Agenda

• Introduction
• Compiler & Vectorization
• Validating Vectorization Success
• Reasons for Vectorization Fails
• Vectorization of Special Program Constructs & Loops
• Intel® Cilk™ Plus
• OpenMP® 4.0
• Guided Auto Parallelism
• Summary
What is Vectorization?

- Transform sequential code to exploit vector processing capabilities (SIMD) of Intel processors
  - Manually by explicit syntax
  - Automatically by tools like a compiler

```c
for(i = 0; i <= MAX; i++)
    c[i] = a[i] + b[i];
```
Many Ways to Vectorize

- **Compiler:** Auto-vectorization (no change of code)
- **Compiler:** Auto-vectorization hints (#pragma vector, ...)
- **Compiler:** OpenMP* 4.0 and Intel® Cilk™ Plus

- **SIMD intrinsic class**
  (e.g.: F32vec, F64vec, ...)

- **Vector intrinsic**
  (e.g.: _mm_fmadd_pd(...), _mm_add_ps(...), ...)

- **Assembler code**
  (e.g.: [v]addps, [v]addss, ...)

Ease of use → Programmer control
Many Ways to Vectorize

Compiler:
Auto-vectorization (no change of code)

Compiler:
Auto-vectorization hints (#pragma vector, ...)

Compiler:
OpenMP* 4.0 and Intel® Cilk™ Plus

SIMD intrinsic class
(e.g.: F32vec, F64vec, ...)

Vector intrinsic
(e.g.: _mm_fmadd_pd(...), _mm_add_ps(...), ...)

Assembler code
(e.g.: [v]addps, [v]addss, ...)

Ease of use

Programmer control
Many Ways to Vectorize

- Compiler: Auto-vectorization (no change of code)
- Compiler: Auto-vectorization hints (#pragma vector, ...)
- Compiler: OpenMP* 4.0 and Intel® Cilk™ Plus
- SIMD intrinsic class (e.g.: F32vec, F64vec, ...)
- Vector intrinsic (e.g.: _mm_fmadd_pd(...), _mm_add_ps(...), ...)
- Assembler code (e.g.: [v]addps, [v]addss, ...)

Ease of use

Programmer control
Intrinsics – Sample

• Example using AVX intrinsics:

```c
#include <immintrin.h>

double A[40], B[40], C[40];
for (int i = 0; i < 40; i += 4) {
    __m256d a = _mm256_load_pd(&A[i]);
    __m256d b = _mm256_load_pd(&B[i]);
    __m256d c = _mm256_add_pd(a, b);
    _mm256_store_pd(&C[i], c);
}
```

• Example using Intel® MIC Architecture/Intel® AVX-512 intrinsics:

```c
#include <immintrin.h>

double A[40], B[40], C[40];
for (int i = 0; i < 40; i += 8) {
    __m512d a = _mm512_load_pd(&A[i]);
    __m512d b = _mm512_load_pd(&B[i]);
    __m512d c = _mm512_add_pd(a, b);
    _mm512_store_pd(&C[i], c);
}
```
Intel® Intrinsics Guide

Intel provides an interactive intrinsics guide:

- Lists all supported intrinsics
- Sorted by SIMD feature version and generation
- Quickly find the intrinsic via instant search
- Rich documentation of each intrinsic
- Filters for technologies, types & categories

## Many Ways to Vectorize

### Compiler:
- **Auto-vectorization (no change of code)**

### Compiler:
- **Auto-vectorization hints** (`#pragma vector, ...`)

### Compiler:
- **OpenMP* 4.0 and Intel® Cilk™ Plus**

#### SIMD intrinsic class
- (e.g.: `F32vec, F64vec, ...`)

#### Vector intrinsic
- (e.g.: `_mm_fmadd_pd(...), _mm_add_ps(...), ...`)

#### Assembler code
- (e.g.: `[v]addps, [v]addss, ...`)

---

**Ease of use**

**Programmer control**
SIMD Intrinsic Class

• For a full list, please refer to the header files!
• Example for AVX:

```c
#include <dvec.h>

// 4 elements per vector * 25 = 100 elements
F64vec4 A[25], B[25], C[25];

for(int i = 0; i < 25; i++)
    C[i] = A[i] + B[i];
```
Agenda

• Introduction
• **Compiler & Vectorization**
  • Validating Vectorization Success
  • Reasons for Vectorization Fails
  • Vectorization of Special Program Constructs & Loops
• Intel® Cilk™ Plus
• OpenMP* 4.0
• Guided Auto Parallelism
• Summary
Many Ways to Vectorize

Compiler:
Auto-vectorization (no change of code)

Compiler:
Auto-vectorization hints (#pragma vector, ...)

Compiler:
OpenMP* 4.0 and Intel® Cilk™ Plus

SIMD intrinsic class
(e.g.: F32vec, F64vec, ...)

Vector intrinsic
(e.g.: _mm_fmadd_pd(...), _mm_add_ps(...), ...)

Assembler code
(e.g.: [v]addps, [v]addss, ...)

Ease of use

Programmer control
Many Ways to Vectorize

Compiler:
Auto-vectorization (no change of code)

Compiler:
Auto-vectorization hints (#pragma vector, ...)

Compiler:
OpenMP* 4.0 and Intel® Cilk™ Plus

SIMD intrinsic class
(e.g.: F32vec, F64vec, ...)

Vector intrinsic
(e.g.: _mm_fmadd_pd(...), _mm_add_ps(...), ...)

Assembler code
(e.g.: [v]addps, [v]addss, ...)

Ease of use

Programmer control
void add(A, B, C)
double A[1000]; double B[1000]; double C[1000];
{
    int i;
    for (i = 0; i < 1000; i++)
        C[i] = A[i] + B[i];
}

subroutine add(A, B, C)
real*8 A(1000), B(1000), C(1000)
do i = 1, 1000
    C(i) = A(i) + B(i)
end do
end

..B1.2:
vmovupd (%rsp,%rax,8), %ymm0
vmovupd 32(%rsp,%rax,8), %ymm2
vmovupd 64(%rsp,%rax,8), %ymm4
vmovupd 96(%rsp,%rax,8), %ymm6
vaddpd 8032(%rsp,%rax,8), %ymm2, %ymm3
vaddpd 8000(%rsp,%rax,8), %ymm0, %ymm1
vaddpd 8064(%rsp,%rax,8), %ymm4, %ymm5
vaddpd 8096(%rsp,%rax,8), %ymm6, %ymm7
vmovupd %ymm1, 16000(%rsp,%rax,8)
vmovupd %ymm3, 16032(%rsp,%rax,8)
vmovupd %ymm5, 16064(%rsp,%rax,8)
vmovupd %ymm7, 16096(%rsp,%rax,8)
addq $16, %rax
cmpq $992, %rax
jb ..B1.2

..B1.2:
movaps (%rsp,%rax,8), %xmm0
movaps 16(%rsp,%rax,8), %xmm1
movaps 32(%rsp,%rax,8), %xmm2
movaps 48(%rsp,%rax,8), %xmm3
addpd 8000(%rsp,%rax,8), %xmm0
addpd 8016(%rsp,%rax,8), %xmm1
addpd 8032(%rsp,%rax,8), %xmm2
addpd 8048(%rsp,%rax,8), %xmm3
movaps %xmm0, 16000(%rsp,%rax,8)
movaps %xmm1, 16016(%rsp,%rax,8)
movaps %xmm2, 16032(%rsp,%rax,8)
movaps %xmm3, 16048(%rsp,%rax,8)
addq $8, %rax
cmpq $1000, %rax
jb ..B1.2
## SIMD Features I

Support of SIMD extensions for Intel processors:

<table>
<thead>
<tr>
<th>SIMD Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATOM_SSE4.2</td>
<td>May generate MOVBE instructions for Intel processors (depending on setting of <code>-minstruction</code> or <code>/Qinstructıon</code>). May also generate Intel® SSE4.2, SSE3, SSE2 and SSE instructions for Intel processors. Optimizes for Intel® Atom™ processors that support Intel® SSE4.2 and MOVBE instructions.</td>
</tr>
<tr>
<td>SSE4.2</td>
<td>May generate Intel® SSE4.2, SSE4.1, SSE3, SSE2, SSE and Intel SSSE3.</td>
</tr>
<tr>
<td>SSE4.1</td>
<td>May generate Intel® SSE4.1, SSE3, SSE2, SSE and Intel SSSE3.</td>
</tr>
<tr>
<td>ATOM_SSSE3</td>
<td>May generate MOVBE instructions for Intel processors (depending on setting of <code>-minstruction</code> or <code>/Qinstructıon</code>). May also generate Intel® SSE3, SSE2, SSE and Intel SSSE3 instructions for Intel processors. Optimizes for Intel® Atom™ processors that support Intel® SSE3 and MOVBE instructions.</td>
</tr>
<tr>
<td>SSE3_ATOM &amp; SSSE3_ATOM</td>
<td></td>
</tr>
<tr>
<td>SSSE3</td>
<td>May generate Intel® SSE3, SSE2, SSE and Intel SSSE3.</td>
</tr>
<tr>
<td>SSE3</td>
<td>May generate Intel® SSE3, SSE2 and SSE.</td>
</tr>
<tr>
<td>SSE2</td>
<td>May generate Intel® SSE2 and SSE.</td>
</tr>
</tbody>
</table>
## SIMD Features II

Support of SIMD extensions for Intel processors (cont’d):

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORE-AVX2</td>
<td>May generate Intel® Advanced Vector Extensions 2 (Intel® AVX2), Intel® AVX, SSE4.2, SSE4.1, SSE3, SSE2, SSE and Intel SSSE3 instructions.</td>
</tr>
<tr>
<td>CORE-AVX-I</td>
<td>May generate Intel® Advanced Vector Extensions (Intel® AVX), including instructions in 3rd generation Intel® Core™ processors, Intel® SSE4.2, SSE4.1, SSE3, SSE2, SSE and Intel SSSE3.</td>
</tr>
<tr>
<td>AVX</td>
<td>May generate Intel® Advanced Vector Extensions (Intel® AVX), SSE4.2, SSE4.1, SSE3, SSE2, SSE and Intel SSSE3.</td>
</tr>
</tbody>
</table>
Basic Vectorization Switches I

- **Linux**, **OS X**: `-x<feature>`, **Windows**: `/Qx<feature>`
  - Might enable Intel processor specific optimizations
  - Processor-check added to “main” routine:
  - Application errors in case SIMD feature missing or non-Intel processor with appropriate/informative message

- **Linux**, **OS X**: `-ax<features>`, **Windows**: `/Qax<features>`
  - Multiple code paths: baseline and optimized/processor-specific
  - Optimized code paths for Intel processors defined by `<features>`
  - Multiple SIMD features/paths possible, e.g.: `-axSSE2,AVX`
  - Baseline code path defaults to `-msse2` (/arch:sse2)
  - The baseline code path can be modified by `-m<feature>` or `-x<feature>` (/arch:<feature> or /Qx<feature>)
Basic Vectorization Switches II

• Linux*, OS X*: `-m<feature>`, Windows*: `/arch:<feature>`
  Neither check nor specific optimizations for Intel processors:
  Application optimized for both Intel and non-Intel processors for selected SIMD feature
  Missing check can cause application to fail in case extension not available
• Default for Linux*: `-msse2`, Windows*: `/arch:sse2`: 
  Activated implicitly
  Implies the need for a target processor with at least Intel® SSE2
• Default for OS X*: `-xsse3` (IA-32), `-xssse3` (Intel® 64)

• For 32 bit compilation, `-mia32` (/arch:ia32) can be used in case target processor does not support Intel® SSE2 (e.g. Intel® Pentium® 3 or older)
Basic Vectorization Switches III

• Special switch for Linux*, OS X*: \texttt{-xHost}, Windows*: \texttt{/QxHost}
  - Compiler checks SIMD features of current host processor (where built on) and makes use of latest SIMD feature available
  - Code only executes on processors with same SIMD feature or later as on build host
  - As for \texttt{-x<feature>} or \texttt{/Qx<feature>}, if “main” routine is built with \texttt{-xHost} or \texttt{/QxHost} the final executable only runs on Intel processors
VectorizationPragma/Directive

• SIMD features can also be set on a function/subroutine level via
  pragmas/directives:
  
  C/C++:
  #pragma intel optimization_parameter target_arch=<CPU>
  
  Fortran:
  !DIR$ ATTRIBUTES OPTIMIZATION_PARAMETER:TARGET_ARCH= <CPU>

• Examples:
  
  C/C++:
  
  #pragma intel optimization_parameter target_arch=AVX
  void optimized_for_AVX()
  {
   ...
  }

  Fortran:
  
  function optimized_for_AVX()
  !DIR$ ATTRIBUTES OPTIMIZATION_PARAMETER:TARGET_ARCH=AVX
  ...
  end function
Some Notes on Vectorization Switches

• It is not possible anymore to generate vector code exclusively for the initial Intel® SSE instruction set introduced by
• Intel® Pentium® 3 processor. Use -mia32 or /arch:IA32 for anything before Intel® SSE2.
• The SIMD feature name for Intel® Atom™ processors has been renamed from SSE3_ATOM to SSSE3_ATOM. It is basically
• Intel SSSE3 plus movbe instructions. The latter can be disabled/enabled with
• Further information:
Control Vectorization I

• Disable vectorization:
  - Globally via switch:
    - Linux*, OS X*: `-no-vec`, Windows*: `/Qvec`
  - For a single loop:
    - C/C++: `#pragma novector`, Fortran: `!DIR$ NOVECTOR`
  - Compiler still can use some SIMD features

• Using vectorization:
  - Globally via switch (default for optimization level 2 and higher):
    - Linux*, OS X*: `-vec`, Windows*: `/Qvec`
  - Enforce for a single loop (override compiler efficiency heuristic) if semantically correct:
    - C/C++: `#pragma vector always`, Fortran: `!DIR$ VECTOR ALWAYS`
  - Influence efficiency heuristics threshold:
    - Linux*, OS X*: `-vec-threshold[n]`
    - Windows*: `/Qvec-threshold[[:]n]`
    - n: 100 (default; only if profitable) … 0 (always)
Control Vectorization II

• Verify vectorization:
  - Globally:
    - Linux*, OS X*: `-opt-repot`, Windows*: `/Qopt-report`
  - Abort compilation if loop cannot be vectorized:
    - C/C++: `#pragma vector always assert`
    - Fortran: `!DIR$ VECTOR ALWAYS ASSERT`

• Advanced:
  - Ignore vector dependencies (IVDEP):
    - C/C++: `#pragma ivdep`
    - Fortran: `!DIR$ IVDEP`
  - “Enforce” vectorization:
    - C/C++: `#pragma simd`
    - Fortran: `!DIR$ SIMD`

• We’ll address those later in more detail
Agenda

• Introduction
• Compiler & Vectorization
• Validating Vectorization Success
• Reasons for Vectorization Fails
• Vectorization of Special Program Constructs & Loops
• Intel® Cilk™ Plus
• OpenMP* 4.0
• Guided Auto Parallelism
• Summary
Validating Vectorization Success I

• Assembler code inspection (Linux*: -S, -Fa, Windows*: /Fa):
  - Most reliable way and gives all details of course
  - Check for scalar/packed or (E)VEX encoded instructions:
  - Assembler listing contains source line numbers for easier navigation

• Using Intel® VTune™ Amplifier:
  - Different events can be selected to measure use of vector units, e.g.
    FP_COMPILE_OPS_EXE.SSE_PACKED_[SINGLE|DOUBLE]
  - For Intel® MIC Architecture: Use metric Vectorization Intensity

• Difference method:
  - Compile and benchmark with -no-vec/Qvec- or on a loop by loop basis via
    #pragma novector/!DIR$ NOVECTOR
  - Compile and benchmark with selected SIMD feature
  - Compare runtime differences
Validating Vectorization Success II

**Intel® Software Development Emulator:**
- Emulate (future) Intel® Architecture Instruction Set Extensions (e.g. Intel® AVX-512, Intel® MPX, …)
- Use the “mix histogramming tool” to check for instructions using vectors
- Also possible to debug the application while emulated

**Intel® Architecture Code Analyzer:**
- Statically analyze the data dependency, throughput and latency of code snippets (aka. kernels)
- Considers ideal front-end, out-of-order engine and memory hierarchy conditions
- Identifies binding of the kernel instructions to the processor ports & critical path
Validating Vectorization Success III

• Optimization report:
  - Linux*, OS X*: `-opt-report=<n>`, Windows*: `/Qopt-report:<n>`
  - `n`: 0, ..., 5 specifies level of detail; 2 is default (more later)
  - Prints optimization report with vectorization analysis
  - Also known as vectorization report for Intel® C++/Fortran Compiler before 15.0:
    - Linux*, OS X*: `-vec-report=<n>`, Windows*: `/Qvec-report:<n>`
    - Deprecated, don’t use anymore – use optimization report instead!

• Optimization report phase:
  - Linux*, OS X*: `-opt-report-phase=<p>`,
  - Windows*: `/Qopt-report-phase:<p>`
  - `<p>` is `all` by default; use `vec` for just the vectorization report

• Optimization report file:
  - `<f>` can be `stderr`, `stdout` or a file (default: `*.optrpt`)
Example **novec.f90**:  
1: subroutine fd(y)  
2:   integer :: i  
3:   real, dimension(10), intent(inout) :: y  
4:   do i=2,10  
5:     y(i) = y(i-1) + 1  
6:   end do  
7: end subroutine fd

$ ifort novec.f90 -opt-report=5
ifort: remark #10397: optimization reports are generated in *.optrpt files in the output location

$ cat novec.optrpt
...
LOOP BEGIN at novec.f90(4,5)
    remark #15344: loop was not vectorized: vector dependence prevents vectorization
    remark #15346: vector dependence: assumed FLOW dependence between y line 5 and y line 5
    remark #25436: completely unrolled by 9
LOOP END
...
Agenda

• Introduction to SIMD for Intel® Architecture
• Vector Code Generation
• Compiler & Vectorization
• Validating Vectorization Success
• Reasons for Vectorization Fails
• Vectorization of Special Program Constructs & Loops
• Intel® Cilk™ Plus
• OpenMP* 4.0
• Guided Auto Parallelism
• Summary
Reasons for Vectorization Fails I

Most frequent reasons:
• Data dependence
• Alignment
• Unsupported loop structure
• Non-unit stride access
• Function calls/in-lining
• Non-vectorizable Mathematical functions
• Data types
• Control dependence
• Bit masking

All those are common and will be explained in detail next!
Reasons for Vectorization Fails II

Other reasons:
• Outer loop of loop nesting cannot be vectorized
• Loop body too complex (register pressure)
• Vectorization seems inefficient (low trip count)
• Many more

Those are less likely and are not described in the following!
Data/Control Dependence

**Dependence is a key term for vectorization:**
- Vectorization is a transformation changing the execution order of statements.
- The execution order of statements as defined by the program source code can be changed as long as the dependencies between all statements are preserved.

**A dependence either is a data or control dependence:**

**Data dependence** from

S1 to S3 and from S2 to S3

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>A = 3.0</td>
</tr>
<tr>
<td>S2</td>
<td>B = 4.0</td>
</tr>
<tr>
<td>S3</td>
<td>C = sqrt(A<strong>2, B</strong>2)</td>
</tr>
</tbody>
</table>

From

S1 to S2

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>if (T != 0)</td>
</tr>
<tr>
<td>S2</td>
<td>A = A / T</td>
</tr>
</tbody>
</table>
Definition of data dependence:
There is a data dependence from statement S1 to statement S2 (written as $S1 \delta S2$) if and only if:
• There is a potential execution flow from S1 to S2
• S1 and S2 reference a common memory location S1 or S2 write to

Note: S1 and S2 can be the very same statement

Data dependence classification:
• $S1 \delta F S2$: S1 writes, S2 reads: Flow Dependence
• $S1 \delta A S2$: S1 reads, S2 writes: Anti Dependence
• $S1 \delta O S2$: S1 writes, S2 writes: Output Dependence
Data Dependence in Loops

Dependencies in loops become more obvious by virtually unrolling the loop:

In case the dependency requires execution of any previous loop iteration, we call it **loop-carried dependence**. Otherwise, **loop-independent dependence**.

E.g.:

S1 δF S2: Loop-independent dependence

S2: Loop-carried dependence
Dependencies: Student Exercise

Find all dependencies in the examples, if any.
Which are the dependency types?

1) for(i = 0; i < MAX - 2; i++)
2) S1 A[i + 2] = A[i] + 1;

1) for(i = 1; i < MAX; i++)
2) {
4) S3 B = A[i - 1];
5) }

1) for(i = 0; i < MAX - 1; i++)
2) S4 A[i + 1, j] = A[i, k] + B;

1) for(i = 0; i < MAX - 1; i++)
Dependence and Vectorization

Vectorization of a loop is similar to parallelization (loop iterations executed in parallel), however not identical:

- **Parallelization** requires all iterations to be independent to be executed in any order:
  - Loop-carried dependencies are not permitted; loop-independent dependencies are OK
- **Vectorization** is applied to single operations of the loop body:
  - The same operations can be applied for multiple iterations at once if they follow serial order; both loop-carried & loop-independent dependencies need to be taken into account!

Example: Loop cannot be parallelized but vectorization possible:

```plaintext
DO I = 1, N
   A(I + 1) = B(I) + C
   D(I) = A(I) + E
END DO
```

```
A(2:N + 1) = B(1:N) + C
D(1:N) = A(1:N) + E
```
A loop can be vectorized if and only if there is no cyclic dependency chain between the statements of the loop body!

- For the formal proof we refer to the literature [3]
- The theorem takes into account that certain semantic-preserving reordering transformations can be applied
  - (e.g. loop distribution, loop fusion, etc.)
- The theorem assumes an “unlimited” vector length (VL).
- In cases where VL is limited, loop carried dependencies might be ignored if more than “VL” iterations are required to exist.

**Thus in some cases vectorization for SSE or AVX might be still valid, opposed to the theorem!**

Example:
Although we have a cyclic dependency chain, the loop can be vectorized for SSE or AVX in case of VL being max. 3 times the data type size of array $A$. 

```
DO I = 1, N
   A(I + 3) = A(I) + C
END DO
```
Failing Disambiguation

Many dependencies assumed by compiler are false dependencies caused by unresolved memory disambiguation:

The compiler has to be conservative and has to assume the worst case regarding “aliasing”!

Example:

```c
void scale(int *a, int *b)
{
    for (int i = 0; i < 10000; i++) b[i] = z * a[i];
}
```

Without additional information, compiler has to assume dependencies.

Use directives, switches and attributes to aid disambiguation:

• This is programming language and operating system specific
• Use with care as the compiler might generate incorrect code in case the hints are not fulfilled!
Disambiguation Hints I

• Disambiguating memory locations of pointers in C99:
  • Linux*, OS X*: `-std=c99`, Windows*: `/Qstd=c99`
  • Intel® C++ Compiler also allows this for other modes
    • (e.g. `-std=c89`, `-std=c++0x`, ...), too - not standardized, though:
    • Linux*, OS X*: `-restrict`, Windows*: `/Qrestrict`

• Declaring pointers with keyword `restrict` asserts compiler that they only reference individually assigned, non-overlapping memory areas
  • Also true for any result of pointer arithmetic (e.g. `ptr + 1` or `ptr[1]`)

Examples:

```c
void scale(int *a, int *restrict b)
{
    for (int i = 0; i < 10000; i++) b[i] = z * a[i];
}

void mult(int a[][NUM], int b[restrict][NUM])
{ ... }
```
Disambiguation Hints II

Directives:
• #pragma ivdep (C/C++) or !DIR$ IVDEP (Fortran)
• #pragma simd (C/C++) or !DIR$ SIMD (Fortran)

For C/C++:
• Assume no aliasing at all (dangerous!):
  • Linux*, OS X*: -fno-alias, Windows*: /Oa
• Assume ISO C Standard aliasing rules:
  • Linux*, OS X*: -ansi-alias, Windows*: /Qansi-alias
• Default with 15.0 and later but not with earlier versions!
• Turns on ANSI aliasing checker, too (thus recommended)
• No aliasing between function arguments:
  • Linux*, OS X*: -fargument-noalias, Windows*: /Qalias-args-
• No aliasing between function arguments and global storage:
  • Linux*, OS X*: -fargument-noalias-global, Windows*: N/A
Disambiguation Hints III

For Fortran:
• Assume no aliasing at all:
  • Linux*, OS X*: -fno-alias, Windows*: /Oa
• Assume Fortran Standard aliasing rules:
  • Linux*, OS X*: -ansi-alias, Windows*: /Qansi-alias
• Opposed to C/C++ this is default since ever!
• No aliasing of Cray* pointers:
  • Linux*, OS X*: -safe-crty-ptr, Windows*: /Qsafe-crty-ptr
Dynamic Data Dependency Check

The compiler can(!) use run-time checks to test for aliasing:
• Both, a vectorized and scalar version of a loop can be created
• (loop versioning) and used upon result of test
• Compiler heuristics select a balance between overhead of testing and performance gains

**Example:**
Loop with body: \( A[...] = B1[...] + B2[...] + \ldots + BN[...] \) causes tests whether arrays \( Bx \) overlap with array \( A \) to avoid dependencies

**Limitation:**
• To avoid combinational growth of code the compiler limits versions
• (e.g. \( N \) max. 2 for the above example)
• Switch to turn off limits:
  • Linux*, OS X*: `-opt-multi-version-aggressive`
  • Windows*: `/Qopt-multi-version-aggressive`
Inter-Procedural Dependency Analysis

- Optimization usually takes place individually for each procedure.
- Dependency analysis of inter-procedural optimization (IPO) works across all procedures and thus allows **global optimization**.
- Switch to turn on IPO for single file (one compilation unit):
  - Linux*, OS X*: `-ip`
  - Windows*: `/Qip`
  - Subset already default for optimization levels 2 and higher.
- Switch to turn on IPO for all compilation units:
  - Linux*, OS X*: `-ipo`
  - Windows*: `/Qipo`
- Example:
- References of function arguments can be analyzed even if located in other compilation unit.
Alignment

Caveat with using unaligned memory access:
• Unaligned loads and stores can be very slow due to higher I/O because two cache-lines need to be loaded/stored (not always, though)
• Compiler can mitigate expensive unaligned memory operations by using two partial loads/stores – still slow
• (e.g. two 64 bit loads instead of one 128 bit unaligned load)
• The compiler can use “versioning” in case alignment is unclear:
• Run time checks for alignment to use fast aligned operations if possible, the slower operations otherwise – better but limited

Best performance: User defined aligned memory
• 16 byte for SSE
• 32 byte for AVX
• 64 byte for Intel® MIC Architecture & Intel® AVX-512
Alignment Hints for C/C++ I

• Aligned heap memory allocation by intrinsic/library call:
  - `void* _mm_malloc(int size, int base)`
  - Linux*, OS X* only:
    - `int posix_memaligned(void **p, size_t base, size_t size)`

• `#pragma vector [aligned|unaligned]`
  - Only for Intel Compiler
  - Asserts compiler that aligned memory operations can be used for all data accesses in loop following directive
  - **Use with care:**
    - The assertion must be satisfied for all(!) data accesses in the loop!
Alignment Hints for C/C++ II

• Align attribute for variable declarations:
  Linux*, OS X*, Windows*: `__declspec(align(base)) <var>`
  Linux*, OS X*: `<var> __attribute__((aligned(base)))`

  **Portability caveat:**
  `__declspec` is not known for GCC and `__attribute__` not for Microsoft Visual Studio*!

• Hint that start address of an array is aligned (Intel Compiler only):
  • `__assume_aligned(<array>, base)`
Alignment Hints for Fortran

• !DIR$ VECTOR [ALIGNED|UNALIGNED]
  Asserts compiler that aligned memory operations can be used for all data accesses in loop following directive
  **Use with care:**
  The assertion must be satisfied for all(!) data accesses in the loop!

• Hint that an entity in memory is aligned:
  • !DIR$ ASSUME_ALIGNED address1:base [, address2:base] ...

• Align variables:
  • !DIR$ ATTRIBUTES ALIGN: base :: variable

• Align data items globally:
  • Linux*, OS X*: -align <a>, Windows*: /align:<a>
    - <a> can be array<n>byte with <n> defining the alignment for arrays
    - Other values for <a> are also possible, e.g.: [no]commons, [no]records, ...

**All are Intel® Fortran Compiler only directives and options!**
Let's assume `a`, `b` and `c` are declared 16 byte aligned in calling routine

**Question:** Would this be correct when compiled for Intel® SSE2?

**Answer:** It depends on `COLWIDTH`!

- `COLWIDTH` is even: Yes
- `COLWIDTH` is odd: No, vectorized code fails with alignment error after first row!

**Solution:**
- Instead of pragma, use `__assume_aligned(<array>, base)`. This refers to the start address only. Vectorization is still limited, though!
Alignment & Processor Architecture

• Instructions with unaligned access are very slow except for SSE vector memory operations (128 bit) on 2nd and 3rd generation Intel® Core™ processors (as fast as aligned access)
• For AVX vectors (256 bit) unaligned accesses are slower compared to their aligned accesses, even on 3rd generation Intel® Core™ processors
• Independent on processor generation and instruction set features, one unaligned instructions can replace a sequence of multiple instructions:
  - Fewer instructions result in less cycles, better use of instruction-cache and less power consumption
  - To benefit make sure to at least use latest SSE/AVX feature set
    (default for Intel® C++/Fortran Compiler is Intel® SSE2)

**Attention:**
• When using SSE instructions directly (e.g. intrinsics) any aligned move on unaligned data still fails!
Alignment Impact: Example

Compiled both cases using `-xAVX`:

```c
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:

```c
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
```
SAXPY Alignment Test for AVX

Float alignment \([\text{src1, src2, dst}] : (\text{addr}\%32)/4\)

- All buffers unaligned
- 2 loads unaligned
- 1 load
- Page split
- Store
- AVX fastest
- 16 Byte alignment

Cycles
Unsupported Loop Structure

• Loops where compiler does not know the iteration count:
  - Upper/lower bound of a loop are not loop-invariant
  - Loop stride is not constant
  - Early bail-out during iterations (e.g. break, exceptions, etc.)
  - Too complex loop body conditions for which no SIMD feature instruction exists
  - Loop dependent parameters are globally modifiable during iteration (language standards require load and test for each iteration)

• Transform is possible, e.g.:

```c
struct _x { int d; int bound; }; 
void doit(int *a, struct _x *x) 
{ 
    for(int i = 0; i < x->bound; i++)
        a[i] = 0;
}
```

```c
struct _x { int d; int bound; }; 
void doit(int *a, struct _x *x) 
{ 
    int local_ub = x->bound;
    for(int i = 0; i < local_ub; i++)
        a[i] = 0;
}
```
Non-Unit Stride Access

• Non-consecutive memory locations are being accessed in the loop
• Vectorization works best with contiguous memory accesses
• Vectorization still be possible for non-contiguous memory access, but…
  Data arrangement operations might be too expensive
  (e.g. access pattern linear/regular)
  Vectorization report issued when too expensive:
  Loop was not vectorized: vectorization possible but seems inefficient
• Examples:

```c
for(i = 0; i <= MAX; i++) {
    for(j = 0; j <= MAX; j++) {
        D[i][j] += 1; // Unit stride
        D[j][i] += 1; // Non-unit stride but linear
        A[j * j] += 1; // Non-unit stride
        A[B[j]] += 1; // Non-unit stride (scatter)
        if(A[MAX - j]) == 1) last = j; // Non-unit stride
    }
}
```
Avoiding Non-Unit Stride Access

• Code transformations like loop interchange can avoid non-unit access frequently in case access is linear
• Compiler can do this automatically via loop interchange in most cases, e.g. matrix multiplication loop:

```c
for(i = 0; i < N; i++)
  for(j = 0; j < N; j++)
    for(k = 0; k < N; k++)
      c[i][j] = c[i][j] + a[i][k] * b[k][j];
```

• But in other cases:

```c
// Non-unit stride
for (j = 0; j < N; j++)
  for (i = 0; i < j; i++)
    c[i][j] = a[i][j] + b[i][j];
```

```c
// Unit stride
for (i = 0; i < N; i++)
  for (j = i + 1; i < N; j++)
    c[i][j] = a[i][j] + b[i][j];
```
Function Calls/In-lining I

- Function calls prevent vectorization in general
- Exceptions:
  - Call of intrinsic routines such as mathematical functions: Implementation is known to compiler
  - Successful in-lining of called routine:
    - IPO enables in-lining of routines across source files

```c
for (i = 1; i < nx; i++) {
    x = x0 + i * h;
    sumx = sumx + func(x, y, xp, yp);
}

// Defined in different compilation unit!
float func(float x, float y, float xp, float yp)
{
    float denom;
    denom = (x - xp) * (x - xp) + (y - yp) * (y - yp);
    denom = 1. / sqrt(denom);
    return denom;
}
```
Function Calls/In-lining II

• Success of in-lining can be verified using the optimization report:
  • Linux*, OS X*: `-opt-report=<n> -opt-report-phase=ipo`
  • Windows*: `/Qopt-report:<n> /Qopt-report-phase:ipo`

• Intel compilers offer a large set of switches, directives and language extensions to control in-lining globally or locally, e.g.:
  `#pragma [no]inline (C/C++), !DIR$ [NO]INLINE (Fortran):`
  
  Instructs compiler that all calls in the following statement can be in-lined or may never be in-lined
  `#pragma forceinline (C/C++), !DIR$ FORCEINLINE (Fortran):`
  
  Instructs compiler to ignore the heuristic for in-lining and to inline all calls in the following statement
  
  See section “Inlining Options” in compiler manual for full list of options

• IPO offers additional advantages to vectorization
  
  Inter-procedural alignment analysis
  Improved (more precise) dependency analysis
Vectorizable Mathematical Functions

• Calls to most mathematical functions in a loop body can be vectorized using “Short Vector Math Library”:
  - Short Vector Math Library (libsvml) provides vectorized implementations of different mathematical functions
  - Optimized for latency compared to the VML library component of Intel® MKL which realizes same functionality but which is optimized for throughput
• Routines in libsvml can also be called explicitly, using intrinsics
  - (see manual)
• These mathematical functions are currently supported:

<table>
<thead>
<tr>
<th>Function</th>
<th>Function</th>
<th>Function</th>
<th>Function</th>
<th>Function</th>
<th>Function</th>
<th>Function</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>acos</td>
<td>acosh</td>
<td>asin</td>
<td>asinh</td>
<td>atan</td>
<td>atanh</td>
<td>cbrt</td>
<td>ceil</td>
</tr>
<tr>
<td>cos</td>
<td>cosh</td>
<td>erf</td>
<td>erfc</td>
<td>erfinv</td>
<td>exp</td>
<td>exp2</td>
<td>fabs</td>
</tr>
<tr>
<td>floor</td>
<td>fmax</td>
<td>fmin</td>
<td>log</td>
<td>log10</td>
<td>log2</td>
<td>pow</td>
<td>round</td>
</tr>
<tr>
<td>sin</td>
<td>sinh</td>
<td>sqrt</td>
<td>tan</td>
<td>tanh</td>
<td>trunc</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Data Types

• Objects (variables, constants, …) used in a statement to be vectorized may have different types and/or sizes
• The compiler frequently can still vectorize them, e.g. using
• SSE/AVX conversion, insertion, extraction, etc. instructions
• To analyze the cases where this is not possible (unsupported data type) consider:
  • Partially surprising rules for implicit data type promotions defined by the programming language standard
  • Potentially SSE/AVX instructions
  • The size differences for source and result operands required for operations like a multiplication
• In case the “complex” data type (Fortran or C99) is being used Intel® SSE3 provides the basic arithmetic instructions to support vectorization
Data Types Impact: Student Exercise

Explain why the code on the right...
• vectorizes if SUM_TYPE is int
• not vectorizes if SUM_TYPE is short
... when using Intel® SSE2

The type of the intermediate result of the addition of \( v[i] \) and \( c \) can be the same as the LHS type, which is SUM_TYPE. Hence the compiler could vectorize the loop with a vector length of 4 for the case SUM_TYPE is int and 8 for SUM_TYPE being short.

To use packed addition with \( v[i] \) and \( c \), the scalar \( c \) needs to be broadcast to each of the 4 or 8 elements of a XMM register vector.

So, why does this only work with SUM_TYPE being int and not short?

Hint:
The key instructions here are \texttt{pshuf[d|w]}.
Control Dependence

• Control dependencies caused by a complex control flow within the loop body prevent vectorization in general.
• However, loops with conditional statements can be vectorized frequently using a bit masking technique, e.g.:

```c
for(int i = 0; i < N; i++)
{
    if (R1[i] > R2[i])
        L1[i] = R1[i];
    else
        L2[i] = R2[i];
}
```

The `SSE` feature set facilitates basic construction of bit mask generation and masking (see earlier example); `AVX` even provides more advanced masked operations.
Bit Masking: Guarding against Errors

• In the previous example the conversion causes both paths (consequence & alternative) to be evaluated for all iterations
• The compiler does this **only in case it won’t introduce errors** which have been protected in the original (sequential) code
• In the previous example the compiler can be sure to not introduce a new exception since the RHS expressions are touched in the test anyway for each iteration
• **Example:**
  • Illegal memory access is possible for element \( a[n] \), depending on values of array \( b \).
  • Vector report would be the following:
    • **loop was not vectorized: condition may**
    • Vectorization related directives like `#pragma vector always, #pragma ivdep & #pragma simd assert` to compiler that the masking transformation is still safe

```c
for (i = 0; i < n; i++)
  if (b[i] == 0)
    a[i] = b[i];
  else
    a[n] = i;
```
Which loops will vectorize?

1) for(int i = 1; i < MAX; i++)
   a[i] = a[i - n] + b[i];

2) for(int i = 0; i < MAX; i += 2)
   b[i] += a[i] * x[i];

3) for(int i = 0; i < MAX; j++)
   for(int j = 0; j < MAX; j++)
     b[i] += a[i][j] * x[j];

4) for(int i = 0; i < MAX; i++)
   b[i] += a[i] * x[index[i]];

5) for(int i = 1; i < MAX; i++)
   sum = sum + a[i] * b[i];

6) for(int i = 0; i < MAX; i++)
   if(s >= 0)
     x2[i] = (-b[i] + sqrt(s)) / (2. * a[i]);
Vectorization: Student Exercise II

• Compile and run a simple program multiplying a matrix with a vector showing some of the topics introduced up to now like vectorization reports, dependence, memory disambiguation and alignment!

• Create a program showing the benefits of inter-procedural optimization!

• Create a program using Short Vector Math Library and verify its use!
How to Succeed in Vectorization?

• Most frequent reason of failing vectorization is **Dependence**: Minimize dependencies among iterations by design!
• **Alignment**: Align your arrays/data structures
• **Function calls in loop body**: Use aggressive in-lining (IPO)
• **Complex control flow/conditional branches**: Avoid them in loops by creating multiple versions of loops
• **Unsupported loop structure**: Use loop invariant expressions
• **Not inner loop**: Manual loop interchange possible? Intel Compilers 12.1 and higher can do outer loop vectorization now as well!
• **Mixed data types**: Avoid type conversions in rare cases Intel Compiler cannot do automatically
How to Succeed in Vectorization? II

• **Non-unit stride between elements:**
  • Possible to change algorithm to allow linear/consecutive access?

• **Loop body too complex reports:** Try splitting up the loops!

• **Vectorization seems inefficient reports:**
  • Enforce vectorization, benchmark and verify results!
Agenda

• Introduction
• Compiler & Vectorization
• Validating Vectorization Success
• Reasons for Vectorization Fails
• Vectorization of Special Program Constructs & Loops
• Intel® Cilk™ Plus
• OpenMP® 4.0
• Guided Auto Parallelism
• Summary
Idiom Recognition

• Intel compilers recognize program constructs which can be mapped onto compact idiomatic SSE/AVX instructions providing optimal performance
• Idiom recognition only works if source code express what can be mapped to corresponding SSE/AVX instructions
• Example:

```c
unsigned char a[N], b[N];
void swap32(int n)
{
    int i;
    for (i = 0; i < n; i += 4) {
        a[i + 0] = b[i + 3];
        a[i + 1] = b[i + 2];
        a[i + 2] = b[i + 1];
        a[i + 3] = b[i + 0];
    }
}
```

• `pshufb` was introduced with Intel SSSE3 feature set extension, e.g.:
  • `-xSSSE3` or `/QxSSSE3`
Idiom Recognition: Saturation

Example:

```c
#define N 1000
void sat_signed_char(char va[N], char vb[N], char vc[N])
{
    int i;
    for (i = 0; i < N; i++) {
        vc[i] = ((vb[i] + va[i] > 127)
                 ? 127
                 : ((vb[i] + va[i] < -128)
                     ? -128
                     : vb[i] + va[i]));
    }
}
```

The compiler will make use of **paddsb** (add packed signed bytes with saturation) because the source code limits both the upper and lower bound of the addition.

For **unsigned char** types a lower bound check can be missing, which is OK!
Vectorization for Complex Arithmetic

• Both C99 and Fortran provide explicit support for complex data types
• Intel compilers can vectorize the corresponding arithmetic instructions using Intel® SSE3 instructions and higher
• Example for C99:

```c
float _Complex zc[10];
float _Complex za = 4 + __I__*2;
    // Real part = 4
    // Imaginary part = 2

void zscale()
{
    for(int i = 0; i < 10; i++)
        zc[i] = za * csin(zc[i]);
}
```

• Compile:
• Linux*, OS X*: `-std=c99 -xsse3`
• Windows*: `/Qstd=c99 /Qxsse3`
Loop Transformations

• Frequently (optimal) vectorization is possible only after adapting the loops beforehand
• The compiler component responsible for these loop transformations is High Level Optimization (HLO) phase of compiler
• While HLO is active for optimization level 2 and higher, with level 3 the full set of transformations is activated and applies the transformations more aggressively
• Intel compilers provide detailed report on HLO activity:
  • Linux*, Mac OS* X: -opt-report-phase=hlo
  • Windows*: /Qopt-report-phase:hlo

... LOOP INTERCHANGE in loops at line: 7 8 9
Loopnest permutation ( 1 2 3 ) --> ( 2 3 1 )

... Loop at line 7 unrolled and jammed by 4
Loop at line 8 unrolled and jammed by 4
...
Example for Loop Transformations

Transformations done by compiler:
1. Outer loop is distributed into 2 loops
2. Nested loop is interchanged to exploit spatial locality on array
3. Outer loop unrolled

Example for Loop Transformations

```c
6: for(int i = 0; i < 100; i++)
7: {
8:   a[i] = 0;
9:   for(int j = 0; j < 100; j++)
10:      a[i] += b[j][i];
11: }
```

```c
for(int i = 0; i < 100; i++)
a[i] = 0;
for(int j = 0; j < 100; j++)
   for(int i = 0; i < 100; i++)
a[i] += b[j][i];
```

HLO REPORT LOG OPENED ON <DATE>

```
<loop.c;-1:-1;hlo;foo;0>
High Level Optimizer Report (foo)
<loop.c;6:6;hlo_distribution;in foo;0>
LOOP DISTRIBUTION in foo at line 6

<loop.c;6:6;hlo_linear_trans;foo;0>
LOOP INTERCHANGE in loops at line: 6 9
Loopnest permutation ( 1 2 ) --> ( 2 1 )

<loop.c;6:6;hlo_reroll;foo;0>
Loop at line:6  memset generated
```
Some HLO Loop Transformations

• For optimization level 3:
  * Loop interchange (for more efficient memory access)
  * Loop unrolling (more instruction level parallelism)
  * Cache blocking (for more reuse of data in cache)
  * Loop peeling (allow for misalignment)
  * Loop versioning (for loop count, data alignment, …)
  * Memcopy recognition (call Intel’s fast `memcpy(…), memset(…)`)
  * Loop splitting (facilitate vectorization)
  * Loop fusion (more efficient vectorization)
  * Scalar expansion (remove dependency)
  * Loop rerolling (enable vectorization)
  * Loop reversal (handle dependencies)

• Red: Partially enabled for level 2.
Student Exercise

• Sample program showing benefit of complex arithmetic vectorization

• Sample program for idiom recognition

• Implement an algorithm using multiple loops (e.g. matrix multiplication) and turn on HLO optimization report. Interpret the messages.

• Compile same example with different optimization levels and HLO optimization reporting on.
Agenda

• Introduction
• Compiler & Vectorization
• Validating Vectorization Success
• Reasons for Vectorization Fails
• Vectorization of Special Program Constructs & Loops
• Intel® Cilk™ Plus
• OpenMP* 4.0
• Guided Auto Parallelism
• Summary
Intel® Cilk™ Plus

Ease of use

1. Compiler:
   Auto-vectorization (no change of code)

2. Compiler:
   Auto-vectorization hints (#pragma vector, ...)

3. Compiler:
   OpenMP* 4.0 and Intel® Cilk™ Plus

- SIMD intrinsic class
  (e.g.: F32vec, F64vec, ...)

- Vector intrinsic
  (e.g.: _mm_fmadd_pd(...), _mm_add_ps(...), ...)

- Assembler code
  (e.g.: [v]addps, [v]addss, ...)

Programmer control
Stumbling Stones for Vectorization*

(*): Not complete; there are way more factors impacting vectorization!

Vectorization for C/C++ is subject of:

• Data Dependency (memory disambiguation/aliasing)
  • Pointers might alias (if same type)
  • Compiler has to assume dependencies
  ⇒ Avoid aliasing side-effects
• Control Flow Dependency:
  • Different operations per vector element
  • Complex & heterogeneous control flow
  ⇒ Avoid side-effects due to order of operations
• Alignment:
  • Unaligned accesses are slow
  • Alignment more problematic for vector ops
  • Guarantee alignment

⇒ Any good vectorization technique must at least address those!

```c
void foo(float *a, float *b, int off)
{
    for(int i = 0; i < …; i++)
        a[i] = a[i + off] * b[i];
}
```

```c
void foo(float *a, float *b)
{
    for(int i = 0; i < …; i++)
        a[i] = (a[i] > 1.0) ?
            a[i] / b[i] :
            a[i];
}
```

```c
void foo(float *a, float *b)
{
    // are arrays “a” or “b” aligned?
}
```
Reference Example

All three problems in one example:

```c
void vec0(float *a, float *b, int off, int len)
{
    __assume(off % 32 == 0);
    __assume_aligned(a, 64);
    __assume_aligned(b, 64);
    for(int i = 0; i < len; i++)
    {
        a[i] = (a[i] > 1.0) ?
                a[i] * b[i] :
                a[i + off] * b[i];
    }
}
```

LOOP BEGIN at simd.cpp(6,5)
remark #15344: loop was not vectorized: vector dependence prevents vectorization
remark #15346: vector dependence: assumed FLOW dependence between a line 8 and a line 8
remark #15346: vector dependence: assumed ANTI dependence between a line 8 and a line 8
LOOP END
Task Level Parallelism

Simple Keywords
Set of keywords, for expression of task parallelism:
cilk_spawn
cilk_sync
cilk_for

Reducers (Hyper-objects)
Reliable access to nonlocal variables without races
cilk::reducer_opadd<int> sum(3);

Data Level Parallelism

Array Notation
Provide data parallelism for sections of arrays or whole arrays
mask[:] = a[:] < b[:] ? -1 : 1;

SIMD-enabled Functions
Define actions that can be applied to whole or parts of arrays or scalars

Execution Parameters
Runtime system APIs, Environment variables, pragmas
**Intel® Cilk™ Plus**

**Task Level Parallelism**

**Simple Keywords**
Set of keywords, for expression of task parallelism:
- `cilk_spawn`
- `cilk_sync`
- `cilk_for`

**Reducers (Hyper-objects)**
Reliable access to nonlocal variables without races
- `cilk::reducer_opadd<int> sum(3);`

**Data Level Parallelism**

**Array Notation**
Provide data parallelism for sections of arrays or whole arrays
- `mask[:]=a[:]<b[:]?-1:1;`

**SIMD-enabled Functions**
Define actions that can be applied to whole or parts of arrays or scalars

**Execution Parameters**
Runtime system APIs, Environment variables, pragmas
C/C++: `#pragma simd [clause [,clause]...]`
Fortran: `!DIR$ SIMD [clause [,clause]...]`

Without any clause, the directive “enforces” vectorization of the loop, ignoring all dependencies (even if they are proved!)

Example:

```c
void addfl(float *a, float *b, float *c, float *d, float *e, int n)
{
    #pragma simd
    for(int i = 0; i < n; i++)
        a[i] = a[i] + b[i] + c[i] + d[i] + e[i];
}
```

Without SIMD directive, vectorization likely fails since there are too many pointer references to do a run-time check for overlapping (compiler heuristic).

The compiler won't create multiple versions here.

Using the directive asserts the compiler that none of the pointers are overlapping.
Restrictions applying SIMD pragma/directive:
• The loop using SIMD pragma/directive has to be a `for` (C/C++) or `DO` loop (FORTRAN) and conform to OpenMP* work-sharing loop construct (see the OpenMP* 3.1 specification, section 2.5.1)
• Induction variable has to be signed/unsigned integer or pointer
• Loop body must be free from C++ exceptions and Windows* Structured Exception Handling, `setjmp(...) & longjmp(...)`
• The vector values must be of POD (plain old data) type, i.e. signed/unsigned 8/16/32/64 bit integer, single/double-precision FP or single/double-precision complex types
• All memory references are treated unconditionally:
• All address computations have to result in valid memory addresses!
• For more information:
#pragma simd

Clauses for C/C++

• `vectorlength(n1 [,n2] ...)`
  • `n1, n2, ...` must be `2, 4, 8, ...`: The compiler can assume a safe vectorization for a vector length of `n1, n2, ...`; alternative: `vectorlengthfor(type)`

• `private(v1, v2, ...)`
  • Variables private to each iteration; supersets (extensions):
    - `firstprivate(...)`: initial value is broadcast to all private instances
    - `lastprivate(...)`: last value is copied out from the last iteration instance

• `linear(v1:step1, v2:step2, ...)`
  • For every iteration of original scalar loop `v1` is incremented by `step1, ...` etc. Therefore it is incremented by `step1 * VL` for the vectorized loop.

• `reduction(operator:v1, v2, ...)`
  • Variables `v1, v2, ...` etc. are reduction variables for operation `operator`

• `[no]assert`
  • Warning (default: `noassert`) or error with failed vectorization
#pragma simd Example for C/C++

```c
void foo(float *restrict a, float *restrict b, int max, int n, int off[n])
{
    #pragma simd vectorlength(4)
    for(int k = 0; k < n - max; k++) a[k + off[k]] = a[k] * b[k];
}
```

- The compiler cannot vectorize the loop, even though the arrays `a` and `b` won't overlap (keyword `restrict`).
- Also multi-versioning won’t help because of the dynamic offsets (`off[]`).
- Using `#pragma ivdep` doesn’t work either because compiler regards accesses to `off[]` as inefficient here.
- Also it’s dangerous, e.g.:
- On AVX the compiler might use a vector length of 8. If any of the offsets is below that there’s a dependency within at least one vector operation!

**Solution:**
If, for example, offsets are at least 4 elements, vectorization is still possible as vector length can be controlled via `#pragma simd`
!DIR$ SIMD Clauses for Fortran

• VECTORLENGTH(n1 [,n2] ...)  
  • n1, n2, ... must be 2, 4, 8, ...: The compiler can assume a safe vectorization for a vector length of n1, n2, ...

• PRIVATE(v1, v2, ...)  
  • Variables private to each iteration; supertsets (extensions):
    FIRSTPRIVATE(...) : initial value is broadcast to all private instances  
    LASTPRIVATE(...) : last value is copied out from the last iteration instance

• LINEAR(v1:step1, v2:step2, ...)  
  • For every iteration of original scalar loop v1 is incremented by step1, ... etc.
    Therefore it is incremented by step1 * VL for the vectorized loop.

• REDUCTION(operator:v1, v2, ...)  
  • Variables v1, v2, ... etc. are reduction variables for operation operator

• [NO]ASSERT  
  • Warning (default: NOASSERT) or error with failed vectorization
**Problem:**
“Enforced” vectorization still fails with the following message:
loop was not vectorized: conditional assignment to a scalar
loop was not vectorized with "simd"

**Solution:**
Clarify that scalar is a reduction with operator +.

**Attention:**
Same as for OpenMP* reduction variables can only be associated to one operator each!

---

```fortran
!DIR$ SIMD
do i = 1,n
  if (a(i) .GT. 0) then
    sum2 = sum2 + a(i) * b(i)
  else
    sum2 = sum2 + a(i)
  endif
enddo
```

```fortran
!DIR$ SIMD REDUCTION(+:sum2)
!DIR$ SIMD
```

```fortran
do i = 1,n
  if (a(i) .GT. 0) then
    sum2 = sum2 + a(i) * b(i)
  else
    sum2 = sum2 + a(i)
  endif
enddo
```
SIMD Pragmas/Directives & OpenMP®

SIMD pragmas/directives are related to OpenMP® semantics:
• When using the auto-parallelization it uses OpenMP® and depends from its features
• When using auto-vectorization it can be influenced by the
• SIMD pragmas/directive (so-called “user mandated vectorization”)
• User mandated vectorization can be configured same as the OpenMP® work-sharing loop construct:
• The clauses have the same semantic as for OpenMP®!
Differences between IVDEP & SIMD pragmas/directives:

• #pragma ivdep (C/C++) or !DIR$ IVDEP (Fortran)
  - Ignore vector dependencies (IVDEP): Ignore assumed but not proven dependencies for a loop
  - Example:
    ```c
    void foo(int *a, int k, int c, int m)
    {
      #pragma ivdep
      for (int i = 0; i < m; i++)
        a[i] = a[i + k] * c;
    }
    ```

• #pragma simd (C/C++) or !DIR$ SIMD (Fortran)
  - Aggressive version of IVDEP: Ignores all dependencies inside a loop and ignore efficiency heuristic
  - It’s an imperative that forces the compiler try everything to vectorize
  - Attention: This can break semantically correct code!
  - However, it can vectorize code legally in some cases that wouldn’t be possible otherwise!
Intel® Cilk™ Plus

**Task Level Parallelism**

**Simple Keywords**
Set of keywords, for expression of task parallelism:
- `cilk_spawn`
- `cilk_sync`
- `cilk_for`

**Reducers**
(Hyper-objects)
Reliable access to nonlocal variables without races
- `cilk::reducer_opadd<int> sum(3);`

**Data Level Parallelism**

**Array Notation**
Provide data parallelism for sections of arrays or whole arrays
- `mask[:,] = a[:,] < b[:,] ? -1 : 1;`

**SIMD-enabled Functions**
Define actions that can be applied to whole or parts of arrays or scalars

**Execution Parameters**
Runtime system APIs, Environment variables, pragmas
SIMD-Enabled Functions Syntax

Windows*:
__declspec(vector([clause [,clause]…]))
     function definition or declaration

Linux*/OS* X:
__attribute__((vector([clause [,clause]…])))
     function definition or declaration

• C/C++ only
• Intent:
  • Express work as scalar operations (kernel) and let compiler create a vector
    version of it. The size of vectors can be specified at compile time (SSE, AVX, …)
    which makes it portable!
• Remember:
  • Both the function definition as well as the function declaration (header file) need
    to be specified like this!
### SIMD-Enabled Functions Clauses

- **processor(cpuid)**
  - `cpuid` for which (Intel) processor to create a vector version
- **vectorlength(len)**
  - `len` must be power of 2: Allow as many elements per argument
- **linear(v1:step1, v2:step2, ...)**
  - Defines `v1, v2, ...` to be private to SIMD lane and to have linear (`step1, step2, ...`) relationship when used in context of a loop
- **uniform(a1, a2, ...)**
  - Arguments `a1, a2, ...` etc. are not treated as vectors (constant values across SIMD lanes)
- **[no]mask**: SIMD-enabled function called only inside branches (masked) or never (not masked)

Intrinsic also available: `__intel_simd_lane()`:

Return the SIMD lane with range: `[0:vector length - 1]`
SIMD-Enabled Functions

Write a function for one element and add `__declspec(vector)`:

```c
__declspec(vector)
float foo(float a, float b, float c, float d)
{
    return a * b + c * d;
}
```

Call

```c
e = foo(a, b, c, d);
```

Call

```c
#pragma simd
for(i = 0; i < n; i++) {
    A[i] = foo(B[i], C[i], D[i], E[i]);
}
```

Call

```c
A[:] = foo(B[:], C[:], D[:], E[:]);
```
## SIMD-Enabled Functions: Invocation

__declspec(vector)float my_simdf (float b) { ... }

<table>
<thead>
<tr>
<th>Construct</th>
<th>Example</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard for loop</td>
<td>for (j = 0; j &lt; N; j++) { a[j] = my_simdf(b[j]); }</td>
<td>Single thread, maybe auto-vectorizable</td>
</tr>
<tr>
<td>#pragma simd</td>
<td>#pragma simd for (j = 0; j &lt; N; j++) { a[j] = my_simdf(b[j]); }</td>
<td>Single thread, vectorized; use the appropriate vector version</td>
</tr>
<tr>
<td>Array notation</td>
<td>a[:] = my_simdf(b[:]);</td>
<td>Single thread, vectorized</td>
</tr>
<tr>
<td>OpenMP* 4.0</td>
<td>#pragma omp parallel for simd for (j = 0; j &lt; N; j++) { a[j] = my_simdf(b[j]); }</td>
<td>Multi-threaded, vectorized</td>
</tr>
</tbody>
</table>
Intel® Cilk™ Plus

**Task Level Parallelism**

**Simple Keywords**
Set of keywords, for expression of task parallelism:
- `cilk_spawn`
- `cilk_sync`
- `cilk_for`

**Reducers (Hyper-objects)**
Reliable access to nonlocal variables without races
```
cilk::reducer_opadd<int> sum(3);
```

**Data Level Parallelism**

**Array Notation**
Provide data parallelism for sections of arrays or whole arrays
```
mask[:,] = a[:,] < b[:,] ? -1 : 1;
```

**SIMD-enabled Functions**
Define actions that can be applied to whole or parts of arrays or scalars

**Execution Parameters**
Runtime system APIs, Environment variables, pragmas
Array Notation Extension: Syntax I

- An extension to C/C++ only
- Perform operations on sections of arrays in parallel
- Example:

```c
for(i = 0; i < ...; i++)
    A[i] = B[i] + C[i];
```

- Well suited for code that:
  - Performs per-element operations on arrays
  - Without an implied order between them (aliasing is ignored)
  - With an intent to execute in vector instructions

Notice: Not exactly the same; Aliasing is ignored by Array Notations!
Array Notation Extension: Syntax II

• Syntax:

- Use a "::" for all elements (if size is known)
- "length" specifies number of elements of subset
- "stride": distance between elements for subset

```
A[:]
A[start_index : length]
A[start_index : length : stride]
```

Explicit Data Parallelism Based on C/C++ Arrays
Array Notation Extension: Example I

Accessing a section of an array:

```c
float a[10], b[6];
...
// allocate *b
...
b[::] = a[2:6];
...```

```
a: 0 1 2 3 4 5 6 7 8 9
```

```
b: 2 3 4 5 6 7
```
Array Notation Extension: Example II

Section of 2D array:

```c
float a[10][10], *b;
...
// allocate *b
...
b[0:10] = a[#:][5];
...```

```
a:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

b:

| 1 | 2 | ... |  |  |  |  |  |
```
Array Notation Extension: Example III

Strided section of an array:

```c
float a[10], *b;
...
// allocate *b
...
b[0:3] = a[0:3:2];
...
```

![Diagram showing array notation example](image_url)
Array Notation Extension: Operators

Most C/C++ operators are available for array sections:
+, -, *, /, %, <, ==, !=, >, |, &, ^, &&, ||, !, - (unary), + (unary),
++, --, +=, -=, *=, /=, * (pointer de-referencing)
Examples:

\[ a[:,] \times b[:,] \quad \text{// element-wise multiplication} \]
\[ a[3:2][3:2] + b[5:2][5:2] \quad \text{// matrix addition} \]
\[ a[0:4][1:2] + b[1:2][0:4] \quad \text{// error, different rank sizes} \]
\[ a[0:4][1:2] + c \quad \text{ // adds scalar c to array section} \]

- Operations on different elements can be executed in parallel without any ordering constraints.
- Array operands must have the same rank and size.
- Scalar operands are automatically expanded.
Array Notation Extension: Reductions

Combine array section elements using a predefined operator, or a user function:

```c
int a[] = {1,2,3,4};
sum = __sec_reduce_add(a[:]); // sum is 10
res = __sec_reduce(0, a[:], func);
    // apply function func to all
    // elements in a[], initial value is 0
int func(int arg1, int arg2)
{
    return arg1 + arg2;
}
```

Other reductions (list not exhaustive):
- __sec_reduce_mul
- __sec_reduce_all_zero
- __sec_reduce_all_nonzero
- __sec_reduce_any_nonzero
- __sec_reduce_max
- __sec_reduce_min
- __sec_reduce_max_ind
- __sec_reduce_min_ind

Much more! Take a look at the specification:
https://www.cilkplus.org/sites/default/files/open_specifications/Intel_Cilk_plus_language_spec_1.2.htm
Array Notation Extension: Example I

Serial version:

```c
float dot_product(unsigned int size, float A[size], float B[size])
{
    int i;
    float dp = 0.0f;
    for (i=0; i<size; i++) {
        dp += A[i] * B[i];
    }
    return dp;
}
```

Array notation:

```c
float dot_product(unsigned int size, float A[size], float B[size])
{
    // A[:] can also be written as A[0:size]
    return __sec_reduce_add(A[:]) * B[:];
}
```
Array Notation Extension: Example II

Ignore data dependencies, indirectly mitigate control flow dependence & assert alignment:

```c
void vec3(float *a, float *b, int off, int len)
{
    __assume_aligned(a, 64);
    __assume_aligned(b, 64);
    a[0:len] = (a[0:len] > 1.0) ?
              a[0:len] * b[0:len] :
              a[off:len] * b[0:len];
}
```

LOOP BEGIN at simd.cpp(5,9)
remark #15388: vectorization support: reference a has aligned access  [ simd.cpp(5,28) ]
remark #15388: vectorization support: reference b has aligned access  [ simd.cpp(5,28) ]
...
remark #15300: LOOP WAS VECTORIZED
...
LOOP END
Intel® Cilk™ Plus

Compilers

The following compilers support Intel® Cilk™ Plus:

- **GNU* GCC 4.9:**
  - Exception: `_cilk_for` (Thread Level Parallelism) which will be added with GCC 5.0
  - Enable with `-fcilkplus`

- **clang/LLVM 3.5:**
  - Not official yet but development branch exists: [http://cilkplus.github.io/](http://cilkplus.github.io/)
  - Enable with `-fcilkplus`

- **Intel® C++/Fortran Compiler:**
  - Beginning with 12.0; newer features added over time (see Release Notes)
Agenda

- Introduction
- Compiler & Vectorization
- Validating Vectorization Success
- Reasons for Vectorization Fails
- Vectorization of Special Program Constructs & Loops
- Intel® Cilk™ Plus
- OpenMP® 4.0
- Guided Auto Parallelism
- Summary
Intel® Cilk™ Plus

**Ease of use**

**Compiler:**
Auto-vectorization (no change of code)

**Compiler:**
Auto-vectorization hints (#pragma vector, ...)

**Compiler:**
OpenMP* 4.0 and Intel® Cilk™ Plus

**SIMD intrinsic class**
(e.g.: F32vec, F64vec, ...)

**Vector intrinsic**
(e.g.: _mm_fmadd_pd(...), _mm_add_ps(...), ...)

**Assembler code**
(e.g.: [v]addps, [v]addss, ...)
OpenMP* 4.0

• OpenMP* 4.0 ratified July 2013
• Specifications:
• Well established in HPC – parallelism is critical there
• Extension to C/C++ & Fortran
• New features with 4.0:
  • Target Constructs: Accelerator support
  • Distribute Constructs/Teams: Better hierarchical assignment of workers
  • SIMD (Data Level Parallelism!)
  • Task Groups/Dependencies: Runtime task dependencies & synchronization
  • Affinity: Pinning workers to cores/HW threads
  • Cancelation Points/Cancel: Defined abort locations for workers
  • User Defined Reductions: Create own reductions
Pragma SIMD:

- The simd construct can be applied to a loop to indicate that the loop can be transformed into a SIMD loop (that is, multiple iterations of the loop can be executed concurrently using SIMD instructions).

[OpenMP* 4.0 API: 2.8.1]

- Syntax:

  ```c
  #pragma omp simd [clause [,clause]…]
  ```

- **for-loop**

  - For-loop has to be in “canonical loop form” (see OpenMP 4.0 API:2.6)
    - Random access iterators required for induction variable
    - (integer types or pointers for C++)
    - Limited test and in-/decrement for induction variable
    - Iteration count known before execution of loop
    - …
Pragma SIMD Clauses

• safelen(n1[,n2] ...)
  • n1, n2, ... must be power of 2: The compiler can assume a vectorization for a vector length of n1, n2, ... to be safe
• private(v1, v2, ...) : Variables private to each iteration
  • lastprivate(...) : last value is copied out from the last iteration instance
• linear(v1:step1, v2:step2, ...) :
  • For every iteration of original scalar loop v1 is incremented by step1, ... etc. Therefore it is incremented by step1 * vector length for the vectorized loop.
• reduction(operator:v1, v2, ...) :
  • Variables v1, v2, ... etc. are reduction variables for operation operator
• collapse(n) : Combine nested loops – collapse them
• aligned(v1:base, v2:base, ...) : Tell variables v1, v2, ... are aligned;
  (default is architecture specific alignment)
Pragma SIMD Example

Ignore data dependencies, indirectly mitigate control flow dependence & assert alignment:

```c
void vec1(float *a, float *b, int off, int len)
{
    #pragma omp simd safelen(32) aligned(a:64, b:64)
    for(int i = 0; i < len; i++)
    {
        a[i] = (a[i] > 1.0) ?
            a[i] * b[i] :
            a[i + off] * b[i];
    }
}
```

LOOP BEGIN at simd.cpp(4,5)
remark #15388: vectorization support: reference a has aligned access [ simd.cpp(6,9) ]
remark #15388: vectorization support: reference b has aligned access [ simd.cpp(6,9) ]
...
remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
...
LOOP END
SIMD-Enabled Functions

• SIMD-Enabled Function (aka. declare simd construct):
  The declare simd construct can be applied to a function […] to enable the creation of one or more versions that can process multiple arguments using SIMD instructions from a single invocation from a SIMD loop.
  [OpenMP* 4.0 API: 2.8.2]

• Syntax:
  • #pragma omp declare simd [clause [,clause]…]
  • function definition or declaration

• Intent:
  Express work as scalar operations (kernel) and let compiler create a vector version of it. The size of vectors can be specified at compile time (SSE, AVX, …) which makes it portable!

• Remember:
  • Both the function definition as well as the function declaration (header file) need to be specified like this!
SIMD-Enabled Function Clauses

• **simdlen(len)**
  • len must be power of 2: Allow as many elements per argument
  • (default is implementation specific)

• **linear(v1:step1, v2:step2, …)**
  • Defines v1, v2, … to be private to SIMD lane and to have linear (step1, step2, …) relationship when used in context of a loop

• **uniform(a1, a2, …)**
  • Arguments a1, a2, … etc. are not treated as vectors (constant values across SIMD lanes)

• **inbranch, notinbranch**: SIMD-enabled function called only inside branches or never

• **aligned(a1:base, a2:base, …)**: Tell arguments a1, a2, … are aligned;
  • (default is architecture specific alignment)
SIMD-Enabled Function Example

Ignore data dependencies, indirectly mitigate control flow dependence & assert alignment:

```c
#include <omp.h>

float work(float *a, float *b, int i, int off)
{
    return (a[i] > 1.0) ? a[i] * b[i] : a[i + off] * b[i];
}

void vec2(float *a, float *b, int off, int len)
{
    #pragma omp simd safelen(64) aligned(a:64, b:64)
    for(int i = 0; i < len; i++)
    {
        a[i] = work(a, b, i, off);
    }
}
```

INLINE REPORT: (vec2(float *, float *, int, int)) [4/9=44.4%] simd.cpp(8,1)
-> INLINE: (12,16) work(float *, float *, int, int) (isz = 18) (sz = 31)

LOOP BEGIN at simd.cpp(10,5)
remark #15388: vectorization support: reference a has aligned access [ simd.cpp(4,20) ]
remark #15388: vectorization support: reference b has aligned access [ simd.cpp(4,20) ]
remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
...

LOOP END
OpenMP* 4.0: Compilers

The following compilers support OpenMP* 4.0:

- GNU* GCC 4.9:
  - 4.9 for C/C++ (4.9.1 for Fortran); no accelerator support (yet)

- clang/LLVM 3.5:
  - Not official yet but development branch exists: [http://clang-omp.github.io/](http://clang-omp.github.io/)

- Intel® C++/Fortran Compiler:
  - Beginning with 14.0; full 15.0 (except user defined reductions)

SIMD extensions require at least `-fopenmp-simd` (or `-fopenmp`)!
OpenMP* runtime is not needed, though.
Agenda

• Introduction
• Compiler & Vectorization
• Validating Vectorization Success
• Reasons for Vectorization Fails
• Vectorization of Special Program Constructs & Loops
• Intel® Cilk™ Plus
• OpenMP* 4.0
• Guided Auto Parallelism
• Summary
Guided Auto Parallelism (GAP) key design:

- Use compiler infrastructure to output diagnostic messages to developer to detect what is blocking certain optimizations (e.g. vectorization, parallelization, data transformations)
- Diagnostic message contain rich description of the found problems with examples and possible solutions for failed vectorization and parallelization
- Not a separate tool but additional feature of Intel Compilers
- Developers can take advantage from experience of compiler engineers who are faced with performance tuning of numerous applications, benchmarks and compute kernels every day
Guided Auto Parallelism II

• What it is **not**:  
  - No automatic vectorizer or parallelizer (no code is generated at all)  
  - GAP does not ask the programmer to change algorithms, transformation ordering or internal heuristics of compiler; it only gives feedback on what’s currently provided and with the option set provided for the build  
  - (implementation level, not architectural level)  
  - It is restricted to information from the source code and compiler options (e.g. no runtime information, no architectural model)
GAP Workflow

• Identify hotspots and/or problematic code sections (optional):
  - Use two-step GAP to improve either the whole application or some code sections:

  1. Identify hotspots and/or problematic code sections (optional):
     - GAP seamlessly integrates into existing build framework

  2. Use two-step GAP to improve either the whole application or some code sections:
     - Improved Binary
     - Modified Sources

C/C++ or Fortran Sources

Compiler

Binary and Reports

Performance Analysis Tools

Hotspots or Problems

C/C++ or Fortran Sources

Compiler

GAP Advice Messages

Modified Sources

GAP Advice Messages

Improved Binary

Compiler

GAP Advice Messages

Modified Sources

C/C++ or Fortran Sources

Compiler

Binary and Reports

Performance Analysis Tools

Hotspots or Problems

C/C++ or Fortran Sources

Compiler

Binary and Reports

Performance Analysis Tools

Hotspots or Problems

C/C++ or Fortran Sources

Compiler

Binary and Reports

Performance Analysis Tools

Hotspots or Problems

C/C++ or Fortran Sources

Compiler

Binary and Reports

Performance Analysis Tools

Hotspots or Problems
Turn on GAP I

• Activate GAP and optionally define guidance level:
  • Linux*, OS X*: `-guide [=level]`, Windows*: `/Qguide[:,level]
• Activate GAP individually:
  • Auto-vectorization:
    • Linux*, OS X*: `-guide-vec [=level]
    • Windows*: `/Qguide-vec[:,level]
  • Auto-parallelization:
    • Linux*, OS X*: `-guide-par [=level]
    • Windows*: `/Qguide-par[:,level]
  • Data transformations:
    • Linux*, OS X*: `-guide-data-trans [=level]
    • Windows*: `/Qguide-data-trans[:,level]
• Optional argument `level` can be 1-4 to control extend of analysis;
• the higher the more advanced (default: 4)
• To receive auto-parallelization guidance `-parallel` (Linux*, OS X*) or `/Qparallel` (Windows*) must be set
Turn on GAP II

• Control the source code part for which analysis is done:
  • Linux*, OS X*: `-guide-opts=<string>`, Windows*: `/Qguide-opts:<string>`
  • Examples for `<string>`:
    - `init.c`, 1-50, 100-150:
      Restrict analysis to file `init.c`, lines 1-50 and 100-150
    - `bar.f90`, 'm1::func_solve`:
      Restrict analysis to file `bar.f90`, Fortran module `m1` and function `func_solve`

• Control where the message are going:
  • New file:
    - Linux*, OS X*: `-guide-file=<file>`
    - Windows*: `/Qguide-file:<file>`
  • Append to existing file:
    - Linux*, OS X*: `-guide-file-append=<file>`
    - Windows*: `/Qguide-file-append:<file>`
GAP Example for Vectorization I

```c
void f(int n, float *x, float *y, float *z, float *d1, float *d2) {
    for(int i = 0; i < n; i++)
        z[i] = x[i] + y[i] - (d1[i] * d2[i]);
}
```

GAP REPORT LOG OPENED ON <DATE>

remark #30761: Add -parallel option if you want the compiler to generate recommendations for improving auto-parallelization.

gap.c(3): remark #30536: (LOOP) Add -fargument-noalias option for better type-based disambiguation analysis by the compiler, if appropriate (the option will apply for the entire compilation). This will improve optimizations such as vectorization for the loop at line 3. [VERIFY] Make sure that the semantics of this option is obeyed for the entire compilation. [ALTERNATIVE] Another way to get the same effect is to add the "restrict" keyword to each pointer-typed formal parameter of the routine "f". This allows optimizations such as vectorization to be applied to the loop at line 3. [VERIFY] Make sure that semantics of the "restrict" pointer qualifier is satisfied: in the routine, all data accessed through the pointer must not be accessed through any other pointer.

Number of advice-messages emitted for this compilation session: 1.
END OF GAP REPORT LOG
GAP Example for Vectorization II

```c
void mul(NetEnv* ne, Vector* rslt, Vector* den, Vector* flux1,
         Vector* flux2, Vector* num)
{
    int i;
    float *r, *d, *n, *s1, *s2;
    r = rslt->data;
    d = den->data;
    n = num->data;
    s1 = flux1->data;
    s2= flux2->data;

    for(i = 0; i < ne->len; i++)
        r[i] = s1[i] * s2[i] + n[i] * d[i];
}
```

1. Use a local variable to store the upper-bound of loop `#len` instead of loading it for each iteration iff loop invariant.

2. Use `#pragma ivdep` to aid vectorization of loop if the following arrays do not have loop-carried dependencies: `r, s1, s2, n, d`

Loop can be vectorized when both improvements are done!
GAP Example for Parallelization

#define N 10000
double A[N], B[N];
int bar(int);

void foo()
{
    int i;
    for(i = 0; i < N; i++)
        A[i] = B[i] * bar(i);
}

1> GAP REPORT LOG OPENED ON <DATE>

1> gap.cpp(8): warning #30528: (PAR) Add "__declspec(const)" to the declaration of 
routine "bar" in order to parallelize the loop at line 7. Alternatively, adding 
"__declspec(concurrency_safe(profitable))" achieves a similar effect. [VERIFY] Make 
sure that the routine satisfies the semantics of this declaration. [ALTERNATIVE] Yet 
another way to help the loop being parallelized is to inline the routine with "#pragma 
forceinline recursive". This method does not guarantee parallelization.
1>
1> Number of advice-messages emitted for this compilation session: 1.
1> END OF GAP REPORT LOG
GAP Example for Data Transformation

```c
struct S {
   int a;
   int useless;
   int b;
};
...
for(int i = 0; i < N; i++) {
   sp->a = i;
   sp->b = i + 1;
   sp++;
}
...
```

GAP REPORT LOG OPENED ON <DATE>

gap.c(1): remark #30758: (DTRANS) Remove unused field(s) 'useless' from the struct 'S'. [VERIFY] The suggestion is based on the field references in the current compilation. Please make sure that there are no references to these fields across the entire application.

Number of advice-messages emitted for this compilation session: 1.

END OF GAP REPORT LOG
Agenda

• Introduction
• Compiler & Vectorization
• Validating Vectorization Success
• Reasons for Vectorization Fails
• Vectorization of Special Program Constructs & Loops
• Intel® Cilk™ Plus
• OpenMP* 4.0
• Guided Auto Parallelism
• Summary
Summary

- Intel® C++ Compiler and Intel® Fortran Compiler provide sophisticated and flexible support for vectorization
- They also provide a rich set of reporting features that help verifying vectorization and optimization in general
- Directives and compiler switches permit fine-tuning for vectorization
- Vectorization can even be enforced for certain cases where language standards are too restrictive
- Understanding of concepts like dependency and alignment is required to take advantage from SIMD features
- Intel® C++/Fortran Compiler can create multi-version code to address a broad range of processor generations, Intel and non-Intel processors and individually exploiting their feature set
References

• Aart Bik: “The Software Vectorization Handbook”
• http://www.intel.com/intelpress/sum_vmmx.htm
• Randy Allen, Ken Kennedy: “Optimizing Compilers for Modern Architectures: A Dependence-based Approach”
• Steven S. Muchnik, “Advanced Compiler Design and Implementation”
• Intel Software Forums, Knowledge Base, White Papers, Tools Support (see http://software.intel.com)
• Sample Articles:
  http://software.intel.com/en-us/articles/requirements-for-vectorizable-loops/
Thank you!
INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Copyright © 2015, Intel Corporation. All rights reserved. Intel, Pentium, Xeon, Xeon Phi, Core, VTune, Cilk, and the Intel logo are trademarks of Intel Corporation in the U.S. and other countries.

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804