# Semi-automatic vectorization by compilers

# Semi-automatic vectorization by compilers

```
for (i=0; i < N; ++i)
{
  a[i] = b[i] + c[i];
}
```

```
for (i=0; i + 3 < N; i += 4)
{
  a[i] = b[i] + c[i];
  a[i+1] = b[i+1] + c[i+1];
  a[i+2] = b[i+2] + c[i+2];
  a[i+3] = b[i+3] + c[i+3];
}
for (; i < N; ++i)
{
  a[i] = b[i] + c[i];
}
```

```
for (i=0; i + 3 < N; i += 4)
{
  _mm_storeu_ps(a+i,
    _mm_add_ps(
      _mm_loadu_ps(b+i),
      _mm_loadu_ps(c+i)));
}
for (; i < N; ++i)
{
  a[i] = b[i] + c[i];
}
```

- Compilers can vectorize loops
  - Unroll the loop by K iterations
  - Perform the unrolled K iterations in parallel – by vector instructions
- ... but only if some conditions are met

# Semi-automatic vectorization by compilers

```
for (i=0; i < N; ++i)
{
  a[i] = b[i] + c[i];
}
```

```
for (i=0; i + 3 < N; i += 4)
{
  a[i] = b[i] + c[i];
  a[i+1] = b[i+1] + c[i+1];
  a[i+2] = b[i+2] + c[i+2];
  a[i+3] = b[i+3] + c[i+3];
}
for (; i < N; ++i)
{
  a[i] = b[i] + c[i];
}
```

```
for (i=0; i + 3 < N; i += 4)
{
  _mm_storeu_ps(a+i,
    _mm_add_ps(
      _mm_loadu_ps(b+i),
      _mm_loadu_ps(c+i)));
}
for (; i < N; ++i)
{
  a[i] = b[i] + c[i];
}
```

- The loop control variable and the condition must be *predictable*
  - Instead of checking the condition for every i, a modified condition is tested for every K-th i
- Compilers often require *countable* loops
  - The number of iterations must be known (at runtime) before entering the loop
  - Compilers have a built-in list of *countable* loop patterns
    - The source code must match one of these patterns

# Semi-automatic vectorization by compilers

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

```
for (i=0; i < N; ++i)
{
  a[i+1] = a[i] + 1;
}
```

```
a = b + 1;

for (i=0; i < N; ++i)
{
  a[i] = b[i] + c[i];
}
```

- There shall be no *loop-carried dependence*
  - An iteration must not depend on the result of previous iterations, e.g.:
    - Via a variable
    - Via array positions overlapped by index arithmetics
    - Via overlapping arrays (*aliasing*)

# Semi-automatic vectorization by compilers

- Compilers can solve possible loop-carried dependences
  - Test overlapping arrays before entering the loop
    - Additional small overhead

```
i=0;
if ( (a<=b || a>b+3)
  && (a<=c || a>c+3) )
{
  for (; i + 3 < N; i += 4)
  {
    _mm_storeu_ps(a+i,
      _mm_add_ps(
        _mm_loadu_ps(b+i),
        _mm_loadu_ps(c+i)));
  }
}
for (; i < N; ++i)
{
  a[i] = b[i] + c[i];
}
```

# Semi-automatic vectorization by compilers

- What a loop may do to be useful...
  - Find something and break early
    - Unpredictable condition
  - Accumulate some value in a variable
    - Loop-carried dependence via a variable
  - Generate an output array
    - It might overlap an input array – potential loop-carried dependence

- In C/C++, almost no loop can be vectorized as is
    - In Fortran, there is no pointer arithmetics – less danger of aliasing
  - The vectorized code can not be strictly equivalent to the original
    - Order of operations must be changed
  - The programmer must help the compiler somehow
    - Often using a pragma that overrides the conservative approach of the compiler
    - The programmer is now responsible for correctness of the vectorization
      - The programmer ensures the absence of aliasing

# Semi-automatic vectorization by compilers

```
for (i=0; i < N; ++i)
{
  if ( a[i] < b[i] )
    b[i] = b[i] - a[i];
  else
    a[i] = a[i] – b[i];
}
```

```
for (i=0; i < N; ++i)
{
  c = a[i] < b[i];
  x = b[i] - a[i];
  y = a[i] – b[i];
  b[i] = c ? x : b[i];
  a[i] = c ? a[i] : y;
}
```

*Note:*

- *This code is not strictly equivalent in parallel environment*
- *But the same is true for any vectorization due to reordering of memory accesses*
- *Non-sequentially-equivalent memory models defined in modern parallel programming languages allow to ignore the problem*

- Vector instructions do not support branching
  - No nested loops allowed
  - If-then-else allowed only if it can be replaced by masking
    - Both branches are executed for every iteration
    - The result of one branch is masked, i.e. forgotten
      - Like a conditional expression without short-circuit evaluation
    - If one of the branches is significantly larger, the code may execute too many unused computations

# Semi-automatic vectorization by compilers

```
for (i=0; i < N; ++i)
{
  a[ b[ i]] = c[ i];
}
```

```
for (i=0; i + 15 < N; i += 16)
{
  _mm512_i32scatter_ps(
    a,
    _mm512_loadu_epi32(b+i),
    _mm512_loadu_ps(c+i),
    4);
}
for (; i < N; ++i)
{
  a[ b[ i]] = c[ i];
}
```

- Non-contiguous memory access is slow or impossible
  - AVX2 supports *gather*

`a[i] = b[c[i]]`

  - AVX-512 supports *scatter*

`a[b[i]] = c[i]`

  - *Scatter/gather* is significantly slower than continuous *load/store*
    - However faster than scalar memory access
  - Scatter is guaranteed to perform writes in the order of increasing lane index i
    - Applies to overlapping write positions. Non-overlapping positions may be written in any order.
  - Compiler support is only experimental

# Semi-automatic vectorization by compilers

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

```
auto ones = _mm512_set1_epi32(1);
for (i=0; i + 15 < N; i += 16)
{
  auto bb = _mm512_loadu_epi32(b+i);
  auto aa = _mm512_i32gather_epi32( a, bb, 4);
  auto aa1 = _mm512_add_epi32( aa, ones);
  _mm512_i32scatter_ps( a, bb, aa1, 4);
}
for (; i < N; ++i)
{
  ++a[ b[ i]];
}
```

- Example: Histogram creation
- The vectorized code is not equivalent
  - If an index j is present more than once in the vector bb, the result value is incremented only once
    - The fact that scatter operates in a guaranteed order does not help
  - Loop-carried dependence in the original code, between writes and subsequent reads from the same a[j]
    - The compiler shall never ignore this dependence
- Remedy: Explicitly check for the repeated indexes using the AVX512CD extension

# Semi-automatic vectorization by compilers

```
auto ones = _mm512_set1_epi32(1);
for (i=0; i + 15 < N; i += 16)
{
  auto bb = _mm512_loadu_epi32(b+i);
  // compute conflicts
  auto cc = _mm512_conflict_epi32(bb);
  auto cm = _mm512_test_epi32_mask(cc, cc);
  auto m = _knot_mask16(cm);
  for (;;) {
    // do original action masked by m (where necessary)
    auto aa = _mm512_mask_i32gather_epi32( m, a, bb, 4);
    auto aa1 = _mm512_add_epi32( aa, ones);
    _mm512_mask_i32scatter_ps( m, a, bb, aa1, 4);
    // stop if there were no conflicts
    auto z = _kortestz_mask16_u8(cm,cm);
    if (z) break;
    // clear lowermost ones in cc (cc = cc & (cc-1))
    auto cc1 = _mm512_sub_epi32(cc, ones);
    cc = _mm512_and_epi32(cc, cc1);
    // setup new masks
    auto cm1 = _mm512_test_epi32_mask(cc, cc);
    m = _kxor_mask16(cm1, cm);
    cm = cm1;
  };
}
for (; i < N; ++i) { ++a[ b[ i]]; }
```

- *conflict* instruction (AVX512)
  - compares all pairs of lanes for equality
  - triangular matrix returned as i bits in lane i
  - bit j in lane i set if j < i && a[i]==a[j]
- conflict handling
  - detect conflicts (cc)
  - do the required action for lanes having no conflict bit set (m)
  - clear the lowermost conflict bits (these are at the positions just processed)
  - repeat if some conflict bits remain (cm)

# Semi-automatic vectorization by compilers

- Compiler does not know the alignment of pointers
  - It must emit slow unaligned loads/stores
  - It may generate tests to check whether all pointers are aligned
    - Overhead introduced into the code
  - The situation improved since AVX
    - Non-aligned load/stores do not cause faults, only longer latency
    - The compilers may produce optimistic code without test for alignment
    - Applies also for SSE instructions when encoded in VEX encoding (available on AVX-aware CPUs)
      - “-mavx” makes SSE faster!

```
ar = (uintptr_t)a % 16;
br = (uintptr_t)b % 16;
cr = (uintptr_t)c % 16;

if ( ar == br && ar == cr )
{
  for (; i < (16 - ar) % 16 / 4; ++i)
  {
    a[i] = b[i] + c[i];
  }
  for (; i + 3 < N; i += 4)
  {
    _mm_store_ps(a+i,
      _mm_add_ps(
        _mm_load_ps(b+i),
        _mm_load_ps(c+i)));
  }
}
else
  for (i=0; i + 3 < N; i += 4)
  {
    _mm_storeu_ps(a+i,
      _mm_add_ps(
        _mm_loadu_ps(b+i),
        _mm_loadu_ps(c+i)));
  }

for (; i < N; ++i)
{
  a[i] = b[i] + c[i];
}
```

# Semi-automatic vectorization by compilers

- C/C++ vectorization pragmas
  - Placed before the loop to be vectorized

```
#pragma simd
#pragma vector always
#pragma clang loop vectorize(enable)
```

- Override compiler's decision that vectorizing is possible but not advantageous
  - Often issues warning/error if vectorization failed

```
#pragma novector
```

- Disable vectorization

```
#pragma loop count(1000)
```

- Override compiler's estimation of number of iterations

# Semi-automatic vectorization by compilers

- C/C++ vectorization pragmas
  - Placed before the loop to be vectorized

```
#pragma ivdep
#pragma GCC ivdep
```

- Tell the compiler that there are no unprovable loop-carried dependences (via aliasing)
  - Compiler still checks for provable loop-carried dependences (via scalars or index arithmetics)

```
restrict
```

- [C99] Declare that a pointer argument is not aliased to any other pointer with the keyword

```
#pragma vector aligned
```

- Tell the compiler that pointers are always aligned

```
_declspec(align(16))
__attribute__((aligned(16)))
```

- Enforce alignment of variables, assert alignment of pointers

- C/C++ vectorization pragmas
  - Reduction operators

```
#pragma simd reduction(+:s)
for (i=0; i < N; ++i)
{
  s = s + a[i];
}
```