General Induction Support

A Proposed Extension for OpenMP*

Ernesto Su, Hideki Saito, Xinmin Tian
Intel Corporation

OpenMPCon 2017
September 18, 2017

Legal Notice and Disclaimers

By using this document, in addition to any agreements you have with Intel, you accept the terms set forth below.

You may not use or facilitate the use of this document in connection with any infringement or other legal analysis concerning Intel products described herein. You agree to grant Intel a non-exclusive, royalty-free license to any patent claim thereafter drafted which includes subject matter disclosed herein.

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.

A "Mission Critical Application" is any application in which failure of the Intel Product could result, directly or indirectly, in personal injury or death. SHOULD YOU PURCHASE OR USE INTEL'S PRODUCTS FOR ANY SUCH MISSION CRITICAL APPLICATION, YOU SHALL INDEMNIFY AND HOLD INTEL AND ITS SUBSIDIARIES, SUBCONTRACTORS AND AFFILIATES, AND THE DIRECTORS, OFFICERS, AND EMPLOYEES OF EACH, HARMLESS AGAINST ALL CLAIMS COSTS, DAMAGES, AND EXPENSES AND REASONABLE ATTORNEYS' FEES ARISING OUT OF, DIRECTLY OR INDIRECTLY, ANY CLAIM OF PRODUCT LIABILITY, PERSONAL INJURY, OR DEATH ARISING IN ANY WAY OUT OF SUCH MISSION CRITICAL APPLICATION, WHETHER OR NOT INTEL OR ITS SUBCONTRACTOR WAS NEGLIGENT IN THE DESIGN, MANUFACTURE, OR WARNING OF THE INTEL PRODUCT OR ANY OF ITS PARTS.

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

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.

For more complete information about performance and benchmark results, visit Performance Test Disclosure

Intel does not control or audit the design or implementation of third party benchmark data or Web sites referenced in this document. Intel encourages all of its customers to visit the referenced Web sites or others where similar performance benchmark data are reported and confirm whether the referenced benchmark data are accurate and reflect performance of systems available for purchase.

Intel processor numbers are not a measure of performance.

Processor numbers differentiate features within each processor family, not across different processor families: Go to: Learn About Intel® Processor Numbers

Intel® Advanced Vector Extensions (Intel® AVX)* are designed to achieve higher throughput to certain integer and floating point operations. Due to varying processor power characteristics, utilizing AVX instructions may cause a) some parts to operate at less than the rated frequency and b) some parts with Intel® Turbo Boost Technology 2.0 to not achieve any or maximum turbo frequencies. Performance varies depending on hardware, software, and system configuration and you should consult your system manufacturer for more information.

*Intel® Advanced Vector Extensions refers to Intel® AVX, Intel® AVX2 or Intel® AVX-512. For more information on Intel® Turbo Boost Technology 2.0, visit http://www.intel.com/go/turbo

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

Motivation

Why can't we do this seemingly simple thing? Financial code from customers has this pattern, and we can't vectorize it with **`simd`**

```
X x = …
#pragma omp simd linear(x:1)
for(int i=0; i<N; i++){
    work(x);
    x ++; // ++ is defined for class X
}
```

← Error! x is not of integral type

Inductions Beyond the `linear` Clause

`linear` : limited to the + operator on integral types

We want to extend OpenMP to support

- More operators
  - Other built-in operators: -, ×, and ÷
  - User-defined operators
- More data types
  - Other built-in data types: float, double, etc.
  - User-defined types, including non-POD types

Agenda

- Induction overview
- The **induction** clause
  - Syntax
  - Example: polynomial evaluation
- User-defined induction (UDI): the **declare induction** construct
  - Syntax
  - Parallelization example
  - Vectorization example
- Conclusions

Induction Overview

- **Recursive form:** $x_i = x_{i-1} \oplus s$
  - $\oplus$ the inductive operator (or “inductor”)
  - i index (zero-normalized)
  - s step expression
  - x_0 initial value
- **Expansion:** $x_i = x_0 \oplus s \oplus s \oplus \ldots \oplus s$ (i times)
- **Collection:** often the i s-terms can be “collected” into $s_i = s \otimes i$
  - $\otimes$ the collective operator (or “collector”)
  - s_i collective step expression
- **Closed form:** when collection is possible, we can express the induction with the closed form $x_i = x_0 \oplus (s \otimes i)$

Induction of Basic Arithmetic Operations

$\oplus$	Recursive form	$s \otimes i$	Closed form
+	$x_i = x_{i-1} + s$	$s * i$	$x_i = x_0 + s * i$
-	$x_i = x_{i-1} - s$	$s * i$	$x_i = x_0 - s * i$
*	$x_i = x_{i-1} * s$	s^i	$x_i = x_0 * s^i$
/	$x_i = x_{i-1} / s$	s^i	$x_i = x_0 / s^i$

Proposal: **induction** Clause

induction(*induction-id* **:** *var-list* **:** *step* **)**

- To be used with the OpenMP loop, **distribute**, and **simd** constructs
- *induction-id* : can be a built-in op (+, -, *, /) or user-defined
- *var-list* : one or more variables. Intergral or FP for built-in operators. Can be of non-POD types for user-defined operators.
- *step* : the step expression
- Examples
  - **induction**(+ : x, y : 1) is equivalent to **linear**(x, y) if x, y are integral
  - **induction**(* : x : s) describes the nonlinear induction $x_i = x_{i-1} * s$
  - **induction**(foo : x : s) uses a user-defined induction operator “foo”; x and s can be of different non-POD types

Example: Evaluating a Polynomial

Compute the value of $\sum_{i=0}^{N} c_i x^i$

```
float c[N];          // the coefficients
float s = x;         // step is the value of x for which
                     //     to evaluate the polynomial
float xi = 1.0;      // x^i; initial value x^0 == 1
float value = 0.0; // accumulator for the result

#pragma omp simd reduction(+:value) induction(* : xi : s)
for(i=0; i<=N; i++){
    value += c[i] * xi;
    xi *= s;
}
```

Going Beyond Built-in Types and Operators

The user needs to specify the following

- Type of induction variable
- Type of the step expression
- Inductive operator
- Collective operator (optional)
  - Allows computation, in constant time, of the initial value of the induction variable for each thread or SIMD lane by using the closed-form
  - Omitting the collector may negatively affect performance
- The proposed syntax has a form similar to UDR's **declare reduction** construct

User-Defined Reduction (UDR) Recap

#pragma omp declare reduction (*reduction-id* **:** *type-list* **:** *combiner* **)**
[**initializer(** *init-expr* **)** *]*

- *reduction-id* : name of this reduction operation
- *type-list* : list of type specifiers for the reduction variables
- *combiner* : expression to combine partial results of the given type(s)
  - Two special variables: **omp_in** and **omp_out**
  - Example1: **omp_out** = **omp_out** + **omp_in**
  - Example2: foo(&**omp_out**, **omp_in**)
- *init-expr*:
  - Two special variables: **omp_priv** and **omp_orig**
  - Example: **omp_priv** = `{ MAX_INT, MAX_INT }`

Proposal: User-Defined Induction (UDI)

#pragma omp declare induction (*induction-id* **:** *induction-type* **:** *step-type* **:** *inductor* **)** *[***collector(** *collector* **)***]*

- *induction-id* : identifier for the operation, to be used in an **induction** clause
- *induction-type* : type specifier for the induction variables
- *step-type* : type specifier for the step expression
- *inductor* ⊕ : specifies the inductive operation: $x = x \oplus s$
  - **omp_out** represents *x*
  - **omp_step** represents *s*
  - C++ Example: **omp_out** = **omp_out** + **omp_step**, where + is overloaded
  - C Example: add(&**omp_out**, **omp_step**)

Proposal: User-Defined Induction (UDI) (cont)

- *collector* $\otimes$:
  - **omp_step** represents *s*
  - **omp_index** represents the logical, zero-normalized index *i*
  - C++ Example: **omp_step** = **omp_step** * **omp_index**, where * may be overloaded
  - C Example: cs(&**omp_step**, **omp_index**)
- If *collector* is available, then the closed form is $x_i = x_0 \oplus (s \otimes i)$

Example1 (Parallelization): Source

```
class A;                                  // class of induction variables
class S;                                  // class of the step expression

#pragma omp declare induction ( op : A : S :                          \
                                  omp_out = omp_out + omp_step )      \
                      collector ( omp_step = omp_step * omp_index )
     // A←A+S and S←S*int must be defined
...
A a; S s;  // initialized by constructors
...

#pragma omp parallel for induction( op : a : s )
for(int i=0; i<N; i++) { work(a); a=a+s; }
```

Example1: Compiler Front-End Output

```
void inductor(A *omp_out_ptr, S *omp_step_ptr) {
  A_add_S(omp_out_ptr, omp_step_ptr);
}

void collector(S *omp_step_ptr, int omp_index) {
  S_mult_int(omp_step_ptr, omp_index);
}
```

Example1: Threaded Pseudocode

```
// ...Call runtime to obtain lb,ub for each thread...
firstprivatize(a,s); // privatized and copy-constructed in each thr
collector(&s, lb);   // Use the closed form to find a's initial
inductor(&a, &s)     //    value for each thread in const time

for(int i=lb; i<ub; i++) {
   work(a);
   a = a + s;
}
lastprivatize(a);    // lastprivate copy-out for a
```

Example2 (Vectorization): Source

```
class A;                                  // class of induction variables
class S;                                  // class of the step expression

#pragma omp declare induction ( op : A : S :                          \
                                  omp_out = omp_out + omp_step )      \
                      collector ( omp_step = omp_step * omp_index )
     // A←A+S and S←S*int must be defined
...
A a; S s;  // initialized by constructors
...

#pragma omp simd induction( op : a : s )
for(int i=0; i<N; i++) { work(a); a=a+s; }
```

Example2: Vectorized Pseudocode

```
A a; S s;                              // original scalar non-POD variables
A aa[VL] = bcast(a);                   // initialize vector for a
S ss[VL] = bcast(s);                   // initialize vector for s

int idx[VL] = {VL-1,VL-2,…,1,0};
simd_collector(ss, idx);               // Get vector step ss[:]=idx[:]*ss[:]
simd_inductor(aa, ss);                 // Get initial vector aa[:]=aa[:]+ss[:]

S t = s;
collector(&t, VL);                     // Use collector to create a vector step
ss[:]=bcast(t);                        //   ss[:] = bcast( s * VL )

for(int i=0; i<N; i+=VL) {
  simd_work(aa[:]);
  aa[:] = aa[:] + ss[:];
}

// copy out last aa to a
```

Compiler Implementation Status

- Intel C/C++ compiler
  - Front-end and IR: done
  - Parallelizer and Vectorizer back-end: In progress
- Intel Fortran compiler
  - Front-end: TBD
  - Back-end (shared with C/C++)
- Runtime
  - No changes expected

Conclusions

- Real applications demand OpenMP support for induction beyond that provided by the **linear** clause
- The proposed **declare induction** construct and **induction** clause will support
  - More induction operations, including user-defined operations
  - More data types, including user-defined class types
- Examples were presented


intel®
experience
what's inside™