard for compiler directive based SIMD vectorization

- loops: (C/C++) \#pragma omp simd (F) !$omp simd
- functions / subroutines: (C/C++) \#pragma omp declare simd (F) !$omp declare simd

- supported by major compilers: GNU, Clang, Intel, Cray
- complements with widely used OpenMP threading (at least in the HPC world)
SIMD Vectorization with OpenMP

Loop constructs: www.openmp.org/mp-documents

```c
#pragma omp simd
for (int i = 0; i < n; ++i)
    y[i] = 2.0 * x[i];

#pragma omp simd simdlen(32)
for (int i = 0; i < n; ++i)
    y[i] = 2.0 * x[i];

#pragma omp simd reduction(+:res)
for (int i = 0; i < n; ++i)
    res += 2.0 * x[i];
```

```c
#pragma omp simd aligned(x, y)
for (int i = 0; i < n; ++i)
    y[i] = 2.0 * x[i];

#pragma omp simd safelen(3)
for (int i = 3; i < n; ++i)
    y[i] = y[i - 3] + 2.0 * x[i];

#pragma omp simd collapse(2)
for (int j = 0; j < m; ++j)
    for (int i = 0; i < n; ++i)
        y[j][i] = 2.0 * x[i] * x[j];
```
SIMD Vectorization with OpenMP

SIMD function/subroutine constructs:  [www.openmp.org/mp-documents](www.openmp.org/mp-documents)

```c
#pragma omp declare simd
double foo(double x, double c) {
  return c + sqrt(x);
}

#pragma omp simd
for (int i = 0; i < n; ++i)
  y[i] = foo(x[i], c);
```
SIMD Vectorization with OpenMP

SIMD function/subroutine constructs:  www.openmp.org/mp-documents

```c
#pragma omp declare simd [uniform(c),..]
double foo(double x, double c) {
    return c + sqrt(x);
}
```

```c
#pragma omp simd
for (int i = 0; i < n; ++i)
    y[i] = foo(x[i], c);
```
SIMD Vectorization with OpenMP

SIMD function/subroutine constructs:  [www.openmp.org/mp-documents](http://www.openmp.org/mp-documents)

```c
#pragma omp declare simd [uniform(c), linear(..), simdlen(..), notinbranch,..]
double foo(double x, double c) {
    return c + sqrt(x);
}
```

```c
#pragma omp simd
for (int i = 0; i < n; ++i)
    y[i] = foo(x[i], c);
```
SIMD Vectorization with OpenMP

+ loop and function/subroutine SIMD constructs
+ portable across compilers and platforms
+ not that code invasive

- all control delegated to the compiler
- how to switch between scalar and vector case?
- debugging can be hard
High-Level Vectors

1. define generic vector data types

#define VL64 8

typedef struct alignas(64) {
  double x[VL64];
} vec_double_t;

typedef struct {
  bool x[VL64];
} mask_double_t;
1. define generic vector data types
2. LOOP: replace scalars by vectors

```c
#define VL64 8

typedef struct alignas(64) {
    double x[VL64];
} vec_double_t;

typedef struct {
    bool x[VL64];
} mask_double_t;

for (int i = 0; i < n; i += VL64) {
    vy = 2.0 * vx;
    vfoo(vx, vy);
}
```
High-Level Vectors

1. define generic vector data types
   
2. LOOP: replace scalars by vectors + loop over their elements for processing

```c
#define VL64 8

typedef struct alignas(64) {
    double x[VL64];
} vec_double_t;

typedef struct {
    bool x[VL64];
} mask_double_t;

for (int i = 0; i < n; i += VL64) {
    for (int ii = 0; ii < VL64; ++ii)
        vy.x[ii] = 2.0 * vx.x[ii];
    vfoo(vx, vy, );
}
```
High-Level Vectors

1. define generic vector data types
2. LOOP: replace scalars by vectors + loop over their elements for processing
3. LOOP: add vector packing/unpacking + mask creation

```
#define VL64 8

typedef struct alignas(64) {
    double x[VL64];
} vec_double_t;

typedef struct {
    bool x[VL64];
} mask_double_t;
```

```
for (int i = 0; i < n; i += VL64) {
    for (int ii = 0; ii < VL64; ++ii) {
        if (mask.x[ii]) vx.x[ii] = x[i + ii];
    }
    for (int ii = 0; ii < VL64; ++ii)
        if (mask.x[ii]) y[i + ii] = vy.x[ii];
}
```

Florian Wende (wende@zib.de)
High-Level Vectors

1. define generic vector data types
   2. LOOP: replace scalars by vectors + loop over their elements for processing
   3. LOOP: add vector packing/unpacking + mask creation
   4. LOOP: enforce vectorization through OpenMP SIMD loop construct

#define VL64 8

typedef struct alignas(64) {
    double x[VL64];
} vec_double_t;

typedef struct {
    bool x[VL64];
} mask_double_t;

for (int i = 0; i < n; i += VL64) {
    for (int ii = 0; ii < VL64; ++ii) {
        mask.x[ii] = (i + ii) < n;
        if (mask.x[ii]) vx.x[ii] = x[i + ii];
    }
    #pragma omp simd
    for (int ii = 0; ii < VL64; ++ii) {
        vy.x[ii] = 2.0 * vx.x[ii];
        vfoo(vx, vy, mask);
        if (mask.x[ii]) y[i + ii] = vy.x[ii];
    }
}
void vfoo(const vec_double_t& x, vec_double_t& y, mask_double_t& mask0) {
    vec_double_t tmp;
    mask_double_t mask1 = mask0;
    bool true_for_any = false;

    #pragma omp simd reduction(|:true_for_any)
    for (int ii = 0; ii < VL64; ++ii) {
        tmp.x[ii] = exp(x.x[ii]);
        if (mask1.x[ii] && tmp.x[ii] > 5.0) {
            true_for_any = true;
        } else {
            mask1.x[ii] = false;
        }
    }

    if (true_for_any) vbar(tmp, mask1);

    #pragma omp simd
    for (int ii = 0; ii < VL64; ++ii)
        if (mask0.x[ii]) y.x[ii] = tmp.x[ii];
}
void vfoo(const vec_double_t& x, vec_double_t& y, mask_double_t& mask0) {
    vec_double_t tmp;
    mask_double_t mask1 = mask0;
    bool true_for_any = false;

    #pragma omp simd reduction(+:true_for_any)
    for (int ii = 0; ii < VL64; ++ii) {
        tmp.x[ii] = exp(x.x[ii]);
        if (mask1.x[ii] && tmp.x[ii] > 5.0) {
            true_for_any = true;
        } else {
            mask1.x[ii] = false;
        }
    }

    if (true_for_any) vbar(tmp, mask1);

    #pragma omp simd
    for (int ii = 0; ii < VL64; ++ii)
        if (mask0.x[ii]) y.x[ii] = tmp.x[ii];
}
void vfoo(const vec_double_t& x, vec_double_t& y, mask_double_t& mask0) {
        vec_double_t tmp;
        mask_double_t mask1 = mask0;
        bool true_for_any = false;

        #pragma omp simd reduction(|:true_for_any)
        for (int ii = 0; ii < VL64; ++ii) {
            tmp.x[ii] = exp(x.x[ii]);
            if (mask1.x[ii] && tmp.x[ii] > 5.0) {
                true_for_any = true;
            } else {
                mask1.x[ii] = false;
            }
        }

        for (int ii = 1; ii < VL64; ++ii)
            tmp.x[ii] += mask1.x[ii] ? tmp.x[ii - 1] : 0.0;

        #pragma omp simd
        for (int ii = 0; ii < VL64; ++ii)
            if (mask0.x[ii]) y.x[ii] = tmp.x[ii];
    }
Why should I use this approach?

- it is a generic approach using OpenMP 4: portability!
- it draws on vectorizing standard loops: many compilers can handle this
- you can easily mix vector and scalar code in loop and function nests
- you have the vector code at hand: beneficial for debugging and performance tuning
- natural way to writing SIMD functions
Part II – Case Studies
High-Level Vectors in VASP

Vienna Ab-initio Simulation Package (VASP)
- widely used electronic structure code
- Fortran90 code base
- modernization within ZIB’s IPCC activities

Solves for the Schrödinger Equation from first principals: $H\psi = E\psi$

Approximations
- density functional theory (DFT)
- Hartree-Fock/DFT-HF hybrid functionals
- random-phase-approximation (FW, ACFDT)

Florian Wende (wende@zib.de)
High-Level Vectors in VASP

(1) SIMD vectorization of part of a calling tree relevant to **hybrid functional computation**
- branching
- math function calls
- nested function call hierarchy

Florian Wende (wende@zib.de)

SIMD Vectorization with High-Level Vectors
(1) SIMD vectorization of part of a calling tree relevant to *hybrid functional computation*

- branching
- math function calls
- nested function call hierarchy

Functionals that have been SIMD vectorized

- 91  Perdew-Wang 91
- PE  Perdew-Burke-Ernzerhof
- RP  Revised Perdew-Burke-Ernzerhof with Pade approximation
- B3  B3LYP
- AM  AM05
- PS  Perdew-Burke-Ernzerhof revised for solids
High-Level Vectors in VASP

(1) Perdew-Burke Ernzerhof functional: different compilers, different platforms
F. Wende et al., Porting VASP from MPI to MPI+OpenMP [SIMD], IWOMP 2017, Stony Brook, NY, USA, LNCS 8766

![Intel Xeon E5-2630v3 (2x8 cores)](chart1.png)

![Intel Xeon Phi 7210 (64 cores)](chart2.png)

Xeon Phi: quad-flat mode, all data in MCDRAM
High-Level Vectors in VASP

(1) other functionals: high-level vector version with 2x SIMD width and ifort 18 compiler

Florian Wende (wende@zib.de)  
SIMD Vectorization with High-Level Vectors
High-Level Vectors in VASP

(2) **GW computation in VASP**: integration of the screened two-electron integrals over frequency (Coulomb interaction)

\[
\sigma(\omega) = \int_{-\infty}^{+\infty} \text{kernel}(W) \, d\omega
\]

\[n = 1\]
\[\text{do } i = 1, n_{\omega}\]
\[\quad \text{if } ((i*\text{delta}) > \omega(n+1)) \text{ then}
\]
\[\quad \quad \text{do}
\]
\[\quad \quad \quad n = n + 1
\]
\[\quad \quad \quad \text{if } (\omega(n+1) > (i*\text{delta})) \text{ exit}
\]
\[\quad \quad \text{enddo}
\]
\[\quad \text{endif}
\]
\[\quad \text{tmp = some_func_of}(W(n), W(n+1))
\]
\[\quad ! \text{computation using 'tmp'}
\]
\[\quad \text{endo}
\]
(2) **GW computation in VASP**: integration of the screened two-electron integrals over frequency (Coulomb interaction)

```fortran
n = 1
do i = 1, n_omega
   if ((i*delta) > omega(n+1)) then
      do
         n = n + 1
         if (omega(n+1) > (i*delta)) exit
      enddo
   endif
   tmp = some_func_of(w(n), w(n+1))
   !computation using ‘tmp’
enddo
```

- **n-value computation prevents SIMD vectorization of the i-loop**
- **hotspot computation also not vectorized!**
High-Level Vectors in VASP

(2) **GW computation in VASP**: integration of the screened two-electron integrals over frequency (Coulomb interaction)

```
n = 1
do i = 1, n_omega, VL64
  do ii = 0, min(VL64-1, n_omega-i)
    if (((i+ii)*delta) > omega(n+1)) then
      do
        n = n + 1
        if (omega(n+1) > ((i+ii)*delta)) exit
      enddo
    endif
    vn%x(ii) = n
  enddo
.. enddo
```

prologue: determine all n-values needed to process this chunk
High-Level Vectors in VASP

(2) **GW computation in VASP**: integration of the screened two-electron integrals over frequency (Coulomb interaction)

\[
\begin{align*}
n &= 1 \\
\text{do } i &= 1, n_{\text{omega}}, \text{ VL64} \\
\ldots \\
\text{do } ii &= 0, \min(\text{VL64}-1, n_{\text{omega}}-i) \\
&\quad \text{idx} = v^n\%x(ii) \\
&\quad vtmp\%x(ii) = \text{some_func_of}(w(idx), w(idx+1)) \\
\text{enddo} \\
\$\text{omp simd} \\
\text{do } ii &= 0, \min(\text{VL64}-1, n_{\text{omega}}-i) \\
&\quad \text{tmp} = vtmp\%x(ii) \\
&\quad !\text{computation using ‘tmp’} \\
\text{enddo} \\
\text{enddo}
\end{align*}
\]

SIMD vectorization of the hotspot computation
High-Level Vectors in VASP

(2) **GW computation in VASP**: integration of the screened two-electron integrals over frequency (Coulomb interaction)

![Graphs showing comparison of execution times for Intel Xeon E5-2630v3 and Intel Xeon Phi 7210 with high-level vectors and reference execution times.](image)

Intel Xeon E5-2630v3 (2x8 cores)

- Reference: 50 s
- High-level vectors: 27 s (1.86x speedup)
- High-level vectors (ifort 17u2): 25 s (1.68x speedup)

Intel Xeon Phi 7210 (64 cores)

- Reference: 50 s
- High-level vectors: 10 s (6.90x speedup)
- High-level vectors (ifort 17u2): 8 s (6.47x speedup)

Xeon Phi: quad-flat mode, all data in MCDRAM

Florian Wende (wende@zib.de)
VASP SIMD optimizations by ZIB will be available with any of the upcoming VASP releases

- modified source code is under review already
- coding scheme will be adapted to other parts of the source code
Mandelbrot set escape time algorithm

A complex number $c$ is in the Mandelbrot set if the sequence

$$z_{n+1} = z_n^2 + c, \, z_0 = 0$$

is bounded, that is, if $|z_n| < z_{\text{max}}$ for all $n$
A complex number $c$ is in the Mandelbrot set if the sequence

$$z_{n+1} = z_n^2 + c, \quad z_0 = 0$$

is bounded, that is, if $|z_n| < z_{\text{max}}$ for all $n$.

**Escape time algorithm:** count the number of recursion steps $n$ until either $|z_n| \geq z_{\text{max}}$ or $n \geq n_{\text{max}}$ (then color-code that $n$)

Collaboration of ZIB, RWTH Aachen and ZIH (Dresden) within the IPCC
(O. Krzikalla et al., *Dynamic SIMD Vector Lane Scheduling*, LNCS vol. 9945, 2016)
Mandelbrot set escape time algorithm

Varying workload intensities across SIMD lanes

- use high-level vectors to implement a dynamic pre-scheduling of loop iterations when applying the recursion to a fragment of the complex plane

![Graph showing kernel execution time for Intel Xeon E5-2630v3 (2x8 cores) and Intel Xeon Phi 7210 (64 cores). The graph compares reference, high-level vectors (static), and high-level vectors (dynamic) with execution time in milliseconds.]

Intel Xeon E5-2630v3 (2x8 cores)
- Reference: 22.0 ms
- High-level vectors (static): 5.5 ms
- High-level vectors (dynamic): 4.2 ms

Intel Xeon Phi 7210 (64 cores)
- Reference: 16.3 ms
- High-level vectors (static): 2.1 ms
- High-level vectors (dynamic): 1.8 ms

Xeon Phi: quad-flat mode, all data in MCDRAM

Florian Wende (wende@zib.de)  SIMD Vectorization with High-Level Vectors
Summary

SIMD vectorization is essential to get performance out of today’s / tomorrow’s CPUs
- whole function / program vectorization
- dealing with control flow divergences
- requires restructuring of many parts of the program including data layout adaptation

OpenMP 4 SIMD constructs as the means to enforce vectorization

High-level vectors to increase expressiveness
- generic vector data type definition
- OpenMP 4 SIMD loop construct to process vector elements
- portability: platforms and compilers
Backup
SIMD Vectorization with OpenMP

SIMD function/subroutine constructs:  www.openmp.org/mp-documents

subroutine foo(x, y, c)
$omp declare simd (foo) uniform(c), linear(ref(x, y)),..
   real(8), intent(in) :: x, c
   real(8), intent(out) :: y
   y = c + sqrt(x)
end function foo

$omp simd
do i = 1, n
   call foo(x(i), y(i), c)
endo