Loop Independence, Compiler Vectorization and Threading of Loops (SSE & AVX)

Michael Klemm
Software & Services Group
Developer Relations Division
Legal Disclaimer

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL’S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked “reserved” or “undefined.” Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to: http://www.intel.com/design/literature.htm
## Optimization Notice

Intel® compilers, associated libraries and associated development tools may include or utilize options that optimize for instruction sets that are available in both Intel® and non-Intel microprocessors (for example SIMD instruction sets), but do not optimize equally for non-Intel microprocessors. In addition, certain compiler options for Intel compilers, including some that are not specific to Intel micro-architecture, are reserved for Intel microprocessors. For a detailed description of Intel compiler options, including the instruction sets and specific microprocessors they implicate, please refer to the “Intel® Compiler User and Reference Guides” under “Compiler Options." Many library routines that are part of Intel® compiler products are more highly optimized for Intel microprocessors than for other microprocessors. While the compilers and libraries in Intel® compiler products offer optimizations for both Intel and Intel-compatible microprocessors, depending on the options you select, your code and other factors, you likely will get extra performance on Intel microprocessors.

Intel® compilers, associated libraries and associated development tools 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 Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3 (Intel® 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.

While Intel believes our compilers and libraries are excellent choices to assist in obtaining the best performance on Intel® and non-Intel microprocessors, Intel recommends that you evaluate other compilers and libraries to determine which best meet your requirements. We hope to win your business by striving to offer the best performance of any compiler or library; please let us know if you find we do not.

Notice revision #20101101
Agenda

- Vector Instructions (SIMD)
- Compiler Switches for Optimization
- Controlling Auto-vectorization
- Controlling Auto-parallelization
- Manual Vectorization (SSE & AVX)
- Hands-on
Agenda

• Vector Instructions (SIMD)
• Compiler Switches for Optimization
• Controlling Auto-vectorization
• Controlling Auto-parallelization
• Manual Vectorization (SSE & AVX)
• Hands-on
What is SSE (and related instruction sets)

- **SSE**: Streaming SIMD extension
- **SIMD**: Single instruction, Multiple Data (Flynn’s Taxonomy)
- **SSE** allows the identical treatment of 2 double, 4 floats and 4 integers at the same time

![Diagram showing vector operations](image)
SSE Data Types

- 2x double
- 4x float
- 16x byte
- 8x short
- 4x integer32
- 2x integer64

128 bit
AVX Data Types

Lane 1

Lane 0
Agenda

- Vector Instructions (SIMD)
- Compiler Switches for Optimization
- Controlling Auto-vectorization
- Controlling Auto-parallelization
- Manual Vectorization (SSE & AVX)
- Hands-on
Intel® Compiler Architecture

C++
Front End

FORTRAN
Front End

Profiler

Interprocedural analysis and optimizations: inlining, constant prop, whole program detect, mod/ref, points-to

Loop optimizations: data deps, prefetch, vectorizer, unroll/interchange/fusion/dist, auto-parallel/OpenMP

Global scalar optimizations: partial redundancy elim, dead store elim, strength reduction, dead code elim

Code generation: vectorization, software pipelining, global scheduling, register allocation, code generation

Disambiguation: types, array, pointer, structure, directives
# A few General Switches

<table>
<thead>
<tr>
<th>Functionality</th>
<th>Linux</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disable optimization</td>
<td>-O0</td>
</tr>
<tr>
<td>Optimize for speed (no code size increase), no SWP</td>
<td>-O1</td>
</tr>
<tr>
<td>Optimize for speed (default)</td>
<td>-O2</td>
</tr>
<tr>
<td>High-level optimizer (e.g. loop unroll), -ftz (for Itanium)</td>
<td>-O3</td>
</tr>
<tr>
<td>Vectorization for x86, -xSSE2 is default</td>
<td>&lt;many options&gt;</td>
</tr>
<tr>
<td>Aggressive optimizations (e.g. -ipo, -O3, -no-prec-div, -static -xHost for x86 Linux*)</td>
<td>-fast</td>
</tr>
<tr>
<td>Create symbols for debugging</td>
<td>-g</td>
</tr>
<tr>
<td>Generate assembly files</td>
<td>-S</td>
</tr>
<tr>
<td>Optimization report generation</td>
<td>-opt-report</td>
</tr>
<tr>
<td>OpenMP support</td>
<td>-openmp</td>
</tr>
<tr>
<td>Automatic parallelization for OpenMP* threading</td>
<td>-parallel</td>
</tr>
</tbody>
</table>
# Architecture Specific Switches

<table>
<thead>
<tr>
<th>Functionality</th>
<th>Linux</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimize for current machine</td>
<td>-xHOST</td>
</tr>
<tr>
<td>Generate SSE v1 code</td>
<td>-xSSE1</td>
</tr>
<tr>
<td>Generate SSE v2 code (may also emit SSE v1 code)</td>
<td>-xSSE2</td>
</tr>
<tr>
<td>Generate SSE v3 code (may also emit SSE v1 and v2 code)</td>
<td>-xSSE3</td>
</tr>
<tr>
<td>Generate SSE v3 code for Atom-based processors</td>
<td>-xSSE_ATOM</td>
</tr>
<tr>
<td>Generate SSSE v3 code (may also emit SSE v1, v2, and v3 code)</td>
<td>-xSSSE3</td>
</tr>
<tr>
<td>Generate SSE4.1 code (may also emit (S)SSE v1, v2, and v3 code)</td>
<td>-xSSE4.1</td>
</tr>
<tr>
<td>Generate SSE4.2 code (may also emit (S)SSE v1, v2, v3, and v4 code)</td>
<td>-xSSE4.2</td>
</tr>
<tr>
<td>Generate AVX code</td>
<td>-xAVX</td>
</tr>
</tbody>
</table>
Interprocedural Optimization
Extends optimizations across file boundaries

Without IPO

- Compile & Optimize → file1.c
- Compile & Optimize → file2.c
- Compile & Optimize → file3.c
- Compile & Optimize → file4.c

With IPO

- Compile & Optimize

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-ip</td>
<td>Only between modules of one source file</td>
</tr>
<tr>
<td>-ipo</td>
<td>Modules of multiple files/whole application</td>
</tr>
</tbody>
</table>
IPO – A Multi-pass Optimization

A Two-Step Process

<table>
<thead>
<tr>
<th>Compiling</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Mac*/Linux*</td>
<td>icc -c -ipo main.c func1.c func2.c</td>
</tr>
<tr>
<td>Windows*</td>
<td>icl -c /Qipo main.c func1.c func2.c</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Linking</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Mac*/Linux*</td>
<td>icc -ipo main.o func1.o func2.o</td>
</tr>
<tr>
<td>Windows*</td>
<td>icl /Qipo main.o func1.o func2.o</td>
</tr>
</tbody>
</table>

Pass 1

Pass 2

virtual .o

executable
What you should know about IPO

• O2 and O3 activate “almost” file-local IPO (-ip)
  – Only a very few, time-consuming IP-optimizations are not done but for most codes, -ip is not adding anything
  – Switch -ip-no-inlining disables in-lining

• IPO extends compilation time and memory usage
  – See compiler manual when running into limitations

• Inlining of functions is the most important feature of IPO but there is much more
  – Inter-procedural constant propagation
  – MOD/REF analysis (for dependence analysis)
  – Routine attribute propagation
  – Dead code elimination
  – Induction variable recognition
  – …many, many more
Profile-Guided Optimizations (PGO)

- Use execution-time feedback to guide (final) optimization
- Helps I-cache, paging, branch-prediction
- Enabled optimizations:
  - Basic block ordering
  - Better register allocation
  - Better decision on which functions to inline
  - Function ordering
  - Switch-statement optimization
PGO Usage: Three Step Process

Step 1
Instrumented Compilation
icc -prof_gen prog.c
Instrumented executable: prog.exe

Step 2
Instrumented Execution
prog.exe (on a typical dataset)
DYN file containing dynamic info: .dyn

Step 3
Feedback Compilation
icc -prof_use prog.c
Merged DYN summary file: .dpi
Delete old dyn files unless you want their info included
Simple PGO Example: Code Re-Order

```c
for (i=0; i < NUM_BLOCKS; i++)
{
    switch (check3(i))
    {
        case 3:                  /* 25% */
            x[i] = 3;  break;
        case 10:                   /* 75% */
            x[i] = i+10;  break;
        default:                   /* 0% */
            x[i] = 99;  break
    }
}
```

“Case 10” is moved to the beginning
– PGO can eliminate most tests&jumps for the common case - less branch mispredicts
What you should know about PGO

- Instrumentation run can be up to twice as long
  - In-lining disabled, trace calls overhead
- Sometimes trace-files cannot be found
  - Looking at right directory?
  - Clean exit() call is necessary to dump info
    - Debugger can help / break in PGO trace start/end calls
- Benefit depends on control flow structure:

  ![Diagram showing significant benefit versus little benefit](image-url)
Memory Reference Disambiguation
Options/Directives related to Aliasing

• -alias_args[-]
• -ansi_alias[-]
• -fno-alias: No aliasing in whole program
• -fno-fnalias: No aliasing within single units
• -restrict (C99): -restrict and restrict attribute
  – enables selective pointer disambiguation
• -safe_cray_ptr: No aliasing introduced by Cray-pointers
• -assume dummy_alias
• Related: Switch -ipo and directive IFDEP
Optimization Report Options

- `opt_report`
  - generate an optimization report to stderr (or file)
- `opt_report_file <file>`
  - specify the filename for the generated report
- `opt_report_phase <phase_name>`
  - specify the phase that reports are generated against
- `opt_report_routine <name>`
  - reports on routines containing the given name
- `opt_report_help`
  - display the optimization phases available for reporting
- `vec-report<level>`
  - Generate vectorization report (IA32, EM64T)
## What did the Compiler do?

<table>
<thead>
<tr>
<th>ipo</th>
<th>ilo</th>
<th>hpo</th>
</tr>
</thead>
<tbody>
<tr>
<td>ipo_inl</td>
<td>ilo_lowering</td>
<td>hpo_analysis</td>
</tr>
<tr>
<td>ipo_cp</td>
<td>ilo_strength_reduction</td>
<td>hpo_openmp</td>
</tr>
<tr>
<td>ipo_align</td>
<td>ilo_reassociation</td>
<td>hpo_vectorization</td>
</tr>
<tr>
<td>ipo_modref</td>
<td>ilo_copy_propagation</td>
<td>hpo_threadization</td>
</tr>
<tr>
<td>ipo_lpt</td>
<td>ilo_convert_insertion</td>
<td></td>
</tr>
<tr>
<td>ipo_subst</td>
<td>ilo_convert_removal</td>
<td></td>
</tr>
<tr>
<td>ipo_vaddr</td>
<td></td>
<td>ecg</td>
</tr>
<tr>
<td>ipo_psplit</td>
<td></td>
<td>ecg_code_cycles</td>
</tr>
<tr>
<td>ipo_gprel</td>
<td>hlo</td>
<td>ecg_code_size</td>
</tr>
<tr>
<td>ipo_pmerge</td>
<td>hlo_scalar_expansion</td>
<td>ecg_predication</td>
</tr>
<tr>
<td>ipo_fps</td>
<td>hlo_unroll</td>
<td>ecg_profiling</td>
</tr>
<tr>
<td>ipo_ppi</td>
<td>hlo_distribution</td>
<td>ecg_inl</td>
</tr>
<tr>
<td>ipo_unref</td>
<td>hlo_fusion</td>
<td>ecg_copy_propagation</td>
</tr>
<tr>
<td>ipo_wp</td>
<td>hlo_prefetch</td>
<td>ecg_brh</td>
</tr>
<tr>
<td>ipo_wp</td>
<td>hlo_loop_collapsing</td>
<td>ecg_speculation</td>
</tr>
<tr>
<td>pgo</td>
<td>hlo_reroll</td>
<td>ecg_swep</td>
</tr>
<tr>
<td></td>
<td>hlo_loadpair</td>
<td>(ecg*, loadpair for IPF only)</td>
</tr>
</tbody>
</table>

List of most phases the user can ask to get detailed reports from.

Only a few phases are relevant for the typical compiler user but these are really helpful! Make use of it - much easier than assembler code inspection.
Sample HLO Report

icc -O3 -opt_report -opt_report_phase hlo

... LOOP INTERCHANGE in loops at line: 7 8 9
Loopnest permutation ( 1 2 3 ) --> ( 2 3 1 )
LOOP INTERCHANGE in loops at line: 15 17
Loopnest permutation ( 1 2 3 ) --> ( 3 2 1 )
...

Loop at line 7 unrolled and jammed by 4
Loop at line 8 unrolled and jammed by 4
Loop at line 15 unrolled and jammed by 4
Loop at line 16 unrolled and jammed by 4
...
# Some Intel-Specific Compiler Directives

<table>
<thead>
<tr>
<th>#pragma</th>
<th>Architecture</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vector/novector</td>
<td>IA-32, Intel64</td>
<td>Indicates to the compiler that the loop should (not) be vectorized overriding the compiler’s heuristics</td>
</tr>
<tr>
<td>loop count(n)</td>
<td>IA-32, Intel64, IA-64</td>
<td>Place before a loop to communicate the approximate number of iterations the loop will execute. Affects software pipelining, vectorization and other loop transformations.</td>
</tr>
<tr>
<td>distribute point</td>
<td>IA-32, Intel64, IA-64</td>
<td>Placed before a loop, the compiler will attempt to distribute the loop based on its internal heuristic. Placed within a loop, the compiler will attempt to distribute the loop at the point of the pragma. All loop-carried dependencies will be ignored.</td>
</tr>
<tr>
<td>unroll, unroll(n), nounroll</td>
<td>IA-32, Intel64, IA-64</td>
<td>Place before an inner loop (ignored on non-inmost loops). #pragma unroll without a count allows the compiler to determine the unroll factor. #pragma unroll(n) tell the compiler to unroll the loop n times. #pragma nounroll is the same as #pragma unroll(0).</td>
</tr>
<tr>
<td>loop count n</td>
<td>IA-32, Intel64, IA-64</td>
<td>Hint to the compile on expected loop iteration count: (n): always n, (n1, n2, n3 ...) either n1, n2, n3 ... expected minimum count &lt;l&gt;, average count &lt;a&gt; and maximal count &lt;u&gt;</td>
</tr>
<tr>
<td>ivdep</td>
<td>IA-32, Intel64, IA-64</td>
<td>Place before a loop to control vectorization/software pipelining. The compiler is instructed to ignore “assumed” (not proven) dependencies preventing vectorization/software pipelining. For Itanium: Assume no BACKWARD dependencies, FORWARD loop-carried dependencies still can exist w/o preventing Swp. Use with –ivdep_parallel option to exclude loop-carried dependencies completely (e.g. for indirect addressing)</td>
</tr>
</tbody>
</table>
C/C++ Compiler for Linux*

• Main goals/values of Intel C/C++ Linux Compiler:
  – Compatible to GCC
  – In general much faster than GCC
    • In particular for typical HPC applications
  – Provide critical features not offered by GCC like support for latest Intel® processors

• Compatibility splits into 3 parts:
  – C/C++ source code language acceptance
    • Almost done; missing features are questionable (e.g. nested function in C supported by GCC)
  – Switch-compatibility
    • Achieved for most relevant options
  – Object Code interoperability
    • 100% reached today – even for complex C++ and OpenMP*

• The ultimate compatibility test: Linux kernel build
  – No manual changes to source code required but requires ‘wrapper’ script
Agenda

• Vector Instructions (SIMD)
• Compiler Switches for Optimization
• **Controlling Auto-vectorization**
• Controlling Auto-parallelization
• Manual Vectorization (SSE & AVX)
• Hands-on
Auto-vectorization

- **Key requirements**
  - A compiler must not alter the program semantics
  - If the compiler cannot determine all dependencies, it has to forego vectorization

- **Compilers sometimes need to act very conservatively**
  - Pointers make it hard for the compiler to deduce memory layout
  - Codes may produce overlapping arrays through pointer arithmetics
  - If the compiler can’t tell, it does not vectorize
Data Dependencies

• Suppose two statements S1 and S2
• S2 depends on S1, iff S1 must be executed before S2
  – Control-flow dependence
  – Data dependence
  – Dependencies can be carried over between loop iterations

• Flavors of data dependencies

FLOW

s1: a = 40
b = 21
s2: c = a + 2

ANTI

b = 40
s1: a = b + 1
s2: b = 21
Compiler Options for Vectorization

• Vectorization is automatically enabled for O2
• Can be controlled through command line switches:
  - vec enable vectorization
  - no-vec disable vectorization
• Vectorization reports help find out about (non-)vectorized code
  - vec-reportn enable vectorization diagnostics
    0 report no diagnostic information.
    1 report on vectorized loops (default)
    2 report on vectorized and non-vectorized loops.
    3 report on vectorized and non-vectorized loops and any proven or assumed data dependences.
    4 report on non-vectorized loops.
    5 report on non-vectorized loops and the reason why they were not vectorized.
source /opt/intel/Compiler/11.1/064/bin/iccvarsh.sh intel64
Vectorization Hints

• Sometimes, the compiler needs some help when looking at your code

• Compiler pragmas:
  – #pragma ivdep
    Indicate that there is no loop-carried dependence in the loop
  – #pragma vector always | aligned | unaligned
    Compiler is instructed to always vectorize a loop (and ignore internal heuristics)
    • always: always vectorize
    • aligned: use aligned load/store instructions
    • anligned: use unaligned load/store instructions
Position of SIMD Features

- Fully automatic vectorization
- Auto vectorization hints (#pragma ivdep)
- SIMD feature (#pragma simd and simd function annotation)
- SIMD intrinsic class (F32vec4 add)
- Vector intrinsic (mm_add_ps())
- ASM code (addps)

Ease of use

Programmer control
Why Didn’t My Loop Vectorize?

• Linux
  
  -vec-report

• Windows
  
  /Qvec-report

• Set diagnostic level dumped to stdout

  n=0: No diagnostic information
  n=1: (Default) Loops successfully vectorized
  n=2: Loops not vectorized – and the reason why not
  n=3: Adds dependency Information
  n=4: Reports only non-vectorized loops
  n=5: Reports only non-vectorized loops and adds dependency info
SIMD Extension for Vector-Level Parallelism

User-mandated vectorization using new **SIMD Pragma** and **SIMD Function Annotation**

- SIMD pragma provides additional information to compiler to enable vectorization of loops (at this time only inner loop)
- SIMD Function Annotations add inter-procedural information to facilitate vectorization
- Supplements automatic vectorization but differently to what traditional pragmas ("automatic vectorization hints") like IVDEP, VECTOR ALWAYS, etc do
  - Traditional pragmas: A hint; not necessary overriding compiler’s heuristic
  - New SIMD extensions: More like an assertion: in case vectorization still fails, it is considered a fault (an option controls whether it is really treated as error);

• Relationship similar to OpenMP versus automatic parallelization

![Diagram showing relationship between SIMD Extension and OpenMP]
foo(float *a, float *b, float *c, int n)
{
    for (int k=0; k<n; k++)   c[k] = a[k] + b[k];
}

Due to the overlapping nature of array accesses from the different call sites, it
might not be semantically correct to use restrict keyword or IVDEP directive
( there are dependencies between iterations for one call )

But it might be true for all calls, that e.g 4 consecutive iterations can be executed
in parallel without violating any dependencies

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

- **No clause** – that is: #pragma simd
  - Enforce vectorization; ignore dependencies etc
- **vectorlength(n₁, n₂, ..., nₙ)**
  - Executing \( n_j \) iterations as one vector instruction of vector length \( n_i \), is semantically equivalent to \( n_i \) scalar iterations
- **private(var₁, var₂, ..., varₙ)**
  - Variables private to each iteration. Initial value is broadcast to all private instances, and the last value is copied out from the last iteration instance.
- **linear(var₁:step₁, var₂:step₂, ..., varₙ:stepₙ)**
  - For every iteration of scalar loop, \( var_i \) is incremented by \( step_i \). Every iteration of the vector loop, therefore increments \( var_i \) by \( VL \times step_i \)
- **reduction(operator:var₁, var₂, ..., varₙ)**
  - Loop code implements reduction (like “+”) on arguments listed which can be vectorized
- **[no]assert**
  - To assert or not to assert when the vectorization fails. Default is to assert for SIMD pragma.
The SIMD annotation introduces ‘foo’ as a function which will be available as a vectorized version. For the calling functions this enables vectorization of the loop and will adapt argument and return value handling correspondingly.
SIMD Function Annotations
(Only a Subset)

__attribute__((simd [(clauses)])) and clause can be

- **processor(cpuid)**
  - Generate a vector version of the function for the given processor. The default processor is taken from the implicit or explicit processor- or architecture- specific flag in the compiler command line.

- **linear(param₁:step₁, param₂:step₂, …, paramₙ:stepₙ)**
  - Consecutive invocations of the function increment paramᵢ by stepᵢ

- **scalar(param₁, param₂, …, paramₙ)**
  - Indicates the values of these parameters can be broadcasted to all iterations as a performance optimization

- **mask**
  - To generate a masked vector version of the function.

- **user**
  - Not to generate this vector version since it is provided by the programmer.
CEAN: C/C++ Extensions for Array Notation

Add array notation to C/C++ similar to what was done for Fortran language in Fortran90 to express data parallel (e.g. SSE-) code explicitly.

Sample:

```
// Traditional:
for (i=0; i < M-K; i++)
{
    s = 0;
    for (j = 0; j < K; j++)
        s += x[i+j] * c[j]
    y[i] = s;
}
```

```
// CEAN Version – outer loop
// “vectorized”

y[0:M-K] = 0;
for (j = 0; j < K; j++)
    y[0:M-K] += x[j:M-K]*c[j];
```

- Nothing totally new; many similar initiatives in the past 30 years
- E.g.: “Vector C” for Control Data 205, +25 years ago
- CEAN requires switch –farray-notation in 12.0 Compiler
CEAN Array Sections

• Array sections
  – Specification per dimension is lower bound : count [: : stride]
  – This is different from Fortran lower bound : upper bound : [ : stride]
  – Samples:
    - A[:] // All of vector A
    - C[:][5] // Column 5 of matrix C
    - D[0:3:2] // Elements 0,2,4 of vector D

• Basic, data parallel operations on array sections
  – Support for most C/C++ arithmetic and logic operators like “+”, “/”, “<”, “&&”, “+=”, ...
  – The shape of the sections must be identical, scalars are expanded implicitly
  – Sample:
    - ( a[0:s]+b[5:s] )* pi // pi * {a[i]+b[i+5], (i=0;i<s;i++)}
CEAN Assignment

Assignment evaluates LHS and assigns values to RHS in parallel

- The shapes of the RHS and LHS array section must be the same
- Scalar is expanded automatically
- Conceptually, RHS is evaluated completely before LHS
  - Compiler ensures semantic in code being generated
  - This can be a critical performance issue (temporary array variable)
- Samples:

```
a[:,] = b[:,][2][:,] + c;
e[:] = d;          // scalar expansion
e[:] = b[:,][1][:]; // error, shapes different
a[:,] = e[:];      // error, shapes different
a[b[0:s]] = c[:];  // scatter operation
c[0:s] = a[b[:]]   // gather operation
```
CEAN – Some Advanced Features

• Functions can take array sections as arguments and can return sections
  – Any assumption on order (side effects) are a mistake
  – Compiler may generate calls to vectorized library functions
  – Examples:
    
    ```c
    a[:] = pow(b[:], c);  // b[:]**c
    a[:] = pow(c, b[:]);  // c**b[:]
    a[:] = foo(b[:])      // user defined
    ```

• Reductions combine elements in an array section into a single value using pre-defined operators or a user function
  – Pre-defined, e.g. `_sec_reduce_{add, mul, min, ...}`
    
    ```c
    sum = __sec_reduce_add(a[:] * b[:]);          // dot product
    res = __sec_reduce(fn, a[:], 0);              // apply function fn
    ```
Agenda

- Vector Instructions (SIMD)
- Compiler Switches for Optimization
- Controlling Auto-vectorization
- **Controlling Auto-parallelization**
- Manual Vectorization (SSE & AVX)
- Hands-on
Auto-parallelization

• **Key requirements**
  – A compiler must not alter the program semantics
  – If the compiler cannot determine all dependencies, it has to forego parallelization

• **Compilers sometimes need to act very conservatively**
  – Pointers make it hard for the compiler to deduce memory layout
  – Codes may produce overlapping arrays through pointer arithmetics
  – If the compiler can’t tell, it does not parallelize

• **Past 30 years have shown that auto-parallelization**
  – is a tough problem in general
  – is only applicable to very regular loops
  – cannot take care of manual parallelization tasks
Compiler Options for Parallelization

• Vectorization is not automatically enabled
• Can be controlled through command line switches:
  -parallel enable parallelization
  -par-threshold\(n\) parallelization threshold
    \(n = 0\) parallelize always
    \(n = 100\) parallelize only if performance gain is 100%
    \(n = 50\) parallelize if probability of performance gain is 50%

• Parallelization reports:
  -par-report\(n\) enable vectorization diagnostics
    0 report no diagnostic information.
    1 report on successfully parallelized loops (default)
    2 report on successfully and unsuccessfully parallelized loops
    3 like 2, but also give information about proven and assumed data dependendies
Compiler Options for Parallelization

• **Controlling scheduling: -par-schedule-** *keyword*
  
  **auto**  
  Lets the compiler or run-time system determine the scheduling algorithm.

  **static**  
  Divides iterations into contiguous pieces.

  **static-balanced**  
  Divides iterations into even-sized chunks.

  **static-steal**  
  Divides iterations into even-sized chunks, but allows threads to steal parts of chunks from neighboring threads.

  **dynamic**  
  Gets a set of iterations dynamically.

  **guided**  
  Specifies a minimum number of iterations.

  **guided-analytical**  
  Divides iterations by using exponential distribution or dynamic distribution.

  **runtime**  
  Defers the scheduling decision until run time.
Agenda

• Vector Instructions (SIMD)
• Compiler Switches for Optimization
• Controlling Auto-vectorization
• Controlling Auto-parallelization
• Manual Vectorization (SSE & AVX)
• Hands-on
void vec_eltwise_product(vec_t* a, vec_t* b, vec_t* c) {
    size_t i;
    for (i = 0; i < a->size; i++) {
        c->data[i] = a->data[i] * b->data[i];
    }
}

Element-wise Vector Multiplication
Element-wise Vector Multiplication (SSE)

```c
void vec_eltwise_product_sse(vec_t* a, vec_t* b, vec_t* c) {
    size_t i;
    __m128 va;
    __m128 vb;
    __m128 vc;
    for (i = 0; i < a->size; i += 4) {
        va = _mm_loadu_ps(&a->data[i]);
        vb = _mm_loadu_ps(&b->data[i]);
        vc = _mm_mul_ps(va, vb);
        _mm_storeu_ps(&c->data[i], vc);
    }
}
```
void vec_eltwise_product_avx(vec_t* a, vec_t* b,
    vec_t* c) {

    size_t i;
    __m256 va;
    __m256 vb;
    __m256 vc;
    for (i = 0; i < a->size; i += 8) {
        va = _mm256_loadu_ps(&a->data[i]);
        vb = _mm256_loadu_ps(&b->data[i]);
        vc = _mm256_mul_ps(va, vb);
        _mm256_storeu_ps(&c->data[i], vc);
    }
}
Zen and the Art of Permutation

- New in-lane PS and PD Permuters
  - Permute controlled via immediate
    \texttt{vPermilPS dest, src, imm}

- New 128-bit permutes
  - Useful for lane-crossing operations
    \texttt{vPerm2F128 dest, src1, src2, imm}

How to compute the control value for the permutation?
Zen and the Art of Permutation

Control value:

8 bits

00 01 10 11 = 27 (0x1B)
Zen and the Art of Permutation

Control value:

8 bits

= 170 (0xAA)
Agenda

• Vector Instructions (SIMD)
• Compiler Switches for Optimization
• Controlling Auto-vectorization
• Controlling Auto-parallelization
• Manual Vectorization (SSE & AVX)
• Hands-on
Lab 1: Vectorization

• Look into the “vectors” directory.
• There you will find a code like this:
  ```c
  void scalar_product(vec_t* a, vec_t* b, vec_t *c) {
    size_t i;
    for (i = 0; i < a->size; i++) {
      c->data[i] = a->data[i] * b->data[i];
    }
  }
  ```

• Try to find out
  – Why the compiler cannot vectorize the code
  – Do you need to change the code?
  – Which compiler options do you need to enable vectorization
Lab 2: Parallelization

• Look into the “vectors” directory.
• There you will find a code like this:

```c
void scalar_product(vec_t* a, vec_t* b, vec_t *c) {
    size_t i;
    for (i = 0; i < a->size; i++) {
        c->data[i] = a->data[i] * b->data[i];
    }
}
```

• Try to find out
  – Why the compiler cannot parallelize the code
  – Do you need to change the code?
  – Which compiler options do you need to enable parallelization
Lab 3: Manual Vectorization

- Look into the “vectors” directory.

- Implement the SSE and AVX functions for computing the scalar product of two vectors.
  - The functions are marked with “TODO”.

- Hints:
  - You will need the following new instructions:
    SSE: \_mm\_hadd\_ps()
    AVX: \_mm256\_maskstore\_ps

- The “Intrinsics Guide for Intel® AVX” is very useful for the hands-on (see http://software.intel.com/en-us/avx/)
Questions?