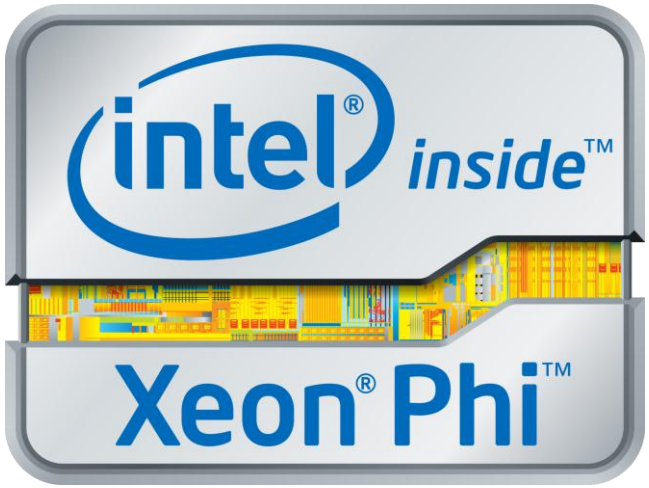
Programming for the Intel® Xeon Phi™ Coprocessor



Dr.-Ing. Michael Klemm
Software and Services Group
Intel Corporation
(michael.klemm@intel.com)

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.

Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, reference www.intel.com/software/products.

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

References

- Knights Corner Quick Start Developers Guide
  https://mic-dev.intel.com/system/files/673_KNC_SDP_Quick_Start_Developers_Guide_Alpha2.pdf

- Knights Corner Instruction Set Reference Manual
  http://software.intel.com/en-us/forums/showthread.php?t=105443&o=a&s=lr

- Knights Corner Software Developers Guide
  https://mic-dev.intel.com/system/files/KNC%20SDG%201.03_0.pdf

- General information at mic-dev
  https://mic-dev.intel.com/

Agenda

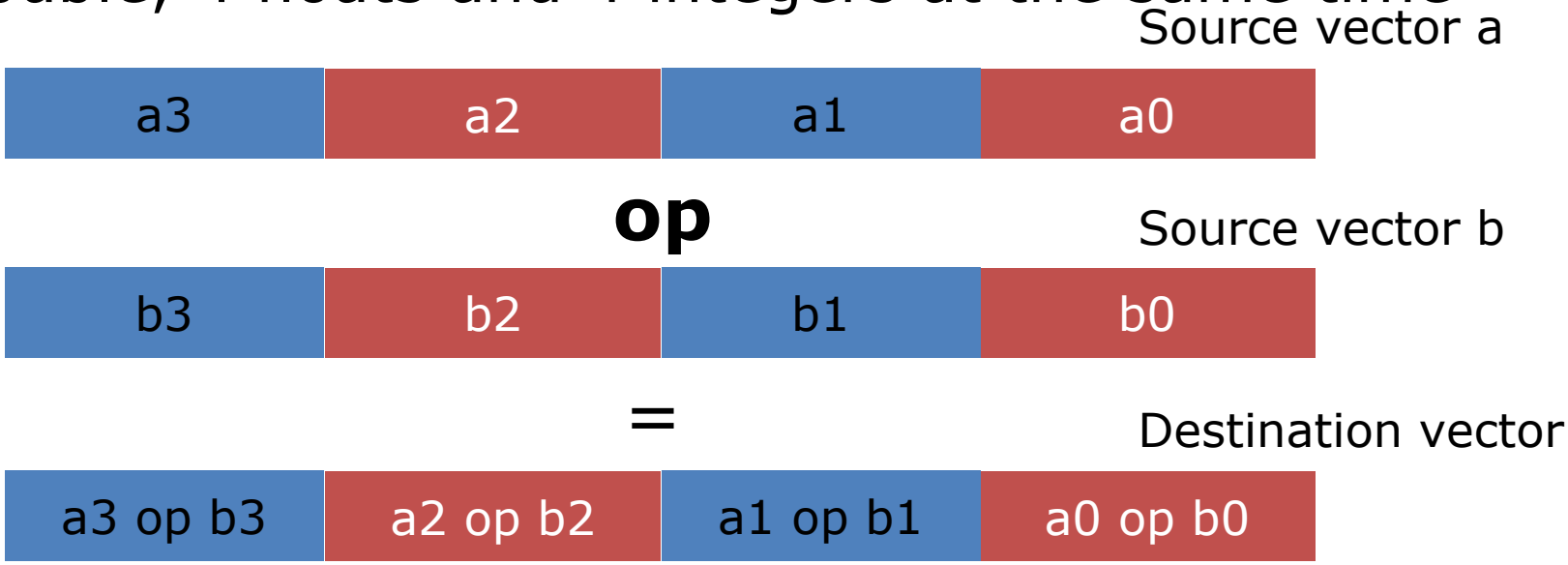
- Introduction to the Intel MIC Architecture
- Programming Models
  - Native Programming
  - Intel MKL
  - Offloading w/ explicit data transfers
  - Offloading w/ implicit data transfers
  - Vectorization

Agenda

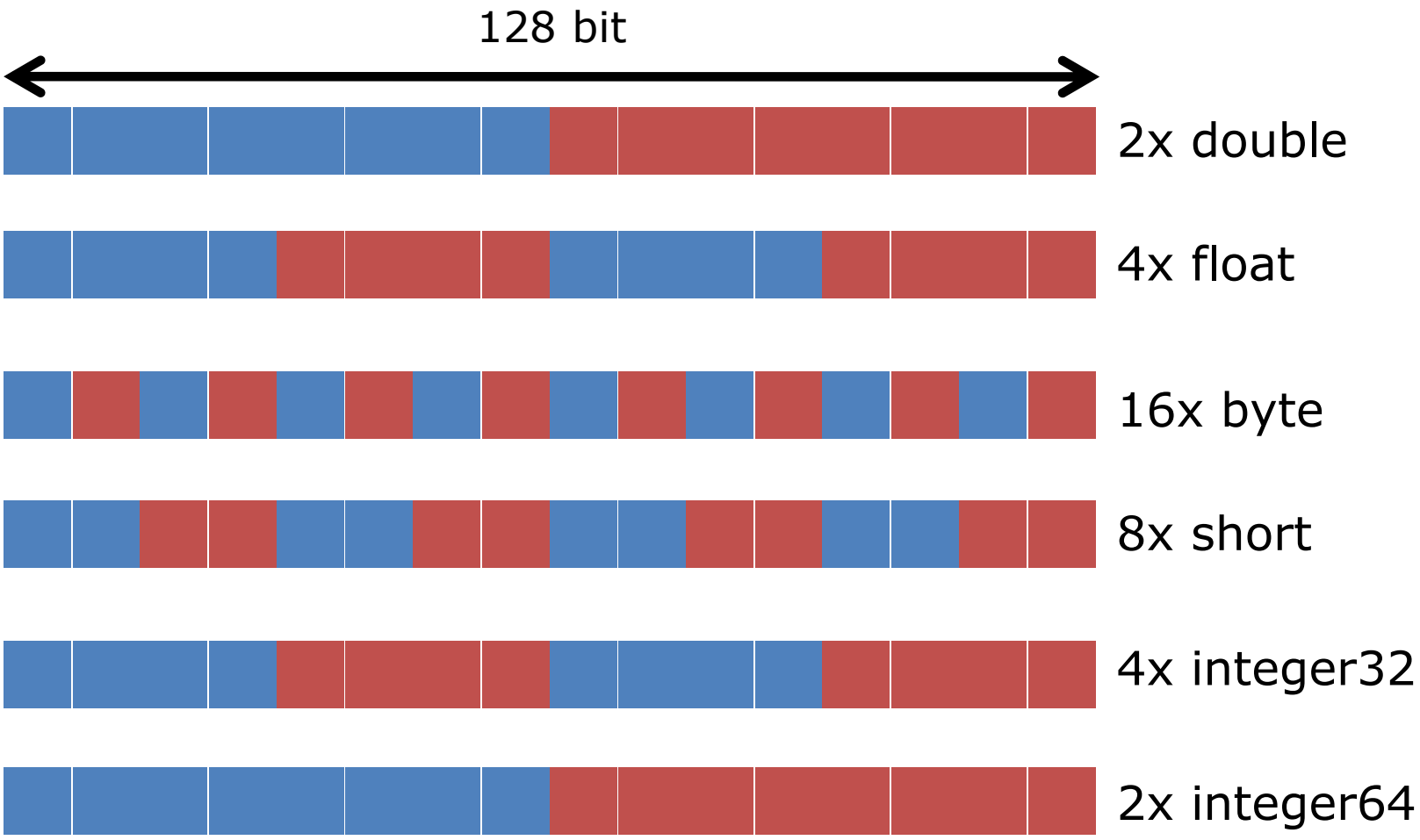
- Introduction to the Intel MIC Architecture
- Programming Models
  - Native Programming
  - Intel MKL
  - Offloading w/ explicit data transfers
  - Offloading w/ implicit data transfers
  - Vectorization

SIMD Instructions / Vectorization

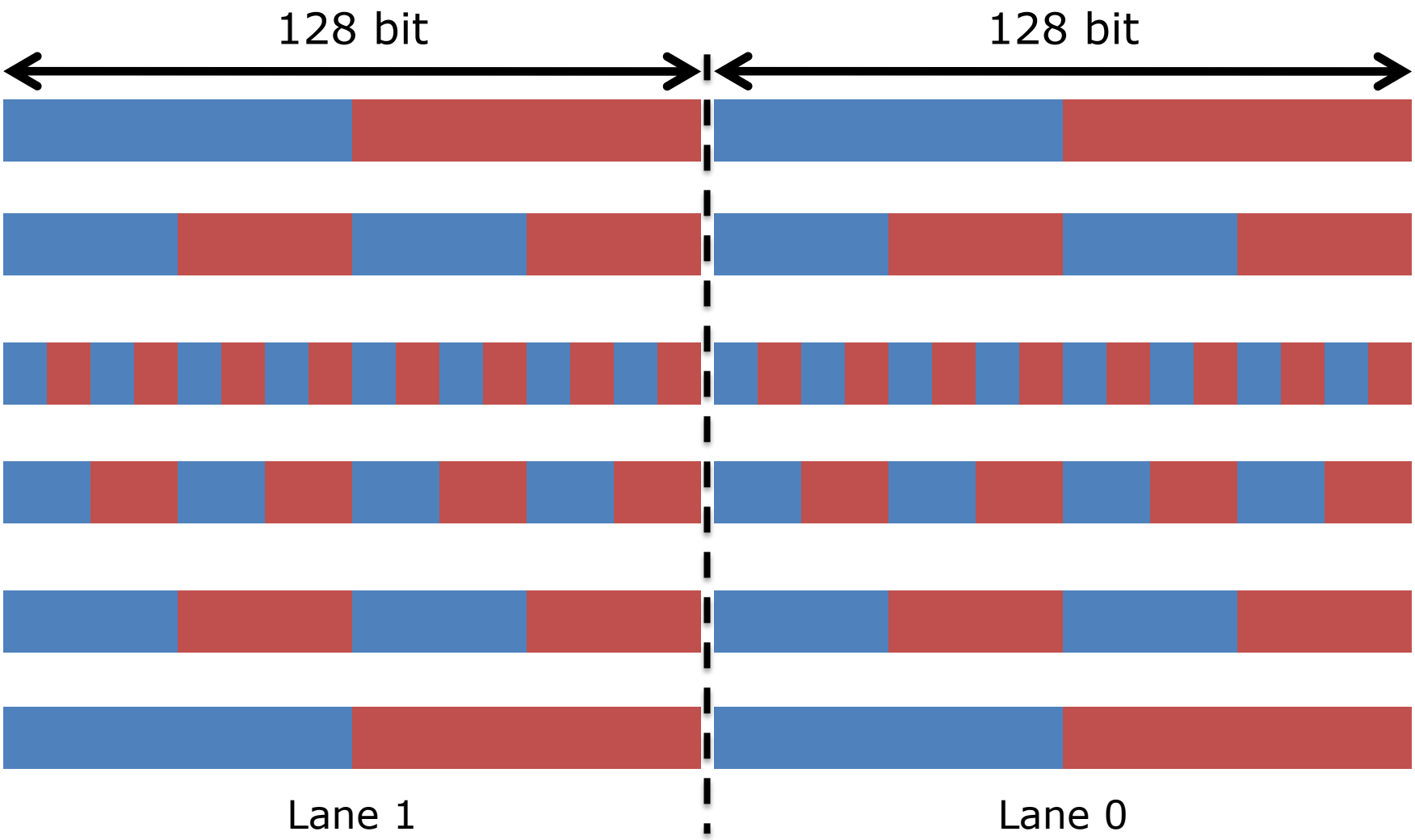
- SSE: Streaming SIMD extension
- SIMD: Single instruction, Multiple Data (Flynn's Taxonomy)
- E.g., SSE allows the identical treatment of 2 double, 4 floats and 4 integers at the same time



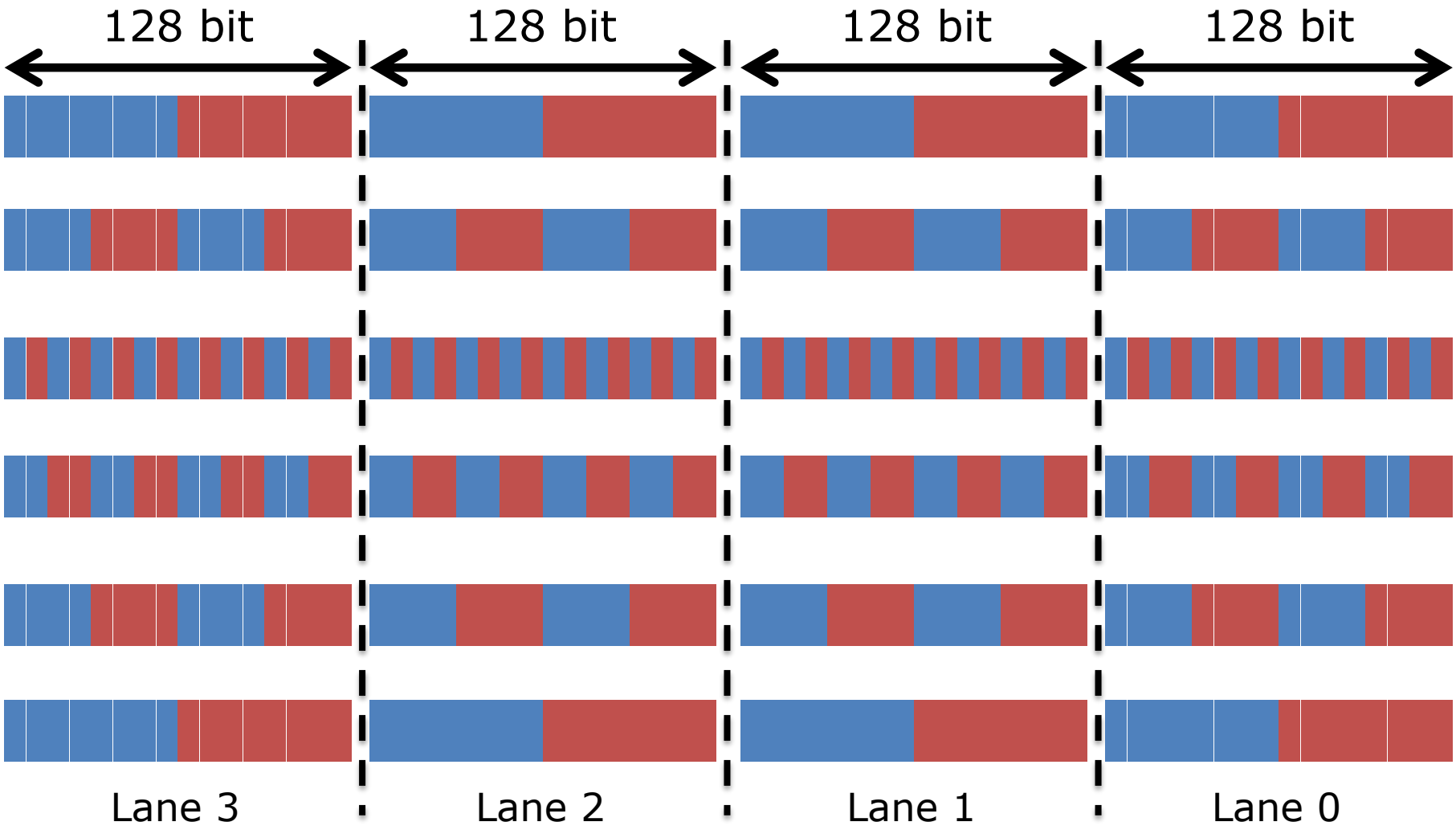
Vectorization: SSE Data Types



Evolution to Intel AVX



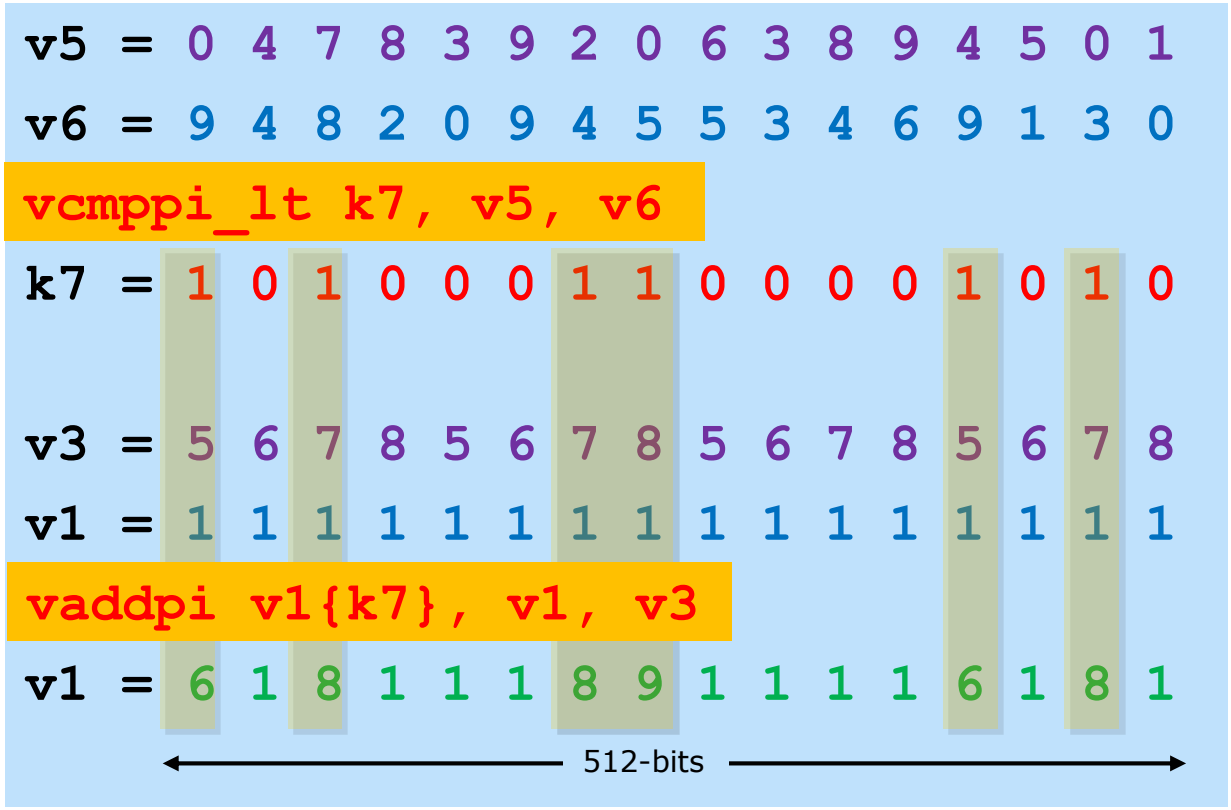
Intel Xeon Phi SIMD Instructions



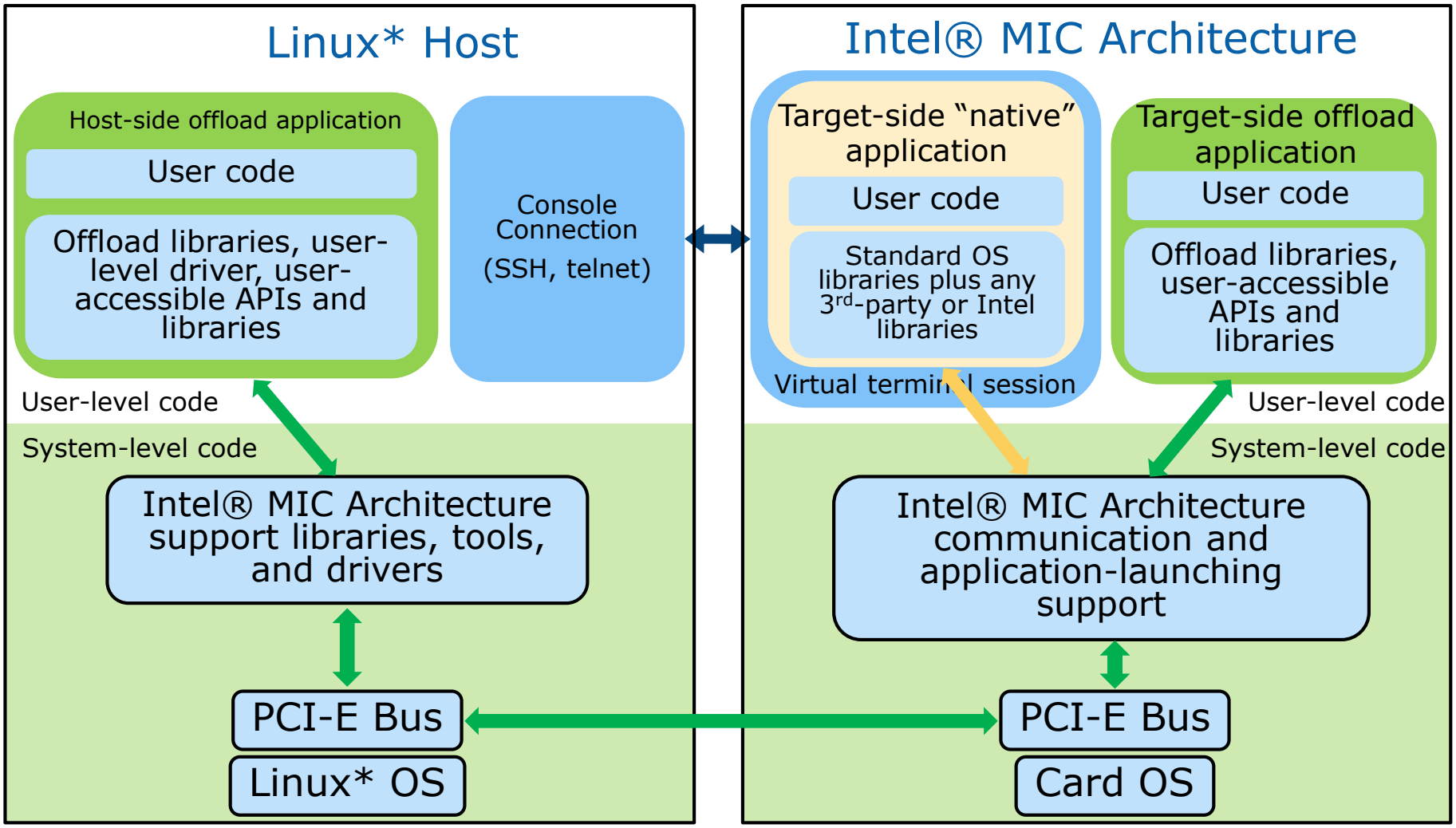
SIMD Width/Hardware Abstraction – Vectorization/SIMD Example

```
for (i = 0; i < 15; i++)
  if (v5[i] < v6[i])
    v1[i] += v3[i];
```

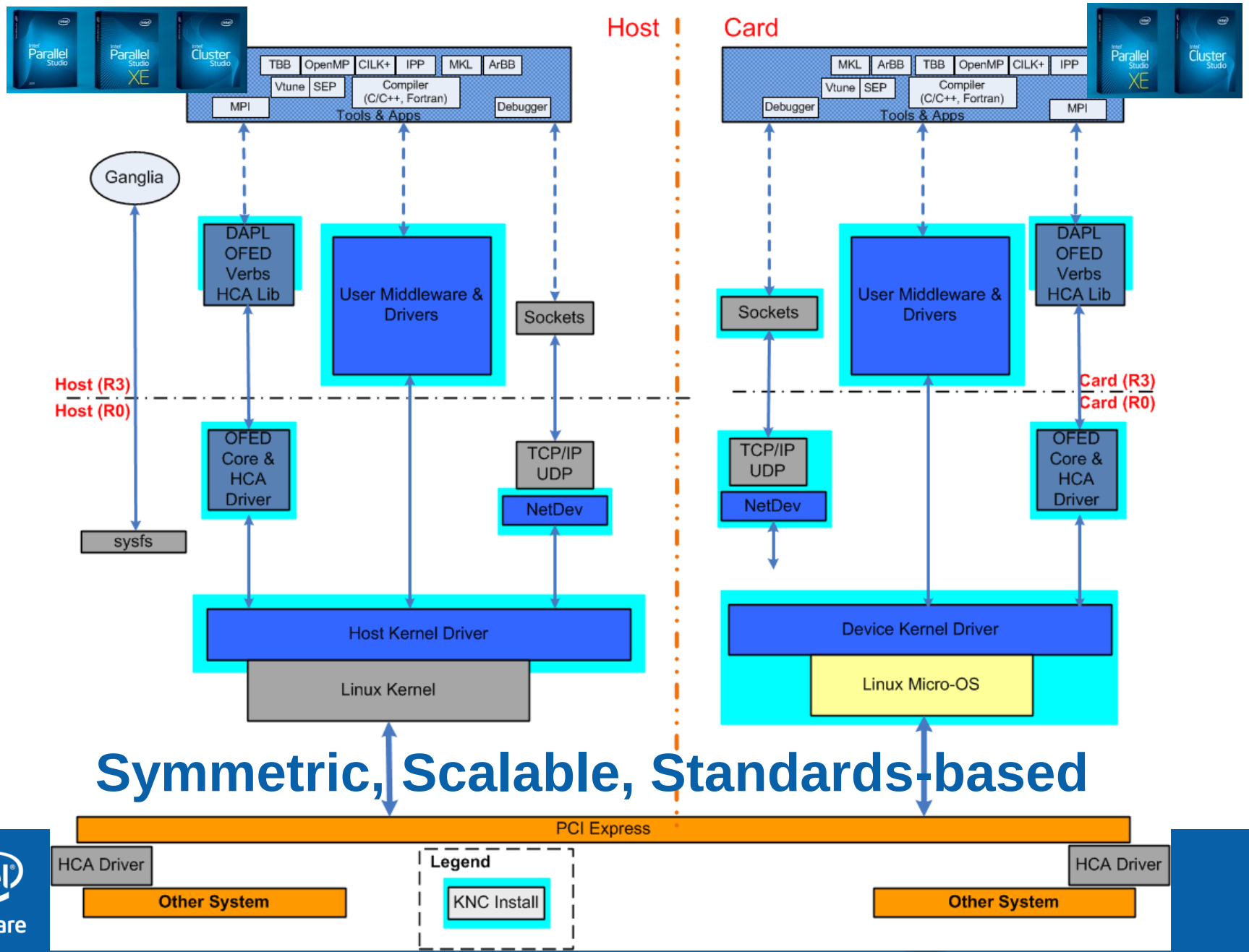
Note the lack of jumps or conditional code branches



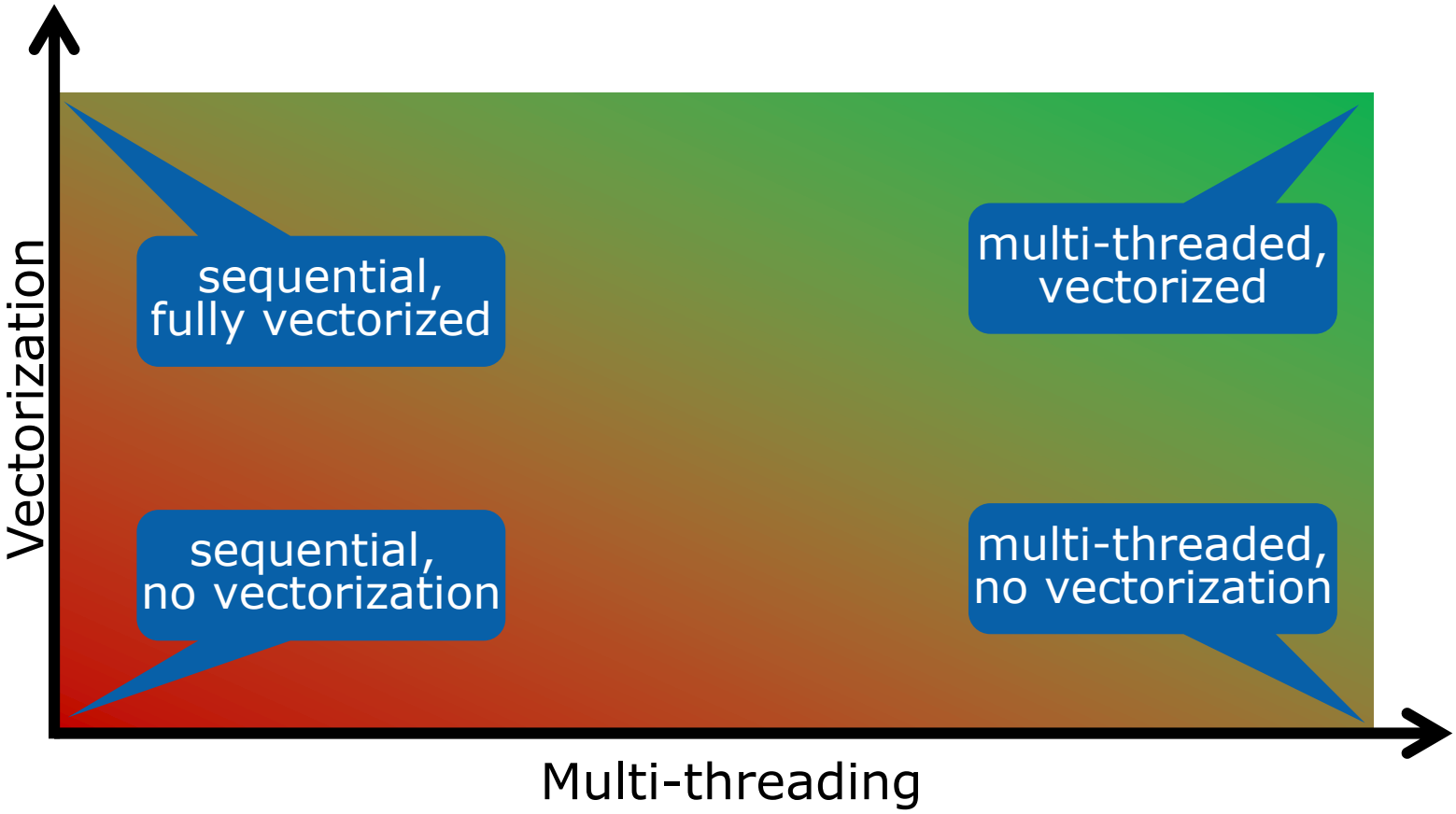
Intel® MIC Architecture Overview – Software Architecture



Intel® MIC Software Architecture Block View



Parallelization and Vectorization are Key



- Performance increasingly depends on both threading and vectorization
- Also true for "traditional" Xeon-based computing

Agenda

- Introduction to the Intel MIC Architecture
- Programming Models
  - Native Programming
  - Intel MKL
  - Offloading w/ explicit data transfers
  - Offloading w/ implicit data transfers
  - Vectorization

A few General Switches

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

Architecture Specific Switches

Functionality	Linux
Optimize for current machine	-xHOST
Generate SSE v1 code	-xSSE1
Generate SSE v2 code (may also emit SSE v1 code)	-xSSE2
Generate SSE v3 code (may also emit SSE v1 and v2 code)	-xSSE3
Generate SSE v3 code for Atom-based processors	-xSSE_ATOM
Generate SSSE v3 code (may also emit SSE v1, v2, and v3 code)	-xSSSE3
Generate SSE4.1 code (may also emit (S)SSE v1, v2, and v3 code)	-xSSE4.1
Generate SSE4.2 code (may also emit (S)SSE v1, v2, v3, and v4 code)	-xSSE4.2
Generate AVX code	-xAVX
Generate code for Intel Xeon Phi coprocessors	-mmic

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)

Compiler Optimizer Phases

ipo
ipo_inl
ipo_cp
ipo_align
ipo_modref
ipo_lpt
ipo_subst
ipo_ratt
ipo_vaddr
ipo_pdce
ipo_dp
ipo_gprel
ipo_pmerge
ipo_dstat
ipo_fps
ipo_ppi
ipo_unref
ipo_wp
ipo_dl
ipo_psplit

ilo
ilo_arg_prefetching
ilo_lowering
ilo_strength_reduction
ilo_reassociation
ilo_copy_propagation
ilo_convert_insertion
ilo_convert_removal
ilo_tail_recursion

hpo
hpo_analysis
hpo_openmp
hpo_threadization
hpo_vectorization
pgo
tcollect
offload
all

hlo
hlo_fusion
hlo_distribution
hlo_scalar_replacement
hlo_unroll
hlo_prefetch
hlo_loadpair
hlo_linear_trans
hlo_opt_pred
hlo_data_trans
hlo_string_shift_replace
hlo_ftae
hlo_reroll
hlo_array_contraction
hlo_scalar_expansion
hlo_gen_matmul
hlo_loop_collapsing

List of 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
…
```

Compiler Vectorization Report

- Linux
  `-vec-report`*n*

- 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

Compiler Vectorization Report

```
35:     subroutine fd( y )
36:     integer :: i
37:     real, dimension(10), intent(inout) :: y
38:     do i=2,10
39:        y(i) = y(i-1) + 1
40:     end do
41:     end subroutine fd
```

```
novec.f90(38): (col. 3) remark: loop was not vectorized: existence of
vector dependence.
novec.f90(39): (col. 5) remark: vector dependence: proven FLOW
dependence between y line 39, and y line 39.
novec.f90(38:3-38:3):VEC:MAIN_:  loop was not vectorized: existence of
vector dependence
```

Agenda

- Introduction to the Intel MIC Architecture
- Programming Models
  - Native Programming
  - Intel MKL
  - Offloading w/ explicit data transfers
  - Offloading w/ implicit data transfers
  - Vectorization

Intel® Math Kernel Library

- Highly optimized, multi-threaded (when appropriate) math libraries providing functions often found in:
  - Engineering and Manufacturing
  - Financial Services
  - Geological/Energy Industries
- Automatically optimized for the platform on which the functions are called:
  - Called on the Intel® MIC Architecture, they will make best use of SIMD and parallelism
- Intel® MKL Functional Domains with Intel® MIC Architecture support:
  - Linear Algebra: BLAS, LAPACK, LINPACK
  - Sparse BLAS
  - Fast Fourier Transforms (FFT) 1D/2D/3D FFT
  - Vector Math and Statistical Libraries

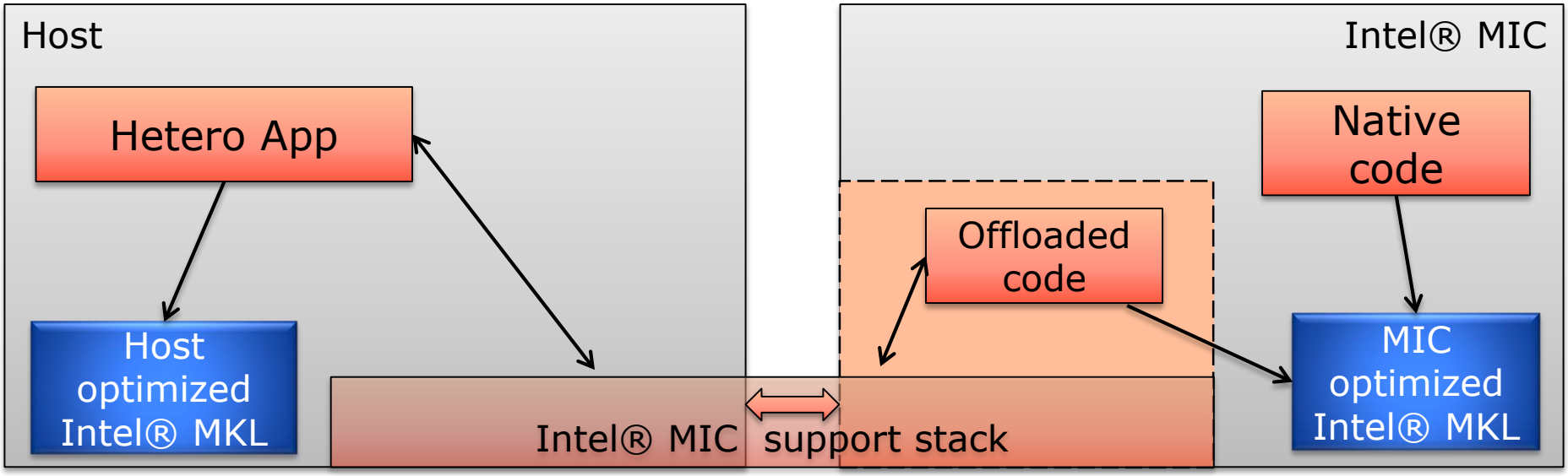
Intel® Math Kernel Library

- User can control parallelism in Intel® MKL like they can with OpenMP* (via `mkl_set_num_threads()`)
- Some Intel® MKL algorithms are designed to be used across multiple threads
  - Example: Vector Statistical Library
    - Generates statistically correct random number streams when called from multiple threads.
    - Employs technique where each thread looks at "windows" into a single random number sequence
  - Use on the Intel® MIC Architecture is the same as on the host
- Some Intel® MKL domains offer automatic offload of work to an available Intel® MIC Architecture coprocessor card

- Intel MKL is part of the Intel C++/Fortran Composer XE 2013

Intel® Math Kernel Library Use in Offload Code

- Native execution (of course)
- Identical usage syntax on host and coprocessor
- Functions called from the host execute on the host, functions called from the coprocessor execute on coprocessor
  - User is responsible for data transfer and execution management between the two domains



Intel® Math Kernel Library Offload Example

Calculates the new value of matrix *C* based on the matrix product of matrices *A* and *B*, and the old value of matrix *C*

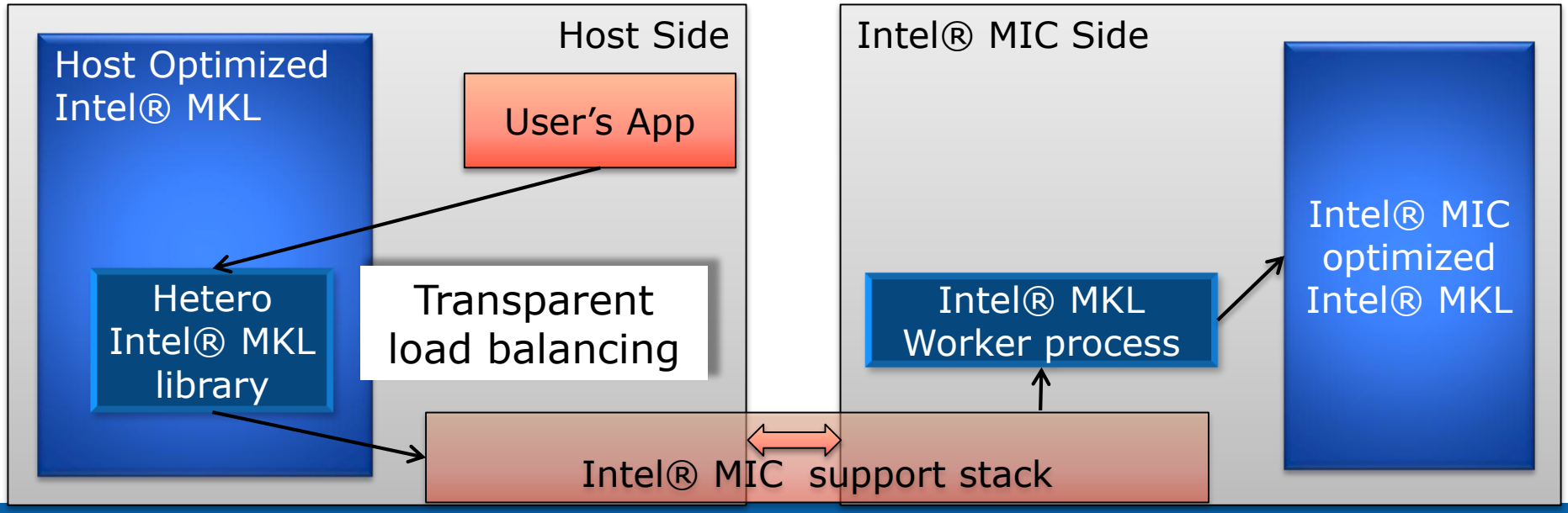
$$C \leftarrow \alpha AB + \beta C$$

where α and β values are scalar coefficients

```
void foo(…)  /*  Intel MKL Offload Example */
{
    float *A, *B, *C; /* Matrices */
            ⋮
    #pragma offload target(mic)
          in(transa, transb, N, alpha, beta) \
          in(A:length(N*N)) \
          in(B:length(N*N)) \
          inout(C:length(N*N))
     sgemm(&transa, &transb, &N, &N, &N, &alpha,
          A, &N, B, &N, &beta, C, &N);
            ⋮
}
```

Intel® Math Kernel Library Automatic Offload

- Transparent load balancing between host and coprocessors
- Initiated by calling `mkl_mic_enable()` on the host before calling Intel® MKL functions that implement Automatic Offload
- Call the function from the host code
  - No "_Offload" or "#pragma offload" needed
  - Intel® MKL is responsible for data transfer and execution management



Agenda

- Introduction to the Intel MIC Architecture
- Programming Models
  - Native Programming
  - Intel MKL
  - Offloading w/ explicit data transfers
  - Offloading w/ implicit data transfers
  - Vectorization

Options for Offloading Application Code

- Intel MKL "only" provides fix-function logic
- Programmers often need to offload application-specific code fragments to the coprocessor

- Intel Composer XE 2013 for MIC supports two models:
  - Offload pragmas
    - Only trigger offload when a MIC device is present
    - Safely ignored by non-MIC compilers
  - Offload keywords
    - Only trigger offload when a MIC device is present
    - Language extensions, need conditional compilation to be ignored

- Offloading and parallelism is orthogonal
  - Offloading only transfers control to the MIC devices
  - Parallelism needs to be exploited by a second model (e.g. OpenMP*)

Heterogeneous Compiler Data Transfer Overview

- *The host CPU and the Intel® MIC Architecture coprocessor do not share physical or virtual memory in hardware*

- Two offload data transfer models are available:
  1. Explicit Copy
     - Programmer designates variables that need to be ***copied*** between host and card *in the offload directive*
     - Syntax: Pragma/directive-based
     - C/C++ Example: `#pragma offload target(mic) in(data:length(size))`
     - Fortran Example: `!dir$ offload target(mic) in(a1:length(size))`
  2. Implicit Copy
     - Programmer marks variables that need to be ***shared*** between host and card
     - The same variable can then be used in both host and coprocessor code
     - Runtime *automatically maintains coherence* at the beginning and end of offload statements
     - Syntax: keyword extensions based
     - Example: `_Cilk_shared double foo;  _Cilk_offload func(y);`

Heterogeneous Compiler Offload using Explicit Copies

	C/C++ Syntax	Semantics
Offload pragma	`#pragma offload <clauses>` `  <statement block>`	Allow next statement block to execute on Intel® MIC Architecture or host CPU
Keyword for variable & function definitions	`__attribute__((target(mic)))`	Compile function for, or allocate variable on, both CPU and Intel® MIC Architecture

Entire blocks of code	`#pragma offload_attribute(push,`
	`  target(mic))`
	`⋮`
	`#pragma offload_attribute(pop)`
		Mark entire files or large blocks of code for generation on both host CPU and Intel® MIC Architecture

Heterogeneous Compiler Offload using Explicit Copies

	Fortran Syntax	Semantics
Offload directive	`!dir$ omp offload <clause>` `  <OpenMP construct>`	Execute next OpenMP* parallel construct on Intel® MIC Architecture
	`!dir$ offload <clauses>` `  <statement>`	Execute next statement (function call) on Intel® MIC Architecture

Keyword for variable/function definitions	`!dir$ attributes`
	`  offload:<MIC> :: <rtn-`
	`  name>`
		Compile function or variable for CPU and Intel® MIC Architecture

Heterogeneous Compiler: Offloading & OpenMP*

C/C++ Sequential

```
#pragma offload target(mic)
for (i=0; i<count; i++)
{
   a[i] = b[i] * c + d;
}
```

Fortran Sequential

```
!dir$ offload target(mic)
call routine()
subroutine routine()
  do i=1, count
     A(i) = B(i) * c + d
  end do
end subroutine
```

C/C++ OpenMP*

```
#pragma offload target(mic)
#pragma omp parallel for
for (i=0; i<count; i++)
{
   a[i] = b[i] * c + d;
}
```

Fortran OpenMP*

```
!dir$ omp offload target(mic)
!$omp parallel do
   do i=1, count
     A(i) = B(i) * c + d
   end do
!$omp end parallel do
```

Heterogeneous Compiler – Conceptual Transformation

Source Code

```
main()
{
f();
}
```

```
f()
{
  #pragma offload
    a = b + g();
}
```

```
__attribute__
((target(mic))) g()
{
}
```

Linux* Host Program

```
main()
{
copy_code_to_mic();
f();
unload_mic();
}
```

```
f() {
  if (mic_available()){
    send_data_to_mic();
    start_on_mic(f_part_mic);
    receive_data_from_mic();
  } else
    f_part_host();
}
```

```
f_part_host()
 {a = b + g();}
```

```
g() {…}
```

Intel ®MIC Program

```
f_part_mic()
 {a = b + g_mic();}
```

```
g_mic() {…}
```

This all happens automatically when you issue a single compile command

Heterogeneous Compiler Offload using Explicit Copies – Clauses

Variables and pointers restricted to scalars, structs of scalars, and arrays of scalars

Clauses	Syntax	Semantics
Target specification	`target( name[:card_number] )`	Where to run construct
Conditional offload	`if (condition)`	Boolean expression
Inputs	`in(var-list modifiers`$_{opt}$`)`	Copy from host to coprocessor
Outputs	`out(var-list modifiers`$_{opt}$`)`	Copy from coprocessor to host
Inputs & outputs	`inout(var-list modifiers`$_{opt}$`)`	Copy host to coprocessor and back when offload completes
Non-copied data	`nocopy(var-list modifiers`$_{opt}$`)`	Data is local to target
Async. offload	`signal(signal-slot)`	Trigger async offload
Async. offload	`wait(signal-slot)`	Wait for completion

Heterogeneous Compiler Offload using Explicit Copies – Modifiers

Variables and pointers restricted to scalars, structs of scalars, and arrays of scalars

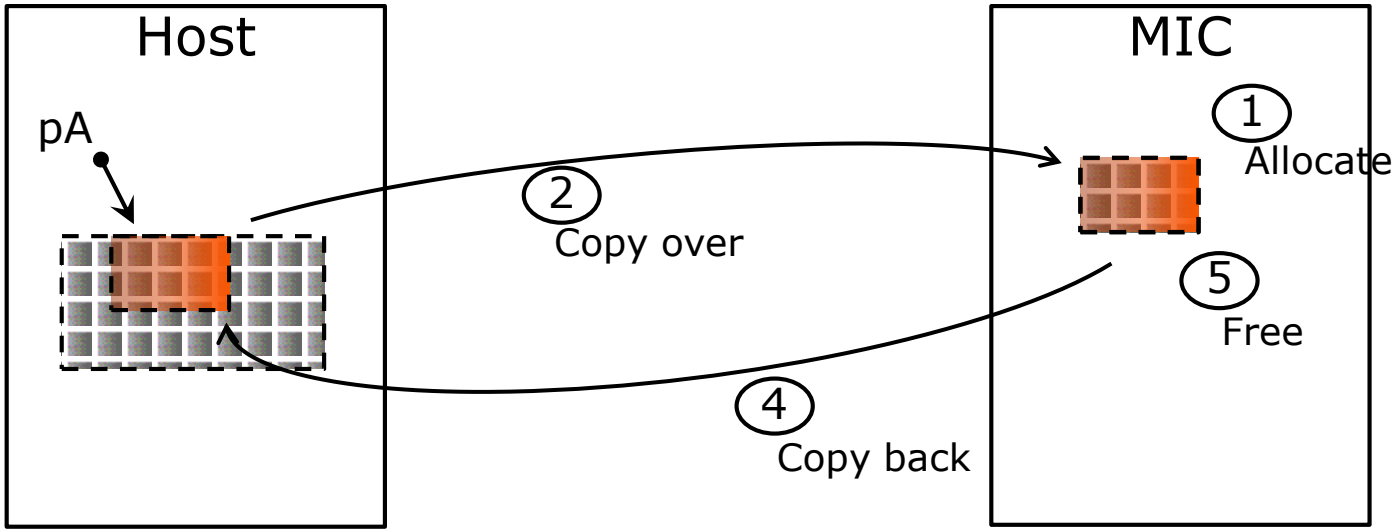
Modifiers	Syntax	Semantics
Specify pointer length	`length(element-count-expr)`	Copy N elements of the pointer's type
Control pointer memory allocation	`alloc_if ( condition )`	Allocate memory to hold data referenced by pointer if condition is TRUE
Control freeing of pointer memory	`free_if ( condition )`	Free memory used by pointer if condition is TRUE
Control target data alignment	`align ( expression )`	Specify minimum memory alignment on target

Heterogeneous Compiler – Offload using Explicit Copies – Modifier Example

```
float reduction(float *data, int numberOf)
{
  float ret = 0.f;
#pragma offload target(mic) in(data:length(numberOf))
  {
#pragma omp parallel for reduction(+:ret)
     for (int i=0; i < numberOf; ++i)
        ret += data[i];
  }
  return ret;
}
```

Note: copies `numberOf` elements to the coprocessor, not `numberOf*sizeof(float)` bytes – the compiler knows `data`'s type

Heterogeneous Compiler – Offload using Explicit Copies – Data Movement



- Default treatment of `in`/`out` variables in a `#pragma offload` statement
  - At the start of an offload:
    - Space is allocated on the coprocessor
    - `in` variables are transferred to the coprocessor
  - At the end of an offload:
    - `out` variables are transferred from the coprocessor
    - Space for both types (as well as `inout`) is deallocated on the coprocessor

```
#pragma offload inout(pA:length(n))
{...}
```
3

Double Buffering Example - Main

```
int main(int argc, char* argv[])   {
   int i;
   double st_time, end_time;
   double sync_tm, async_in_tm;
   //  Allocate & initialize in1, res1,
   // in2, res2 on the host
```

Allocate arrays on target:
alloc_if(1)

Retain for duration of sample:
free_if(0)

in1 and in2 represent 2 separate buffers.

```
#pragma offload_transfer target(mic:0) in(cnt)  \
        nocopy(in1, res1, in2, res2 : length(cnt) \
        alloc_if(1) free_if(0) )

   do_async_in();

   // Validate results and print timings
```

Free target allocations:
free_if(1)

```
#pragma offload_transfer target(mic:0) \
        nocopy(in1, res1, in2, res2 : length(cnt) \
        alloc_if(0) free_if(1) )
```

Double Buffering – do_asynch_in, evens

```
void do_async_in() {
  float lsum;
  int i;
  lsum = 0.0f;
```

Begin an initial transfer of the first dataset for the target to work on. Transfer begins immediately, is non-blocking, and will signal when complete.

```
#pragma offload_transfer target(mic:0)   \
      in(in1 : length(cnt) alloc_if(0) free_if(0) ) signal(in1)
```

```
  for (i=0; i < iter; i++) {
    if (i%2 == 0) {
```

For even loop iterations (except the final), first start another non-blocking transfer of the next dataset.

```
#pragma offload_transfer target(mic:0) if(i!=iter-1)  \
      in(in2 : length(cnt) alloc_if(0) free_if(0) ) \
      signal(in2)
```

```
#pragma offload target(mic:0) nocopy(in1) wait(in1)  \
      out(res1 : length(cnt) alloc_if(0) free_if(0) )
{
  compute(in1, res1);
}
```

```
    lsum = lsum + sum_array(res1);
  }
```

While that transfer progresses, process the previous dataset through the compute() function, first waiting for its transfer to complete, then return a result. Execution on the host waits for the function to return.

Double Buffering – do_asynch_in, odds

```
    else {

      #pragma offload_transfer target(mic:0) if(i!=iter-1) \
              in(in1 : length(cnt) alloc_if(0) free_if(0) ) \
              signal(in1)


      #pragma offload target(mic:0) nocopy(in2) wait(in2) \
              out(res2 : length(cnt) alloc_if(0) free_if(0) )
      {
        compute13(in2, res2);
      }

        lsum = lsum + sum_array(res2);
      }
    }
    async_in_sum = lsum / (float) iter;
}
```

For odd iterations (except the final), work on the other buffer. Start another non-blocking transfer.

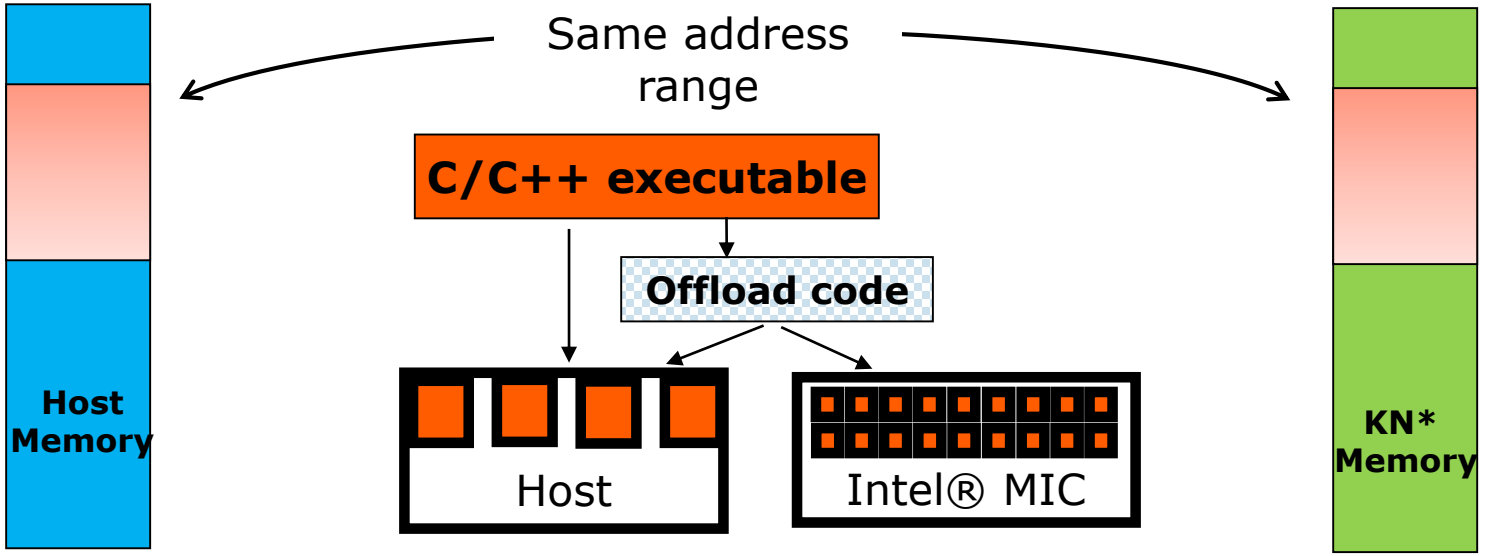
Offload the compute function, and host waits for the result. Repeat, alternating between the buffers (in1 & in2).

Agenda

- Introduction to the Intel MIC Architecture
- Programming Models
  - Native Programming
  - Intel MKL
  - Offloading w/ explicit data transfers
  - Offloading w/ implicit data transfers
  - Vectorization

Heterogeneous Compiler Offload using Implicit Copies

- Section of memory maintained at the same virtual address on both the host and Intel® MIC Architecture coprocessor
- Reserving same address range on both devices allows
  - Seamless sharing of complex pointer-containing data structures
  - Elimination of user marshaling and data management
  - Use of simple language extensions to C/C++



Heterogeneous Compiler Offload using Implicit Copies

- When "shared" memory is synchronized
  - Automatically done around offloads (so memory is only synchronized on entry to, or exit from, an offload call)
  - Only modified data is transferred between CPU and coprocessor
- Dynamic memory you wish to share must be allocated with special functions: `_Offload_shared_malloc`, `_Offload_shared_aligned_malloc`, `_Offload_shared_free`, `_Offload_shared_aligned_free`
- Allows transfer of C++ objects
  - Pointers are no longer an issue when they point to "shared" data
- Well-known methods can be used to synchronize access to shared data and prevent data races *within* offloaded code
  - E.g., locks, critical sections, etc.

This model is integrated with the Intel® Cilk™ Plus parallel extensions

Note: Not supported on Fortran - available for C/C++ only

Heterogeneous Compiler
Keyword _Cilk_shared for Data/Functions

What	Syntax	Semantics
Function	`int _Cilk_shared f(int x) { return x+1; }`	Versions generated for both CPU and card; may be called from either side
Global	`_Cilk_shared int x = 0;`	Visible on both sides
File/Function static	`static _Cilk_shared int x;`	Visible on both sides, only to code within the file/function
Class	`class _Cilk_shared x {…};`	Class methods, members, and and operators are available on both sides
Pointer to shared data	`int _Cilk_shared *p;`	*p* is local (not shared), can point to shared data
A shared pointer	`int *_Cilk_shared p;`	*p* is shared; should only point at shared data

Entire blocks of code	`#pragma offload_attribute( push, _Cilk_shared)`
	⋮
	`#pragma offload_attribute(pop)`
		Mark entire files or large blocks of code _Cilk_shared using this pragma

Heterogeneous Compiler
Implicit: Offloading using `_Offload`

Feature	Example	Description
Offloading a function call	`x = _Cilk_offload func(y);`	`func` executes on coprocessor if possible
	`x = _Cilk_offload_to (card_number) func(y);`	`func` ***must*** execute on specified coprocessor
Offloading asynchronously	`x = _Cilk_spawn _Cilk_offload func(y);`	Non-blocking offload
Offload a parallel for-loop	`_Offload _Cilk_for(i=0; i<N; i++) { a[i] = b[i] + c[i]; }`	Loop executes in parallel on target. The loop is implicitly outlined as a function call.

Heterogeneous Compiler
Implicit: Offloading using _Offload Example

```
void findpi()
{
  int count = 10000;

  // Initialize shared global
  // variables
  pi = 0.0f;

  //  Compute pi on target
  _Cilk_offload
    compute_pi(count);

  pi /= count;
}
```

```
// Shared variable declaration for pi
_Cilk_shared float pi;

// Shared function declaration for
// compute
_Cilk_shared void compute_pi(int count)
{
    int i;

    #pragma omp parallel for \
                reduction(+:pi)
    for (i=0; i<count; i++)
    {
        float t = (float)((i+0.5f)/count);
        pi += 4.0f/(1.0f+t*t);
    }
}
```

Heterogeneous Compiler Command-line Options

Offload-specific arguments to the Intel® Compiler:

- Produce a report of offload data transfers at compile time (not runtime)

  `-opt-report-phase:offload`

- Add Intel® MIC Architecture compiler switches

  `-offload-copts:"switches"`

- Add Intel® MIC Architecture archiver switches

  `-offload-aropts:"switches"`

- Add Intel® MIC Architecture linker switches

  `-offload-ldopts:"switches"`

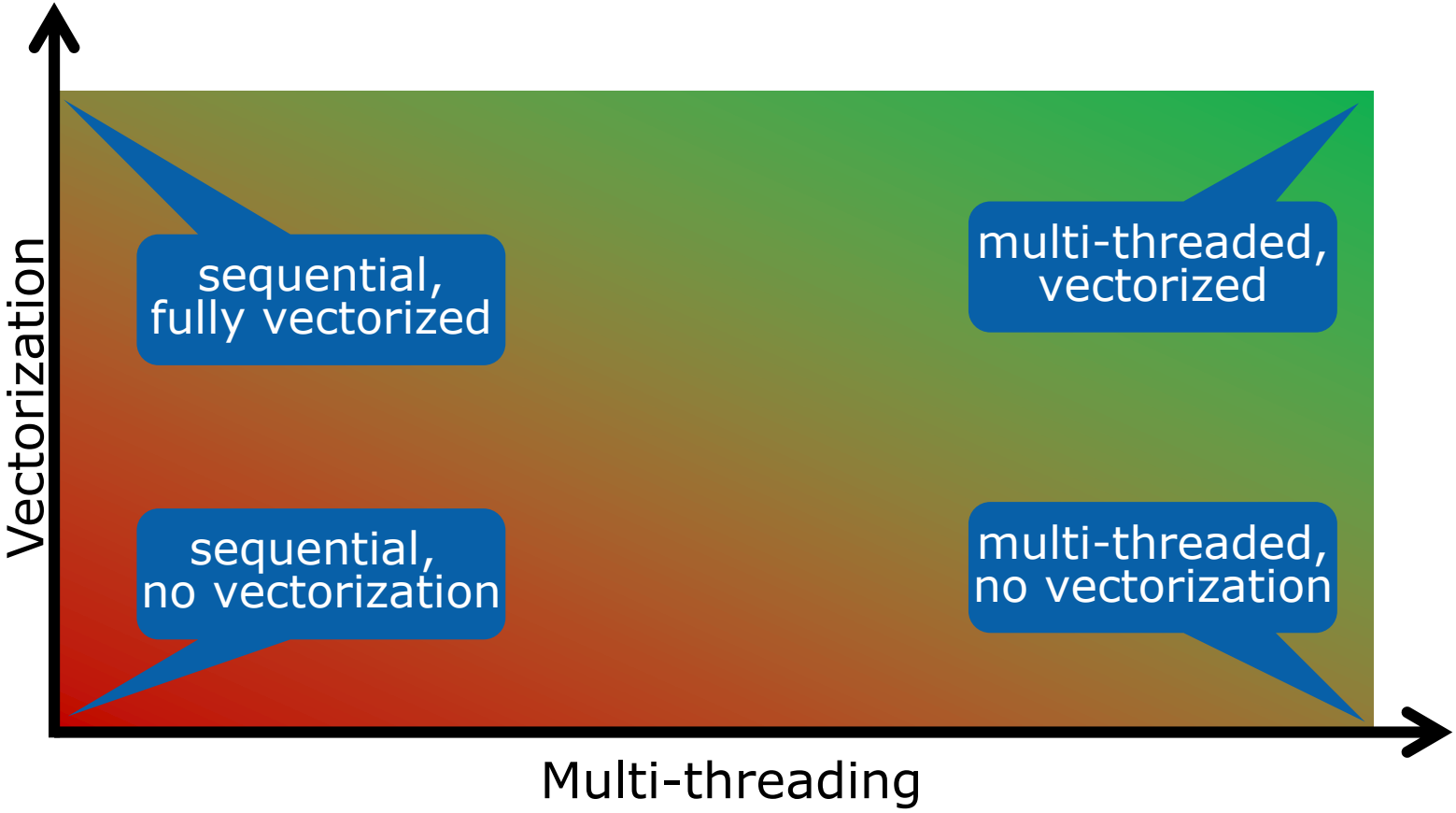
Example:

```
icc -g -O2 -mkl -offload-build -offload-copts:"-g -03"
     -offload-ldopts:"-L/opt/intel/composerxe_mic/mkl/lib/mic"
     foo.c
```

Agenda

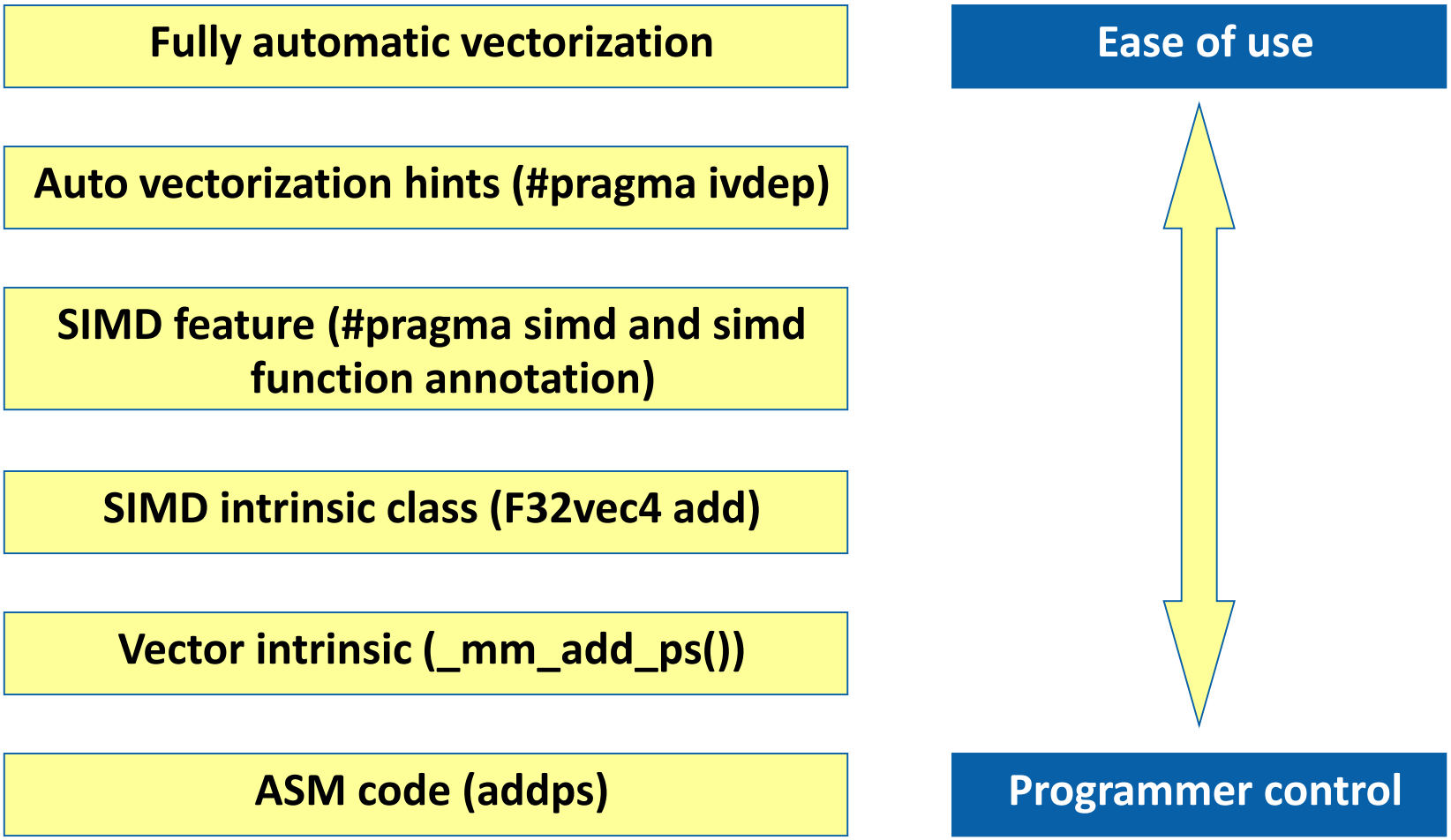
- Introduction to the Intel MIC Architecture
- Programming Models
  - Native Programming
  - Intel MKL
  - Offloading w/ explicit data transfers
  - Offloading w/ implicit data transfers
  - Vectorization

Parallelization and Vectorization are Key



- Performance increasingly depends on both threading and vectorization
- Also true for "traditional" Xeon-based computing

Positioning of SIMD Features



Auto-vectorization

- Be "lazy" and try auto-vectorization first
  - If the compiler can vectorize the code, why bother
  - If it fails, you can still deal w/ (semi-)manual vectorization

- Compiler switches of interest:
  - **`-vec`** (automatically enabled with **`-O3`**)
  - **`-vec-report`**
  - **`-opt-report`**

Why Didn't My Loop Vectorize?

- Linux
  `-vec-report`*n*

  Windows
  `/Qvec-report`*n*

- 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

Compiler Vectorization Report

```
35:     subroutine fd( y )
36:     integer :: i
37:     real, dimension(10), intent(inout) :: y
38:     do i=2,10
39:        y(i) = y(i-1) + 1
40:     end do
41:     end subroutine fd
```

```
novec.f90(38): (col. 3) remark: loop was not vectorized: existence of
vector dependence.
novec.f90(39): (col. 5) remark: vector dependence: proven FLOW
dependence between y line 39, and y line 39.
novec.f90(38:3-38:3):VEC:MAIN_:  loop was not vectorized: existence of
vector dependence
```

When Vectorization Fails …

- Most frequent reason: Data dependencies
  - Simplified: Loop iterations must be independent
- Many other potential reasons
  - Alignment
  - Function calls in loop block
  - Complex control flow / conditional branches
  - Loop not "countable"
    - E.g. upper bound not a run time constant
  - Mixed data types (many cases now handled successfully)
  - Non-unit stride between elements
  - Loop body too complex (register pressure)
  - Vectorization seems inefficient
  - Many more … but less likely to occur

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
```

Disambiguation Hints
The `restrict` Keyword for Pointers

Linux	Windows
`-restrict`	`/Qrestrict`
`-std=c99`	`/Qstd=c99`

- Assertion to compiler, that only the pointer or a value based on the pointer - such as (pointer+1) - will be used to access the object it points to
- Only available for C, not C++

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

// two-dimension example:
void mult(int a[][NUM],int b[restrict][NUM]);
```

Unsupported Loop Structure

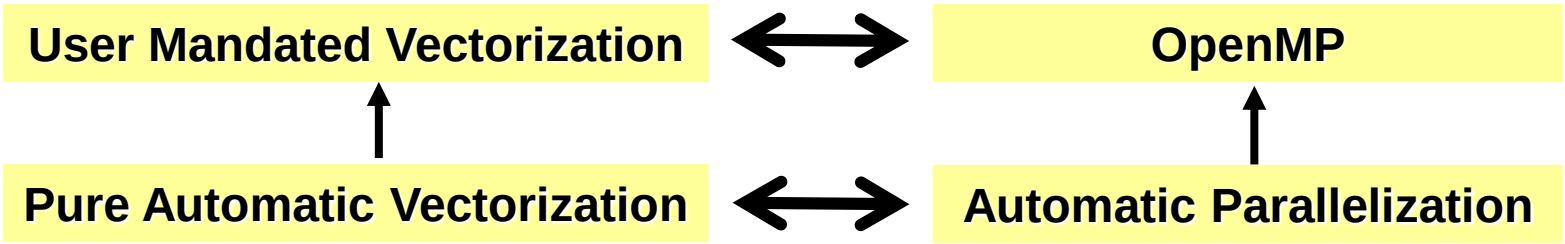
- Unsupported loop structure usually means, the compiler can't construct a runtime expression for the loop's trip-count
  - E.g. a while-loop where the number of iterations cannot be determined at (run-time) start of loop
  - Upper/lower bound of a for-loop cannot be a determined to be loop-invariant

```
struct _x { int d; int bnd;};
doit1(int *a, struct _x *x)
{
  for (int i=0; i < x->bnd; i++)
    a[i] = 0;
}
```

```
struct _x { int d; int bnd;};
doit1(int *a, struct _x *x)
{
  int ub = x->bnd;
  for (int i=0; i < ub; i++)
    a[i] = 0;
}
```

Guided Vectorization

- SIMD Directives
  - The SIMD directive provides additional information to compiler to enable vectorization of loops (at this time only inner loop)
  - More a command to the compiler than a hint
  - The compiler's heuristics are completely overwritten as long as a clear logical fault is not being introduced
  - Inspired by the OpenMP directives:



SIMD Directive Notation

C/C++: `#pragma simd` *[clause [,clause] …]*

Fortran: `!DIR$ SIMD` *[clause [,clause] …]*

- Without any additional clause, the directive enforces vectorization of the (innermost) loop

```
void add_fl(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 the SIMD directive, vectorization will fail (too many pointer references to do a run-time overlap-check).

Clauses of SIMD Directive

`vectorlength(num1, num2, …, numN)`

- Each iteration in the vector loop will execute the computation equivalent to the VL-iters of scalar loop execution.

`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, varX is incremented by stepX,. Every iteration of the vector loop, therefore increments varX by VL*stepX

`reduction(operator:var1, var2,…, varN)`

- perform vector reduction of operator kind has to be applied to varX

`[no]assert`

- assert or do not assert when the vectorization fails. Default is to assert for SIMD pragma.

Sample: Reductions

```
float sprod(float *a, float *b, int n)
{
  float sum = 0.0f;
  for (int k=0; k<n; k++)
    sum += a[k] * b[k];
  return sum;
}
```

```
void sprod(float *a, float *b, int n)
{
  float sum = 0.0f;
#pragma simd vectorlength(16) reduction(+:sum)
  for (int k=0; k<n; k++)
    sum += a[k] * b[k];
  return sum;
}
```

Vectorizable Mathematical Functions

- Calls to most mathematical functions in loops can be "vectorized" by calling their vector versions in libsvml (Short Vector Math Library)
  - libsvml is optimized for latency
  - Intel® MKL is optimized for throughput
  - Routines in libsvml can be called explicitly (see manual)
  - KNC will have machine instructions for transcendentals

acos	ceil	fabs	round
acosh	cos	floor	sin
asin	cosh	fmax	sinh
asinh	erf	fmin	sqrt
atan	erfc	log	tan
atan2	erfinv	log10	tanh
atanh	exp	log2	trunc
cbrt	exp2	pow	

Vectorization: Array Section Notation

- Part of Intel Cilk Plus
- Array Section Notation

  <array base> [<lower bound> : <length> [: <stride>]]
  [<lower bound> : <length> [: <stride>]].....

- Note that length is chosen.
  - Not upper bound as in Fortran
    ([lower bound : upper bound])

```
A[:]          // All elements of vector A
B[2:6]        // Elements 2 to 7 of vector B
C[:][5]       // Column 5 of matrix C
D[0:3:2]      // Elements 0,2,4 of vector D
E[0:3][0:4]   // 12 elements from E[0][0] to E[2][3]
```

Operator Maps

- Most arithmetic and logic operators for C/C++ basic data types are available for array sections:

```
+, -, *, /, %, <,==,!=,>,|,&,^,&&,||,!,-(unary),
  +(unary),++,--, +=, -=, *=, /=, *(p)
```

- An operator is implicitly mapped to all the elements of the array section operands:

```
a[0:s]+b[2:s] => {a[i]+b[i+2], forall (i=0;i<s;i++)}
```

  - Operations are parallel among all the elements
  - Array operands must have the same *rank*
  - Scalar operand is automatically expanded to fill the whole section

```
a[0:s]*c => {a[i]*c, forall (i=0;i<s;i++)}
```

Reduction

Combine all the elements in an array section using a predefined operator, or a user function

```
int a[] = {1,2,3,4};
sum = __sec_reduce_add(a[:]); // sum is 10
res = __sec_reduce(fn, a[:], 0);
        // apply function fn to all
        // elements in a[], identity value is 0
```

Other reductions:

__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

Manual Vectorization

- If automatic and guided vectorization do not meet expected results, programmers can manually vectorize
  - Intrinsic functions are functions with "special" meaning to the compiler
  - Vector intrinsics map to the machine's vector instructions
- Last resort of getting performance
  - Maximum of control
  - (Hopefully) maximum of performance
  - (Surely) maximum of "pain": prone to errors, cumbersome

Sample: Manual Vectorization

```
void vecmul(float *a, float *b, float *c, int n)
{
  for (int k=0; k<n; k++)   c[k] = a[k] * b[k];
}
```

- For the following slide we make the following assumptions (otherwise, we'd run out of space)
  - Input and output data is properly aligned to 64 bytes
  - Vector length is a multiple of the vector length

- If assumptions do not hold, add code to:
  1. Peel off iterations 0..m to get rid of alignment issue
  2. Have a vectorized loop to do the work
  3. Peel off iterations n..N-1 to deal with remaining data

Sample: Manual Vectorization

```
#include <immintrin.h>

void vecmul(float *a, float *b, float *c, int n) {
    int i;
    __m512 va;
    __m512 vb;
    __m512 vc;
    for (i = 0; i < n; i += 16,
                       a += 16, b += 16, c += 16) {
        _mm_prefetch((const char*) (a + 16), _MM_HINT_T0);
        va = _mm512_load_ps(a);
        vb = _mm512_extload_ps(b, _MM_UPCONV_PS_NONE,
                                  _MM_BROADCAST32_NONE,
                                  _MM_HINT_NONE);
        vc = _mm512_mul_ps(va, vb);
        _mm512_store_ps(c, vc);
    }
}
```

Vector registers

- Loop unrolling by 16 (i.e. vector length)
- Increment pointers

Vector instructions

Questions?