Vectorization with Compilers (C/C++)

Georg Zitzlsberger

georg.zitzlsberger@vsb.cz

IT4Innovations
national supercomputing
center

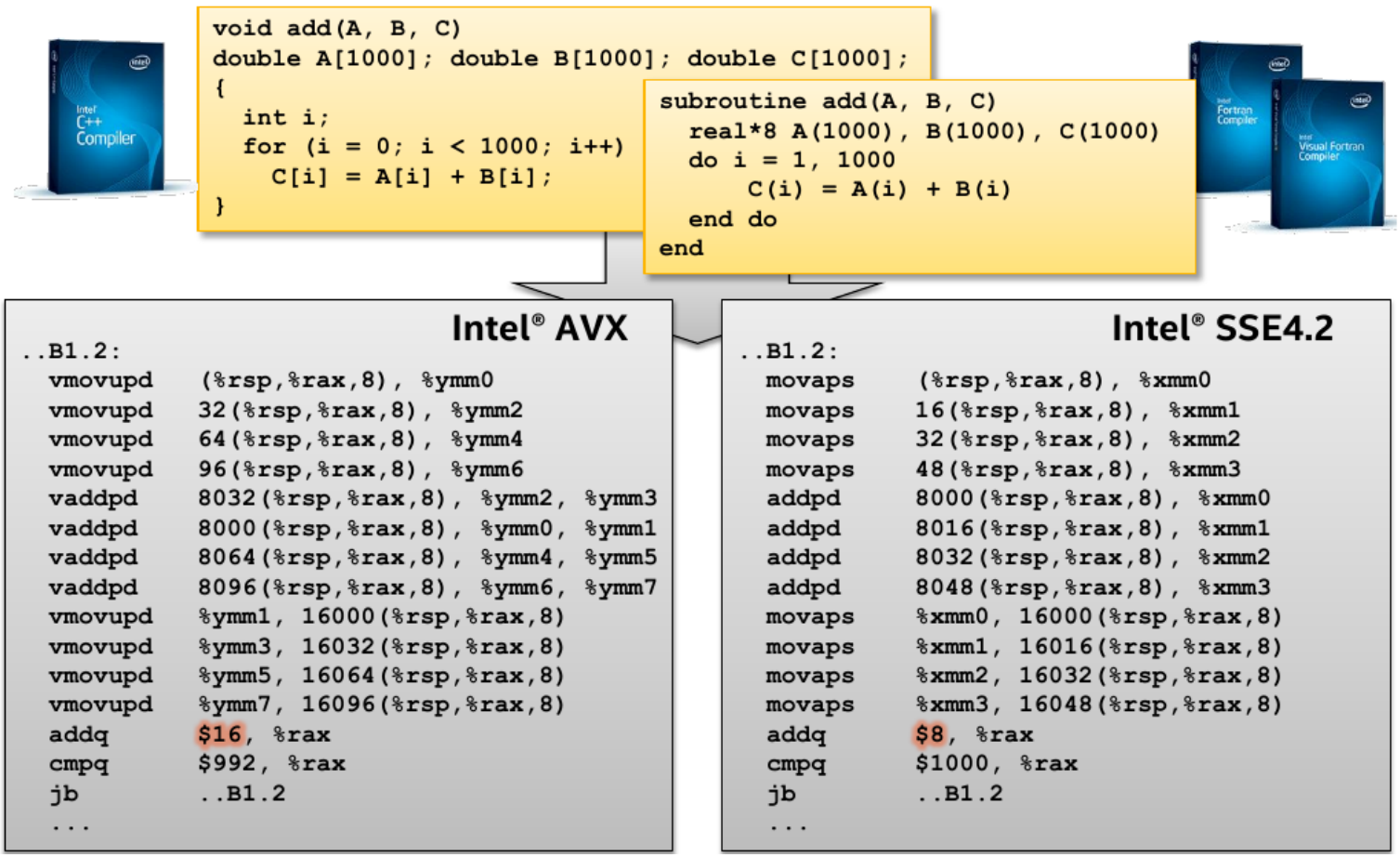
5th of July 2017

Agenda

- How to Vectorize?
- Tell Compiler
- What did the Compiler do?
- More Control on Vectorization
- What Hinders Vectorization?
  - Memory Access Patterns
  - Language Side-Effects
  - Data Dependence
  - Data Dependence - Aliasing
  - Data Dependence - Memory Disambiguation
- Alignment

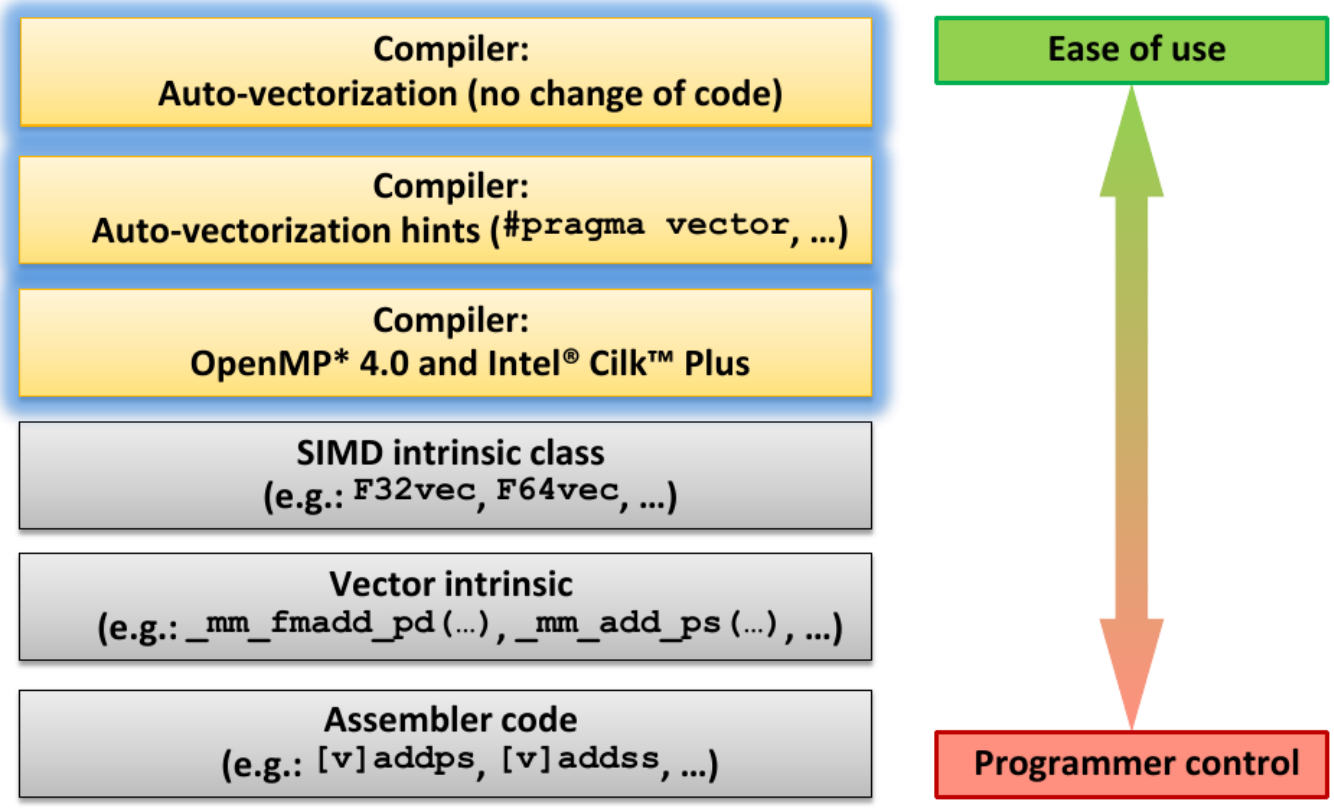
How to Vectorize?

- Use high level language rather than assembler or intrinsics
- Tell compiler to use SIMD
- Switch between different SIMD types
- Help compiler to vectorize



(Image: Intel)

Many Ways to Vectorize



(Image: Intel)

- OpenCL/CUDA
- OpenACC
- VC/VCL/Boost.SIMD
- many more...

Also consider 3rd party libraries already optimized! (e.g. FFTW, Intel MKL, PETSc, . . .)

Tell Compiler

SIMD Feature	ICC Name	GCC/Clang Name
SSE	sse	sse
SSE2	sse2 (default)	sse2
SSE3	sse3	sse3
SSSE3	ssse3	ssse3
SSE4.1	sse4.1	sse4.1
SSE4.2	sse4.2	sse4.2
AVX	avx	avx
AVX2	core-avx2	avx2
MIC	mic	N/A
common AVX512	common-avx512	avx512f, avx512cd
Skylake AVX512	core-avx512	skylake-avx512[1]
KNL AVX512	mic-avx512	knl[2]

[1] avx512f & avx512cd & avx512vl & avx512bw & avx512dq

[2] avx512f & avx512cd & avx512er & avx512pr

Tell Compiler

Use options:

- GCC/Clang:
  `-m...-ftree-vectorize -O2`
  Note that vectorization is not done automatically and also requires minimum `-O2`!
- ICC:
  `-x...` or `-m...`
  Note that `-O2` is default including vectorization but also `-msse2`!

Example:

```
> icc -xcore-avx2 vec.cpp
> icc -mcore-avx2 vec.cpp
> gcc -mavx2 -ftree-vectorize -O2 vec.cpp
```

Tell Compiler

Some more information on options:

- ICC's `-x...` options add test of architecture to `main` routine when linked
  (only works on Intel architecture!)
- ICC supports multiple SIMD versions in same executable
  `-ax<simd1, simd2, ...>`
- ICC supports `-m...` but does not add processor check
- Always specify SIMD because defaults are too conservative (e.g. `-msse2` for ICC)
- Tell GCC/Clang to optimize and use vectorization (off by default)
- ICC has `-fp-model fast` as default; use GCC's `-ffast-math` for comparison

What did the Compiler do?

Modern compilers inform about vectorization:

- ICC `-qopt-report=[0-5]`

  Example:

```
> icpc -xcore-avx2 -qopt-report=5 vec.cpp
icpc: remark #10397: optimization reports are generated in *.optrpt files...
> cat vec.optrpt
...
LOOP BEGIN at vec.cpp(8,5)
   remark #15388: vectorization support: reference c[i] has aligned access
   remark #15388: vectorization support: reference a[i] has aligned access
   remark #15388: vectorization support: reference b[i] has aligned access
   remark #15305: vectorization support: vector length 4
   remark #15399: vectorization support: unroll factor set to 4
   remark #15300: LOOP WAS VECTORIZED
   ...
LOOP END
...
```

  More information can be found here

- GCC `-fopt-info-all`[3]

  Example:

```
> gcc -O2 -ftree-vectorize -mavx2 -fopt-info-all vec.cpp
...
vec.cpp:8:22: note: LOOP VECTORIZED
...
```

[3] `-ftree-vectorizer-verbose` is not supported anymore

More Control on Vectorization

- For GCC/Clang turn off vectorization:
  `-fno-tree-vectorize`
- For ICC, turn off vectorization:
  `-no-vec -no-simd -qno-openmp-simd`
- Turn off vectorization for selected loops (ICC only):
  `#pragma novector`
  Example:

```
#pragma novector
for(int i = 0; i < N; ++i) {
    ...
}
```

What Hinders Vectorization?

Most frequent reasons:

- **Data dependence**
- **Alignment**
- **Unsupported loop structure (language side-effects)**
- **Memory (non-unit stride) accesses**
- Function calls/in-lining
- Non-vectorizable mathematical functions
- Unsupported data types
- Control dependence
- A lot more...

Memory Access Patterns

Memory access patterns:

- Accesses are fastest unit-strided
  (e.g. `a[i]`, `a[i+1]`, `a[i+2]`, `...`)
- Non-unit strided accesses can be overcome if stride is fixed.
  Might require gather/scatter support of architecture but still is slower than unit-stride accesses!
  (e.g. `a[i]`, `a[i+4]`, `a[i+8]`, `...`)
- Random accesses are worst in terms of performance;
  gather/scatter can mitigate too, but still is worst performance!

Memory Access Patterns

Watch out for Arrays of Structures (AoS) accesses, e.g.:

```
struct {
    double x;
    double y;
} point_t;
...
point_t points[N];
for(int i = 0; i < N; ++i) {
    result += (points[i].x + points[i].y) / 2;
}
```

⇒ Transform AoS to Structures of Arrays (SoA), e.g.:

```
struct {
    double x[N];
    double y[N];
} points;
...
for(int i = 0; i < N; ++i) {
    result += (points.x[i] + points.y[i]) / 2;
}
```

- Requires changing data structures
- Sometimes a hybrid AoSoA is good compromise
- Intel SIMD Data Layout Templates (SDLT) can be used (C++ only) ▸ more here

Language Side-Effects

Watch out for side-effects of the language:

- Varying upper bound of loop, e.g.:

```
for(int i = 0; i < get_upper_bound(); ++i) {
    result += (points.x[i] + points.y[i]) / 2;
}
```

  Function has to be invoked with every iteration in worst case

- Global vs. local data, e.g. `stride` defined in other compilation unit:

```
for(int i = 0; i < N; i = i + stride) {
    result += (points.x[i] + points.y[i]) / 2;
}
```

  Does not even help if `stride` is **compile time** constant **in other compilation unit**!
  But not a problem if `stride` is defined (and fixed) in local scope

Data Dependence

How a compiler vectorizes (simplified):
With unrolling the loop...

```
for(int i = 0; i < N; i++) {
    result += (points.x[i] + points.y[i]) / 2;
}
```

the compiler could generate this[4]...

```
for(int i = 0; i < N; i+=4) {
    tmp[0] = (points.x[i+0] + points.y[i+0]) / 2;
    tmp[1] = (points.x[i+1] + points.y[i+1]) / 2;
    tmp[2] = (points.x[i+2] + points.y[i+2]) / 2;
    tmp[3] = (points.x[i+3] + points.y[i+3]) / 2;
    result += reduce_add(tmp[0:4]);
}
```

Imagine vectorization as...

```
for(int i = 0; i < N; i+=4) {
    %*mm0 = {points.x[i+0], points.x[i+1], points.x[i+2], points.x[i+3]}
    %*mm1 = {points.y[i+0], points.y[i+1], points.y[i+2], points.y[i+3]}
    %*mm2 = (%*mm1 + %*mm2) / 2; // element-wise operations
    result += reduce_add(%*mm2);
}
```

With vector registers `*mm[0-2]` of VL 4

[4] Ignoring remainders for simplicity

Data Dependence

- Key theorem for vectorization:
  A loop can be vectorized iff there is no cyclic dependency chain between statements of a loop body.
- For SIMD with limited VL any dependencies outside VL (over registers) are OK
- Data dependencies within VL (within a SIMD vector register) are problematic
  $\Rightarrow$ Vector instructions don't account for dependencies between elements[5]
- Example: Is there a data dependence when vectorized?

```
for(int i = 0; i < N; i++) {
    a[i + 3] = a[i] + 1.0;
}
```

[5] Exception is AVX512-CD

Data Dependence - Aliasing

- C/C++ aliasing:
  Pointers to same data type are assumed to alias (i.e. can have same target address)
- In most cases that's not desired (every pointer typically is supposed to have a different target)
- C/C++ compilers have to assume aliasing for correctness
- Aliasing maps to data dependences with targets overlapping (due to assumed aliasing)
- Example:

```
void vec(double *a, double *b, double *c)
{
    for(int i = 0; i < N; i++) {
        c[i] = a[i] + b[i];
    }
}
```

Data Dependence - Memory Disambiguation

- Using option `-fargument-noalias` turns off pointer aliasing for function arguments
- Affects entire compilation unit (source file) and hence all pointer arguments of all functions
- In case global pointers should be anti-aliased, use `-fargument-noalias-global`
- `-fno-alias` is more aggressive, but also more dangerous
- Always check correctness of application when using those options!

Data Dependence - Memory Disambiguation

- Ignoring assumed dependencies: `#pragma ivdep`
- Works for a loop only
- Compiler will ignore assumed dependencies, but not dependencies it is able to proof
- Has not only an effect on inherent pointer aliasing but also to other semantic
- Example 1:

```
void vec(double *a, double *b, double *c)
{
#pragma ivdep
    for(int i = 0; i < N; i++) {
        c[i] = a[i] + b[i];
    }
}
```

- Example 2:

```
void vec(double *a, double *b, int off)
{
#pragma ivdep
    for(int i = 0; i < N; i++) {
        b[i] = a[i + off];
    }
}
```

Data Dependence - Memory Disambiguation

- Using keyword `restrict` for pointer declarations
- Pointers with `restrict` keyword are not aliasing any other pointer
- Was introduced for C99 (`-std=c99`)
- Some compilers support it with C++ and C89 by enabling `-restrict` option (not standardized)
- Example:

```
void vec(double *a, double *b, double * restrict c)
{
    for(int i = 0; i < N; i++) {
        c[i] = a[i] + b[i];
    }
}
```

Alignment - Excursion

- What is alignment?
- Loops are created by compiler threefold:
  - Peeling loop: Mostly scalar till aligned addresses are reached
    $\Rightarrow$ Avoided by proper alignment
  - Main loop: Possibly vectorized starting at aligned addresses
    $\Rightarrow$ Should be (well) vectorized
  - Remainder loop: If elements are not a multiple of VL, a remainder loop needs to handle the rest
    $\Rightarrow$ Get rid of it by hinting compiler
- Give hints to compiler, e.g. `__assume(cols % 64 == 0)`

Alignment

- Has impact on performance (cache lines transferred)
- Also impacts on whether peeling loop is needed
- Should be 32 (AVX/AVX2) or 64(MIC/AVX-512) bytes on Intel architectures
- Alignment can be controlled by:
  - Dynamic allocation:
    - `void *_mm_malloc(int size, int base)`
    - `int posix_memaligned(void **p, size_t base, size_t size)`
  - Static allocation:
    - `__attribute__((aligned(base))) <var>`
    - `__declspec(align(base)) <var>`
  - Assuming alignment:
    `__assume_aligned(<array>, base)`

Alignment

- `#pragma vector [aligned|unaligned]` only for Intel compiler
- Asserts compiler that aligned memory operations can be used for all data accesses inside a loop
- Assertion must be satisfied for all data accesses in loop for correct semantic!

Compiled both cases using **-xAVX**:

```
void mult(double* a, double* b, double* c)
{
  int i;
#pragma vector unaligned
  for (i = 0; i < N; i++)
    c[i] = a[i] * b[i];
}
```

```
..B2.2:
  vmovupd    (%rdi,%rax,8), %xmm0
  vmovupd    (%rsi,%rax,8), %xmm1
  vinsertf128 $1, 16(%rsi,%rax,8), %ymm1, %ymm3
  vinsertf128 $1, 16(%rdi,%rax,8), %ymm0, %ymm2
  vmulpd     %ymm3, %ymm2, %ymm4
  vmovupd    %xmm4, (%rdx,%rax,8)
  vextractf128 $1, %ymm4, 16(%rdx,%rax,8)
  addq       $4, %rax
  cmpq       $1000000, %rax
  jb         ..B2.2
```

More efficient if aligned:

```
void mult(double* a, double* b, double* c)
{
  int i;
#pragma vector aligned
  for (i = 0; i < N; i++)
    c[i] = a[i] * b[i];
}
```

```
..B2.2:
  vmovupd    (%rdi,%rax,8), %ymm0
  vmulpd     (%rsi,%rax,8), %ymm0, %ymm1
  vmovntpd   %ymm1, (%rdx,%rax,8)
  addq       $4, %rax
  cmpq       $1000000, %rax
  jb         ..B2.2
```

(Image: Intel)

Lab Time!