Performance essentials using OpenMP 4.0 vectorization with C/C++

Authors: Anoop Madhusoodhanan Prabha, Bob Chesebrough

Motivation

Why do developers care about this technology

Why explicit vector programming?

Problem Statement:

- Vector widths are increasing per core and extensions to languages are needed to give best performance on new architectures

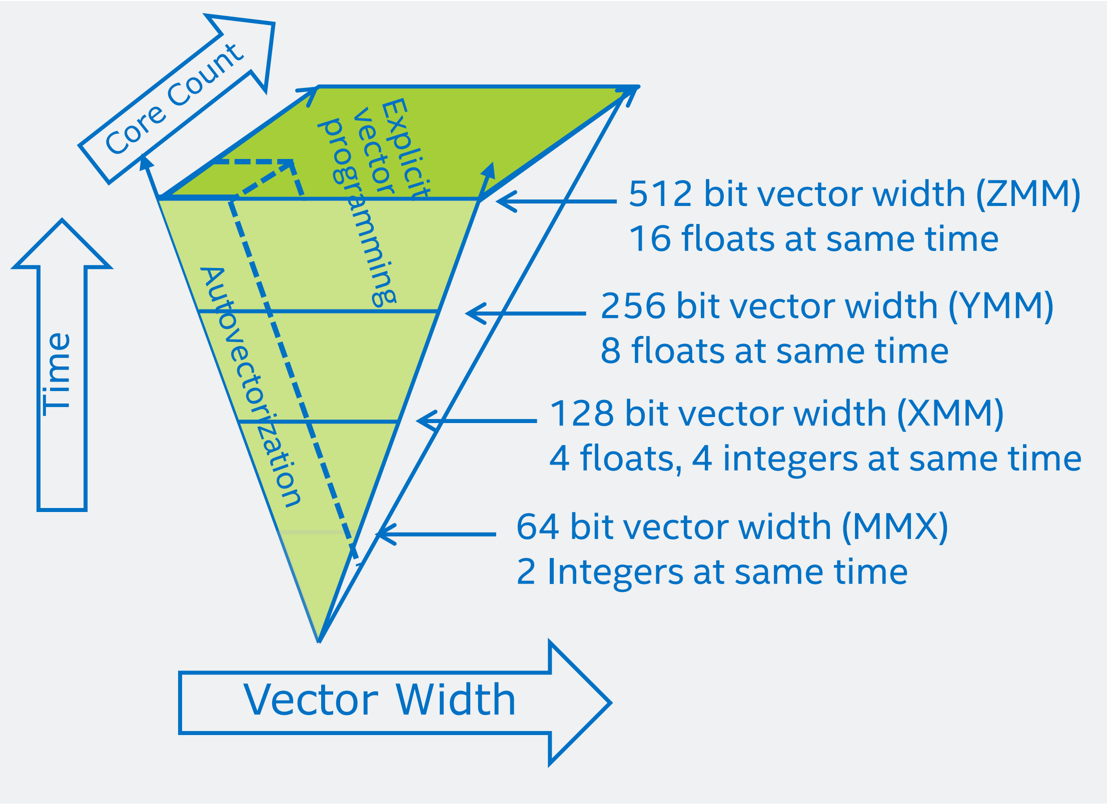
Solution:

- Multiple methods are available to developers to program using explicit vector programming
- We will explore the OpenMP* 4.0 SIMD approach

Goal: Provide language extensions to simplify vector parallelism; Enable developers to extract more performance from SIMD processors

Growth trends for vector registers

Trend: Vector widths and core counts are both increasing. Intel provides developers with explicit methods address these trends



512 bit vector width (ZMM)
16 floats at same time

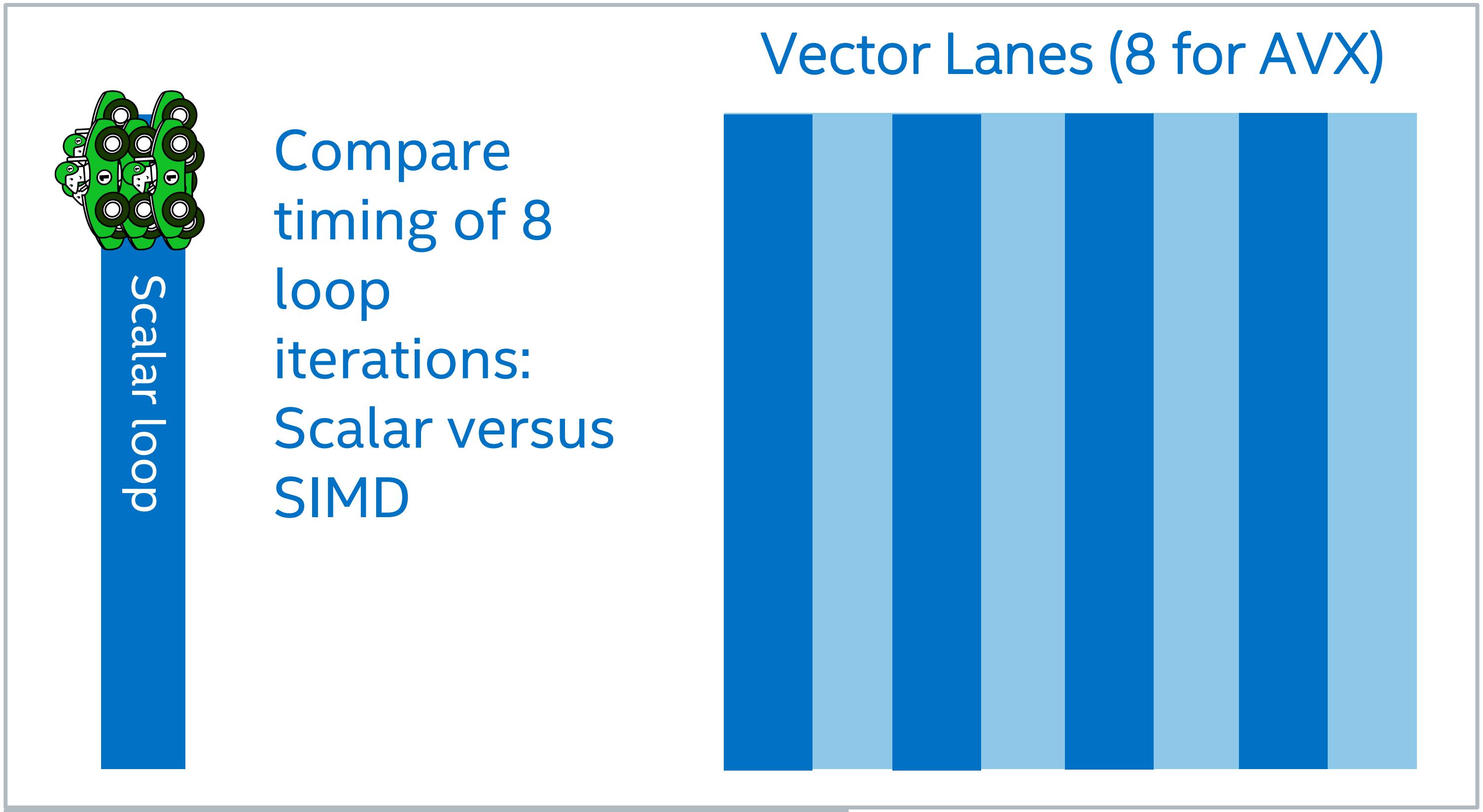
256 bit vector width (YMM)
8 floats at same time

128 bit vector width (XMM)
4 floats, 4 integers at same time

64 bit vector width (MMX)
2 Integers at same time

4/29/2014

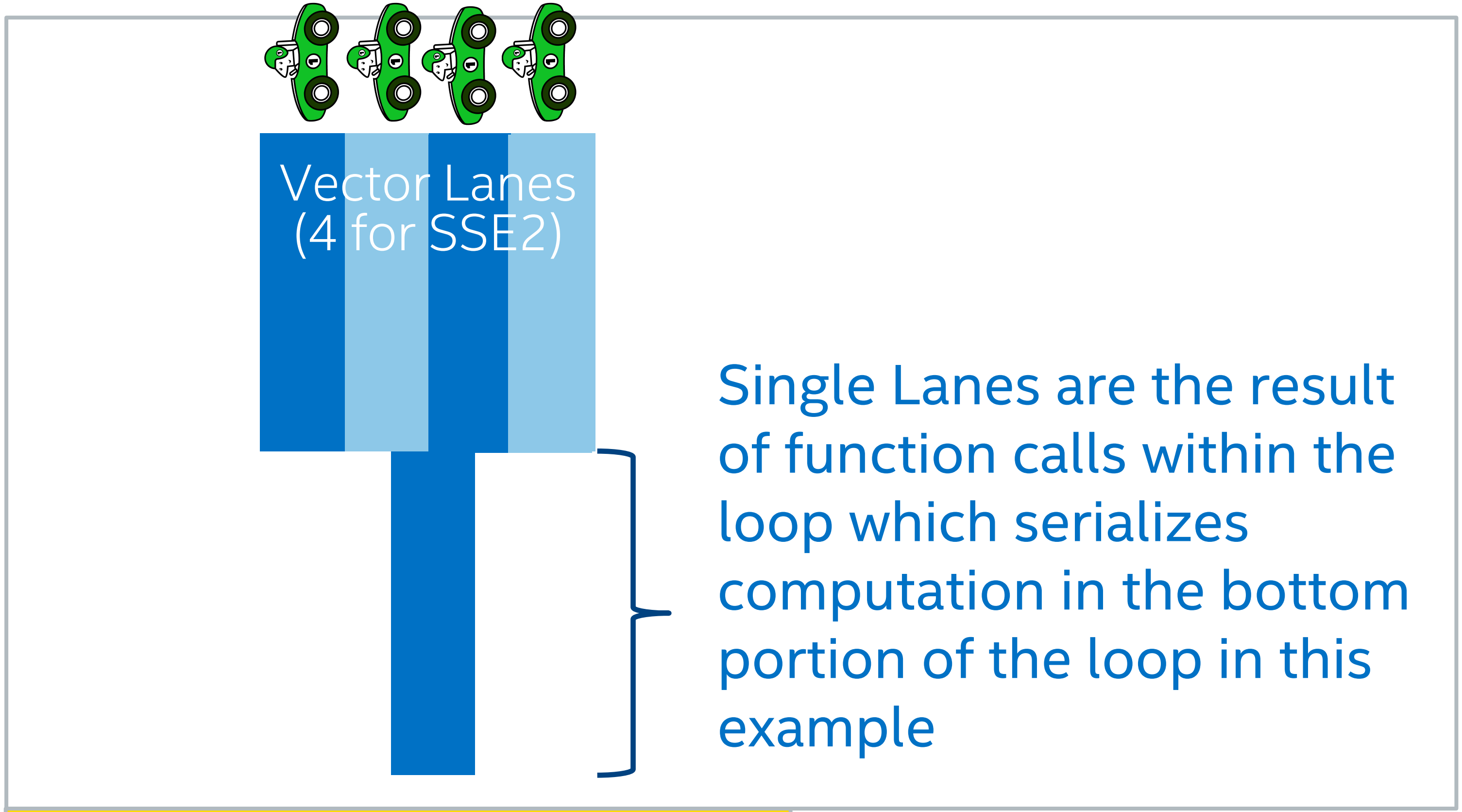
Performance Objective - Maximize Use of SIMD HW per core



Compare timing of 8 loop iterations: Scalar versus SIMD

Use all vector lanes if possible

Performance Objective: Maximize Use of SIMD HW per core



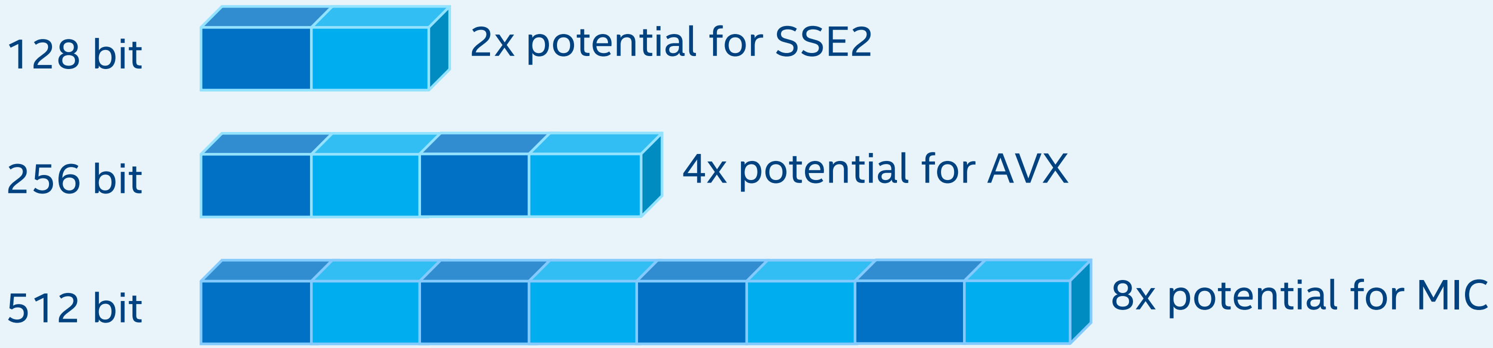
Single Lanes are the result of function calls within the loop which serializes computation in the bottom portion of the loop in this example

Use SIMD-Enabled functions to remove these barriers

4/29/2014

Potential Performance Speedups

Double Precision FP vector width vs speedup potential



128 bit — 2x potential for SSE2

256 bit — 4x potential for AVX

512 bit — 8x potential for MIC

Note:

Wider vectors allow for higher potential performance gains

Gains of 4x and 8x within reach using vectorization capability

SIMD Concepts

Necessary conceptual background

Many Ways to Vectorize

Compiler:
Auto-vectorization (no change of code)

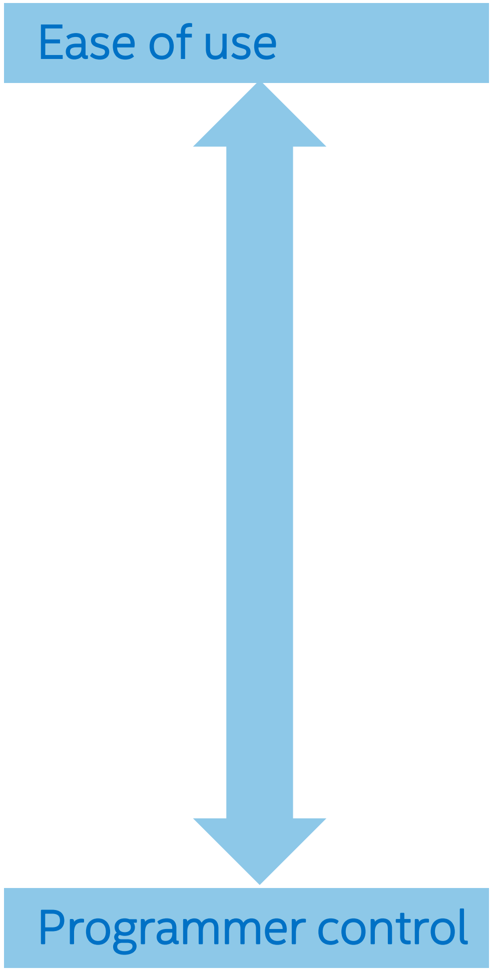
Compiler:
Auto-vectorization hints (#pragma vector, ...)

Explicit Vector Programming

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

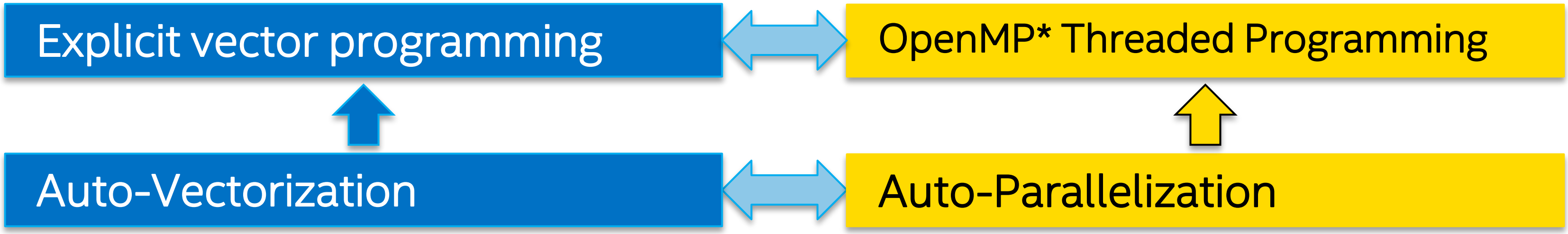
Vector intrinsic
(e.g.: _mm_fmadd_pd(...), _mm_add_ps(...), ...)

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



4/29/2014

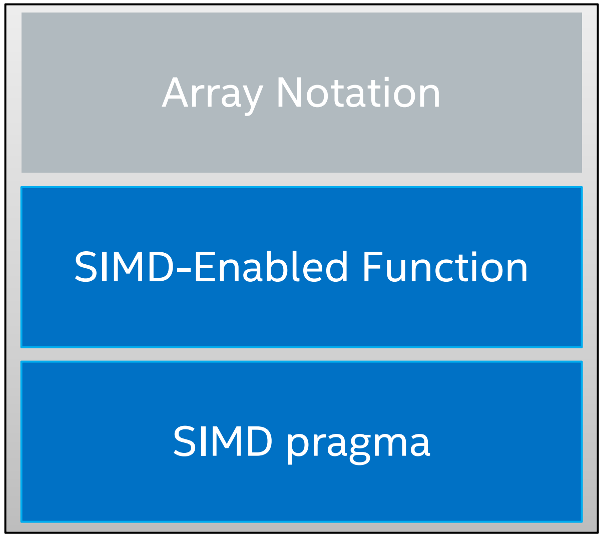
Need Common Programming Models: Explicit Vector Programming


Explicit vector programming

OpenMP* Threaded Programming

Auto-Vectorization

Auto-Parallelization


Array Notation

SIMD-Enabled Function

SIMD pragma

- When auto-vectorization is limited we need to explore explicit vector programming to enable the potential performance in your application

Ways to Write Vector Code

Serial Code

```
for(i = 0; i < N; i++){
  A[i] = B[i] + C[i];
}
```

Array Notation for C/C++

```
A[:] = B[:] + C[:];
```

SIMD Pragma/Directive

```
#pragma omp simd
for(i = 0; i < N; i++) {
  A[i] = B[i] + C[i];
}
```

SIMD-Enabled Function with Intel® Cilk™ Plus Array Notation

```
#pragma omp declare simd
float foo(float B, float C)
{
  return B + C;
}
…

A[:] = foo(B[:], C[:]);
```

Data Level Parallelism with OpenMP* 4.0 Vectorization

OpenMP* 4.0 SIMD-Enabled Functions

Features and use

Overview: SIMD-enabled functions

SIMD-enabled functions allow user defined functions to be vectorized when:

- called from within vectorized loops
- or are called with array notation array arguments.

The vector declaration and associated modifying clauses specify the vector or scalar nature of the function arguments.

It is recommended to add the simd-enabled directive to the function prototype or header file

Implementations exist for :

- Intel® Cilk™ Plus
- OpenMP* 4.0

SIMD-enabled functions

Write a function for one element and add pragma as follows

```
#pragma omp declare simd
float foo(float a, float b, float c, float d)
{
  return a * b + c * d;
}
```

- Call the scalar version:

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

- Call vector version via SIMD loop:

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

- Call it with Intel® Cilk™ Plus array notations:

```
A[:] = foo(B[:], C[:], D[:], E[:]);
```

Concept of SIMD-enabled functions

Allows use of scalar syntax to describe an operation on a single element

The programmer:

- Writes a standard function which operates on scalar values
- Annotates it the function with vector attribute and modifier clauses #pragma omp declare simd
  - Utilize appropriate modifier clause for vector attribute
- Invokes the function to operate on arrays of arguments rather than scalar arguments

The compiler:

- Generates a scalar and a short vector version(s).
- Can call the vector function from vectorized loop
- Can call the scalar function from a scalar loop (legacy code)

SIMD-enabled functions: Linear/ Uniform

Why do we need them?

Because unless uniform or linear are specified each parameter to the function will be treated as a vector

```
#pragma omp declare simd uniform(a) linear(i:1)
void foo(float *a, int i):
   a is a pointer
   i is a sequence of integers [i, i+1, i+2, …]
   a[i] is a unit-stride load/store ([v]movups)
```

```
#pragma omp declare simd
void foo(float *a, int i):
  a is a vector of pointers
  i is a vector of integers
  a[i] becomes gather/scatter.
```

- Reference: http://software.intel.com/en-us/articles/usage-of-linear-and-uniform-clause-in-elemental-function-simd-enabled-function-clause

SIMD-enabled functions: Invocation

```
#pragma omp declare simd
float my_simdf (float b)    { …  }
```

Construct	Example	Semantics

Standard for loop	`for (j = 0; j < N; j++) {`
	`   a[j] = my_simdf(b[j]);`
	`}`
		Single thread, potentially auto-vectorizable

#pragma omp simd	`#pragma omp simd`
	`for (j = 0; j < N; j++) {`
	`   a[j] = my_simdf(b[j]);`
	`}`
		Single thread, vectorized; use the appropriate vector version

Intel® Cilk™ Plus Array notation	`a[:] = my_simdf(b[:]);`	Single thread, vectorized; use the appropriate vector version

Call site dependence

- Callee Site

```
#pragma omp declare simd uniform(a),linear(i:1),simdlen(4)
void foo(int *a, int i){
     std::cout<<a[i]<<"\n";
}
```

- Call site

```
#pragma omp simd safelen(4)
for(int i = 0; i < n; i++)
         foo(a, i);
```

- Vectorization report

```
testmain.cc(5): (col. 13) remark: OpenMP SIMD LOOP WAS VECTORIZED
header.cc(3): (col. 24) remark: FUNCTION WAS VECTORIZED
header.cc(3): (col. 24) remark: FUNCTION WAS VECTORIZED
header.cc(3): (col. 24) remark: FUNCTION WAS VECTORIZED
header.cc(3): (col. 24) remark: FUNCTION WAS VECTORIZED
```

- Reference: http://software.intel.com/en-us/articles/call-site-dependence-for-elemental-functions-simd-enabled-functions-in-c

Call site dependence (cont)

- Callee Site

```
#pragma omp declare simd uniform(a),linear(i:1),simdlen(4)
void foo(int *a, int i){
    std::cout<<a[i]<<"\n";
}
```

- Call site

```
#pragma omp simd safelen(4)
for(int i = 0; i < n; i++)  foo(a, i);
#pragma omp simd safelen(4)
for(int i = 0; i < n; i++){
        k = b[i];  // k is not linear
        foo(a, k);
}
```

- Vectorization report

```
testmain.cc(14): (col. 13) remark: OpenMP SIMD LOOP WAS VECTORIZED
testmain.cc(21): (col. 9) remark: No suitable vector variant of function
'_Z3fooPii' found
testmain.cc(18): (col. 1) remark: OpenMP SIMD LOOP WAS VECTORIZED
header.cc(3): (col. 24) remark: FUNCTION WAS VECTORIZED
```

SIMD-enabled function
Multiple vector definitions allowed

- Callee Site

```
#pragma omp declare simd uniform(a),linear(i:1),simdlen(4)
#pragma omp declare simd uniform(a),simdlen(4)
void foo(int *a, int i){
     std::cout<<a[i]<<"\n";
}
```

- Call site

```
#pragma omp simd safelen(4)
for(int i = 0; i < n; i++) foo(a, i);
#pragma omp simd safelen(4)
for(int i = 0; i < n; i++){
         k = b[i]; // k is not linear
         foo(a, k);
}
```

- Vectorization report

```
testmain.cc(14): (col. 13) remark: OpenMP SIMD LOOP WAS VECTORIZED
testmain.cc(18): (col. 1) remark: OpenMP SIMD LOOP WAS VECTORIZED
header.cc(3): (col. 24) remark: FUNCTION WAS VECTORIZED
```

Reference: http://software.intel.com/en-us/articles/call-site-dependence-for-elemental-functions-simd-enabled-functions-in-c

OpenMP* 4.0 SIMD Loops

Features and use

Pragma omp SIMD Motivation

The following example will likely fail to auto vectorize

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

Without SIMD directive, vectorization will fail since there are too many pointer references to do a run-time check for overlapping arrays

Auto-Vectorization – Serial Constraints

Compiler checks for:

```
for(i = 0; i < *p; i++) {
  A[i] = B[i] * C[i];
  sum = sum + A[i];
}
```

- Is *p loop invariant?
- Do A[], B[], C[] overlap?
- Is sum aliased with B[] and/or C[]?
- Does the order of math operations matter?
- Vector computation expected to be faster than scalar code? (efficiency heuristic)

Auto vectorization is limited by the language rules: you can't say what you want!

Explicit Vector Programming with SIMD Pragma/Directive

Programmer asserts:

```
#pragma omp simd reduction(+:sum)
for(i = 0; i < *p; i++) {
  A[i] = B[i] * C[i];
  sum = sum + A[i];
}
```

- *p is loop invariant
- sum not aliased with B[] or C[]
- A[] does not overlap with B[] or C[]
- sum should be treated as a reduction
- Allow compiler to reorder for better vectorization
- Vector code should be generated even if efficiency heuristic does not indicate a gain in performance

Explicit vector programming lets you express what you mean!

Data in Vector Loops

The two statements with the += operations have different meaning from each other

The programmer should be able to express those differently

The compiler has to generate different code

The variables i, p and step have different “meaning” from each

```
float sum = 0.0f;
float *p = a;
int step = 4;

#pragma omp simd
for (int i = 0; i < N; ++i) {
        sum += *p;
        p += step;
}
```

Data in Vector Loops

Linear and reduction clauses make this usage explicit.

```
float sum = 0.0f;
float *p = a;
int step = 4;

#pragma omp simd linear(p:step)reduction(+:sum)
for (int i = 0; i < N; ++i) {
        sum += *p;
        p += step;
}
```

SIMD Pragma Notation

OpenMP 4.0: #pragma omp simd [clause [,clause] …]

Targets loops

- Can target inner or outer loops

Developer responsible for results

- Developer asserts loop is suitable for SIMD
  - no loop-carried dependencies and iterations can be evaluated in parallel
- Can choose from lexicon of clauses to modify behavior of SIMD directive
- Developer should validate results

Data in Vector Loops

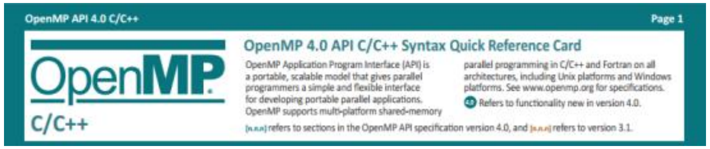
```
extern float *a;
float sum = 0.0f;
float *p = a;
int step = 4;
int i,j;
#pragma omp simd collapse(2) reduction(+:sum)
linear(p:step) aligned(p:16) safelen(4)
for(i = 0; i < N; i+=8) {
        for(j = i; j < i+8; j++) {
                sum += *p;
                 p += step;
        }
}
```

Increase Performance with Explicit Vector Programming

OpenMP* 4.0 SIMD extensions is supported by:

- Intel® Cluster Studio XE
  - MPI hybrid cluster development tools
- Intel® Parallel Studio XE Suites
  - C, C++ and Fortran compilers, libraries and analysis tools
- Intel® Composer XE Suites
  - Compilers and performance libraries

Try it for free!
intel.ly/perf-tools



OpenMP API 4.0 C/C++ Page 1

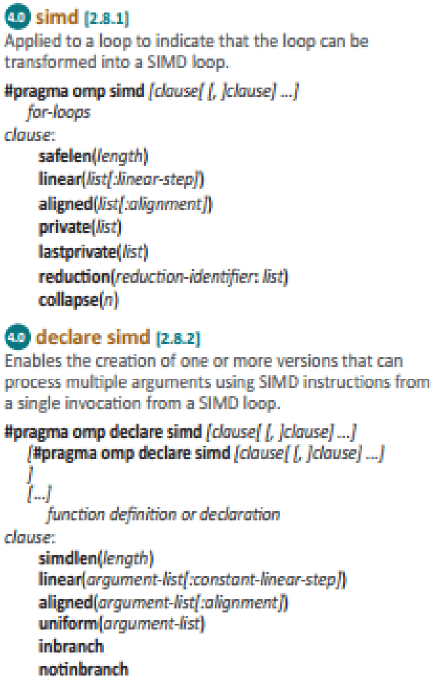
OpenMP C/C++

OpenMP 4.0 API C/C++ Syntax Quick Reference Card

OpenMP Application Program Interface (API) is a portable, scalable model that gives parallel programmers a simple and flexible interface for developing portable parallel applications. OpenMP supports multi-platform shared-memory parallel programming in C/C++ and Fortran on all architectures, including Unix platforms and Windows platforms. See www.openmp.org for specifications.

4.0 Refers to functionality new in version 4.0.

[n.n.n] refers to sections in the OpenMP API specification version 4.0, and [n.n.n] refers to version 3.1.



4.0 **simd** [2.8.1]

Applied to a loop to indicate that the loop can be transformed into a SIMD loop.

#pragma omp simd *[clause[[,]clause] ...]*
  for-loops

clause:
  safelen(*length*)
  linear(*list[:linear-step]*)
  aligned(*list[:alignment]*)
  private(*list*)
  lastprivate(*list*)
  reduction(*reduction-identifier: list*)
  collapse(*n*)

4.0 **declare simd** [2.8.2]

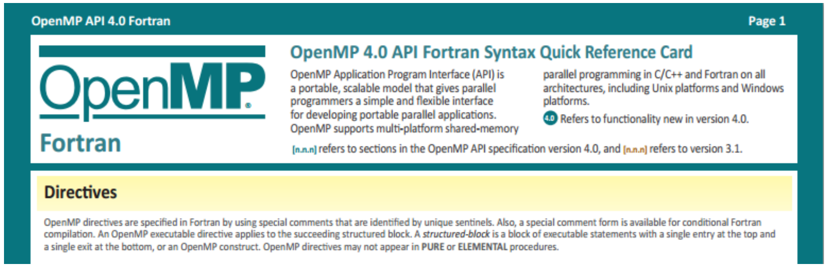
Enables the creation of one or more versions that can process multiple arguments using SIMD instructions from a single invocation from a SIMD loop.

#pragma omp declare simd *[clause[[,]clause] ...]*
  *[***#pragma omp declare simd** *[clause[[,]clause] ...]*
  ]
  [...]
   function definition or declaration

clause:
  simdlen(*length*)
  linear(*argument-list[:constant-linear-step]*)
  aligned(*argument-list[:alignment]*)
  uniform(*argument-list*)
  inbranch
  notinbranch

References

- http://openmp.org/
- Performance Essentials with OpenMP 4.0 Vectorization: https://software.intel.com/articles/performance-essentials-with-openmp-40-vectorization
- Explicit Vector Programming –Best Known Methods –Article https://software.intel.com/en-us/articles/explicit-vector-programming-best-known-methods
- OpenMP 4.0 Summary Card - C/C++ (October 2013 PDF)
- OpenMP 4.0 Summary Card - Fortran (October 2013 PDF)
- OpenMP 4.0.1 Examples (February 2014 PDF)
- Enabling SIMD in program using OpenMP4.0



OpenMP API 4.0 Fortran — Page 1

OpenMP Fortran

OpenMP 4.0 API Fortran Syntax Quick Reference Card

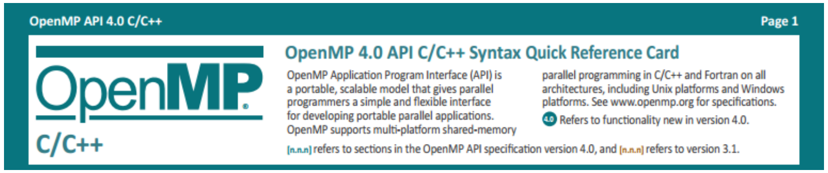
OpenMP Application Program Interface (API) is a portable, scalable model that gives parallel programmers a simple and flexible interface for developing portable parallel applications. OpenMP supports multi-platform shared-memory parallel programming in C/C++ and Fortran on all architectures, including Unix platforms and Windows platforms.

4.0 Refers to functionality new in version 4.0.

[n.n.n] refers to sections in the OpenMP API specification version 4.0, and [n.n.n] refers to version 3.1.

Directives

OpenMP directives are specified in Fortran by using special comments that are identified by unique sentinels. Also, a special comment form is available for conditional Fortran compilation. An OpenMP executable directive applies to the succeeding structured block. A *structured-block* is a block of executable statements with a single entry at the top and a single exit at the bottom, or an OpenMP construct. OpenMP directives may not appear in **PURE** or **ELEMENTAL** procedures.



OpenMP API 4.0 C/C++ — Page 1

OpenMP C/C++

OpenMP 4.0 API C/C++ Syntax Quick Reference Card

OpenMP Application Program Interface (API) is a portable, scalable model that gives parallel programmers a simple and flexible interface for developing portable parallel applications. OpenMP supports multi-platform shared-memory parallel programming in C/C++ and Fortran on all architectures, including Unix platforms and Windows platforms. See www.openmp.org for specifications.

4.0 Refers to functionality new in version 4.0.

[n.n.n] refers to sections in the OpenMP API specification version 4.0, and [n.n.n] refers to version 3.1.

Q & A

Legal Disclaimer & Optimization Notice

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 © 2014, 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.

Optimization Notice

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.

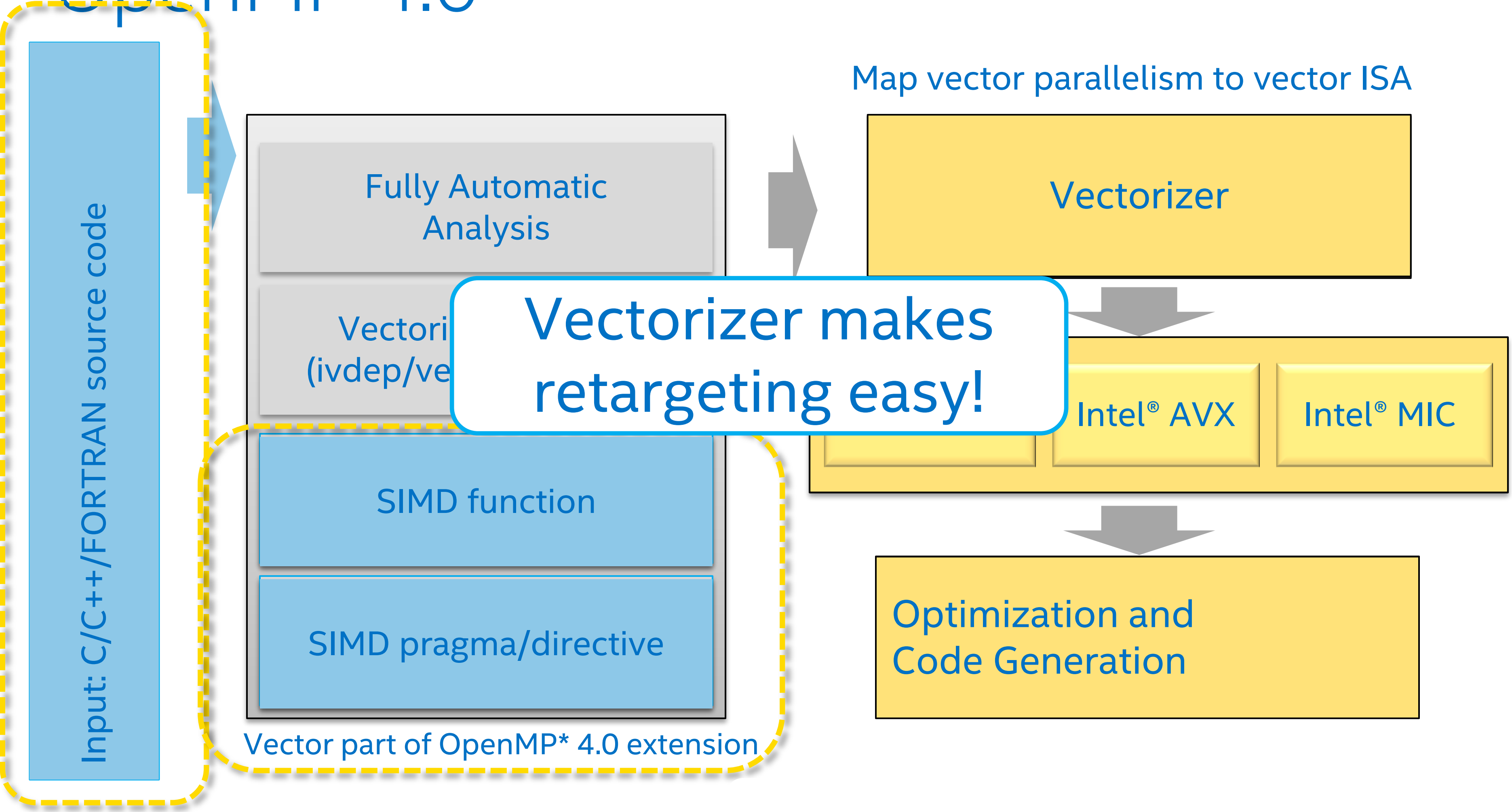
Notice revision #20110804

intel®

Abstract

Performance essentials using OpenMP* 4.0 vectorization with C/C++ This webinar teaches you about Vectorization, what it is and why you should care about it as a software developer. It will cover terms such as SIMD and vectorization, Vector Lanes, Vector Length and discusses performance expectations per core. It will also explores the tradeoff between using compiler autovectorization versus explicit vector programming versus SIMD intrinsics and assembly. It compares explicit vector programming as being similar to explicit parallel programming using OpenMP parallelism constructs, where the developer takes control and responsibility for vectorizing specified loops. also gives quick examples of the two big ideas in explicit vector programming: omp SIMD loops, and SIMD-enabled functions enabled with the pragma omp declare simd family of constructs.

Explicit Vector Programming with OpenMP 4.0


Input: C/C++/FORTRAN source code
Fully Automatic Analysis
Vectori
(ivdep/ve
SIMD function
SIMD pragma/directive
Vector part of OpenMP* 4.0 extension
Map vector parallelism to vector ISA
Vectorizer
Intel® AVX
Intel® MIC
Optimization and Code Generation

Vectorizer makes retargeting easy!

Express/expose vector parallelism **/Openmp [/Qx[SSE2|AVX]]**

#pragma omp declare simd -modifiers

Optional modifier clauses:

- uniform(param1[, param2]…):
  Shared, scalar parameters are broadcasted to all iterations
- linear(param1:step1[, param2:step2]…):
  In serial execution parameters are incremented by steps, examples are induction variables with constant stride
- simdlen(num): the largest size for a vector that the compiler is free to assume, usually 2,4,8,16
- aligned(argument-list[:alignment]): all arguments in the argument-list are aligned on a known boundary not less than the specified alignment.

Refer to OpenMP 4.0 Specification.

http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf

Restrictions: SIMD-enabled functions

Each argument can appear in at most one uniform or linear clause.

In a linear clause the step value must be a constant positive integer expression.

The function or subroutine body must be a structured block.

No OpenMP constructs allowed inside the declared function.

The execution of the function cannot have any side effects regarding concurrent iterations of a SIMD chunk.

branching into or out of the function is not allowed.

C/C++: No calls to the longjmp or setjmp

OMP SIMD Pragma Clauses

reduction(operator:v1, v2, …)

- v1 etc are reduction variables for operation “operator”
- Examples include computing averages or sums of arrays into a single scalar value : reduction (+:sum)

linear(v1:step1, v2:step2, …)

- declares one or more list items to be private to a SIMD lane and to have a linear relationship with respect to the iteration space of a loop : linear (i:2)

safelen (length)

- no two iterations executed concurrently with SIMD instructions can have a greater distance in the logical iteration space than this value
- Typical values are 2, 4, 8, 16

Refer to OpenMP 4.0 Specification.

http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf

OMP SIMD Pragma Clauses cont…

aligned(v1:alignment, v2:alignment)

- declares that the object to which each list item points is aligned to the number of bytes expressed in the optional parameter of the aligned clause.

collapse(number of loops)

- Nested loop iterations are collapsed into one loop with a larger iteration space.

private(v1, v2, …), lastprivate (v1, v2, …)

- declares one or more list items to be private to an implicit task or to a SIMD lane, lastprivate causes the corresponding original list item to be updated after the end of the region..

Refer to OpenMP 4.0 Specification.

http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf

OpenMP 4.0 SIMD Pragma

Restrictions applying pragma omp simd (partial list):

- Applied to for loops only
- Induction variables should be signed or unsigned int
- The associated loops must be structured blocks
- A program must not branch into or out of a SIMD region.
- No OpenMP* construct can appear inside a simd region
- No C++ exceptions and Windows* Structured Exception Handling, setjmp(…) & longjmp(…) in loop body

Refer to OpenMP 4.0 Specification.

http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf