Explicit Vectorization (of Functions) with OpenMP 4.0

This lab focuses on the use of OpenMP* 4.0's SIMD functions within OpenMP parallel regions, covering the following topics:

1. How to balance load among threads.
2. The overhead of calling scalar functions from vector loops.
3. How to correctly use "omp declare simd".
4. Optimization beyond "omp parallel" and "omp simd".

Setup

Building and Running the Application

Begin by loading the Intel Compiler 15.0 Beta on Beacon:

```
module swap intel-compilers intel-compilers/2015.0.008
```

Use the scripts provided to compile and execute the Mandelbrot example.

To run on the host:

```
sh –x make_n_run.sh
```

To run on the MIC:

```
sh –x make_n_run_mic.sh
```

Do this now for both platforms and record the run times below:

Runtime (Host):	
Runtime (MIC):	

Is this what you expected?

The code's hotspot is the Mandelbrot function in mandel.cpp. Examine the source code in this file – can you see anything that explains the difference in run times that you have observed?

Remember: an Intel® Xeon® processor has a low number of high-clocked cores, whereas an Intel® Xeon Phi™ coprocessor has a high number of low-clocked cores. Applications that do not exploit thread and vector parallelism will not see any speed-up on the coprocessor.

Thread Parallelism

Adding Thread Parallelism to an Application

Our Mandelbrot example performs so poorly because it is serial (i.e. single-threaded) and scalar (i.e. not vectorized). The first step to improving its performance is to introduce multi-threading, for which we will use the OpenMP shorthand "omp parallel for". The behaviour of this shorthand is identical to an "omp for" nested within an "omp parallel" region.

As a reminder, the syntax for C is as follows:

```
#pragma omp parallel for [clause[, clause …]]
```

where clause can be any of the clauses accepted by the parallel or for directives.

Which of the loops in Mandelbrot() should be parallelised? Once you have decided, add the "omp parallel" pragma to this loop.

Build and re-run the application, then record the new run times:

Runtime (Host):	
Runtime (MIC):	

How has the run time changed? How does the speed-up achieved on each platform compare to the number of cores?

Current generation Intel® Xeon Phi™ coprocessors cannot issue instructions back-to-back from a single thread – at least two threads are required per core in order to achieve maximum performance. It is therefore not unusual to see speed-ups higher than the number of cores when parallelism is introduced.

Load Balancing

The run time of an OpenMP application's parallel section is equivalent to the run time of the longest task. Ensuring that each thread has an equivalent amount of work is therefore an important step in tuning a parallel application's performance.

Examine again the source code in mandel.cpp. Can you see anything that would contribute to load imbalance?

OpenMP supports a number of different "schedules" for parallel loops, which control the mapping of loop iterations to threads. These are specified using the "schedule" clause of "omp for":

```
#pragma omp parallel for schedule(kind[, chunk size])
```

where kind is one of the following values:

`static`	Divide the loop into equal-sized chunks (or as equal as possible). By default, chunk size is (loop count / # threads).
`dynamic`	Threads work on chunk size loop iterations at a time. When a thread finishes with one chunk, it retrieves the next from a work queue. By default, chunk size is 1.
`guided`	Similar to dynamic scheduling, but the chunk size starts large and decreases (automatically) to handle imbalance. The chunk size parameter specifies the minimum chunk size. By default, chunk size is approximately (loop count / # threads).
`auto`	The compiler is free to choose any possible mapping of loop iterations to threads.
`runtime`	Uses the schedule specified in the "OMP_SCHEDULE" environment variable at runtime.

Which of these schedules do you expect to give the best performance in this case?

Build and re-run the application with different schedules, and record the best new run times below:

Runtime (Host):	
Runtime (MIC):	

How has the run time changed?

The default chunk sizes may not give the best performance. In particular, dynamic scheduling with a chunk size of 1 may be unwise, due to the extra overhead of removing tasks from the work queue. Tuning the chunk size for an application can therefore have dramatic performance impact – this is left as an exercise to the reader.

Vectorization

The OpenMP 4.0 standard introduces two different ways to enable vectorization:

1. SIMD Loops
   Specify loops that should be vectorized by the compiler.

2. SIMD Functions
   Specify functions that will be called by vectorized loops.

Most applications will require some combination of these two approaches.

SIMD Loops

Start by marking the *x* loop in Mandelbrot() as being a SIMD loop. As a reminder, the syntax is as follows:

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

where clause can be any of the following:

`safelen(`*`length`*`)`	Maximum distance between two iterations executed concurrently by a SIMD instruction.
`linear(`*`list[:linear-step]`*`)`	List items are private and have a linear relationship with respect to the iteration space.
`aligned(`*`list[:alignment]`*`)`	List items are aligned to a platform-dependent value (or the value of the optional parameter).

`private(`*`list`*`)`, `lastprivate(`*`list`*`)`, `reduction(`*`reduction-identifier:list`*`)` and `collapse(`*`n`*`)` are also supported, with functionality matching that of omp for.

Are any of these clauses required here?

Build and re-run the application, and record the new run times below:

Runtime (Host):	
Runtime (MIC):	

How has the run time changed?

Although we have marked the *x* loop as one that can (and should) be converted to SIMD form, we see no speed-up. In fact, we may even see some slow-down! This is because the function call to "mandel()" is not vectorized, and will still be run in scalar – converting between vector code (which the compiler will use to load from cvals[] and store to res[]) and scalar code (which the compiler will use for the mandel function) introduces some overhead, and may decrease performance.

SIMD Functions

To ensure that everything vectorizes in the way that we want it to, we must also mark the mandel() function as a SIMD function. The syntax for this is below:

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

where clause can be any of the following:

`simdlen(`*`length`*`)`	Maximum number of concurrent arguments to the function (i.e. maximum SIMD width).
`uniform(`*`argument-list`*`)`	List items have the same value for all SIMD lanes, and can therefore be broadcast.
`inbranch` `notinbranch`	Function always called inside a conditional. Function never called inside a conditional.

`linear(`*`argument-list[:linear-step]`*`)` and `aligned(`*`argument-list[:alignment]`*`)` are also supported, with functionality matching that of "omp simd".

Are any of these clauses required here?

Build and re-run the application, and record the new run times below:

Runtime (Host):	
Runtime (MIC):	

How has the run time changed? Is this in line with your expectations?

Just because a code is vectorized does not necessarily mean that it is efficient – there may still be issues with its memory behaviour or long-latency math operations that act as a performance bottleneck. You should not be discouraged if your vector speed-up is less than the SIMD width, but investigate further to uncover the root cause.

Further Optimization

The Mandelbrot example used in this lab uses a Complex data type, with single precision. Some complex operations (such as cabsf) use higher intermediate precision by default.

We can improve the performance of cabsf by adding the following compiler flag to the command.sh scripts:

```
-complex-limited-range
```

Build and re-run the application, and record the new run times below:

Runtime (Host):	
Runtime (MIC):	

How has the runtime changed?

Note that this particular optimization was not required for the Intel® Xeon Phi™ coprocessor – it already uses a fast implementation of cabsf due to the IMF flags supplied to the compiler.

Homework: Complex Data Types

Assignment 1

Rather than a Complex data type, our Mandelbrot example could have used two separate floats to represent the real and imaginary parts of each number.

Make this change – how does it impact performance?

Assignment 2

Computing the absolute value of a complex number with cabsf() requires a sqrt() operation. These are expensive, and it is best to avoid them where possible.

Having re-implemented the Mandelbrot code using floats, is it possible to check for the break condition in mandel() without a sqrt() operation?

Make this change – how does it impact performance?