Explicit Vectorization with OpenMP* 4.0

This lab introduces explicit vectorization techniques, using the SIMD-focused features introduced in OpenMP 4.0. We follow a simple seven-step process:

1. Identify hotspots that can benefit from SIMD instructions.
2. Select a sufficiently wide data type, adjust floating-point accuracy, and consider mixed precision depending on algorithm phase.
3. Re-arrange data for SIMD efficiency.
4. Align data structures for SIMD efficiency.
5. Convert code to SIMD form.
6. Optimize memory access patterns.
7. Further (instruction-level) optimization.

This process can be applied to any code, and is not specific to OpenMP 4.0 – the conversion of the code to SIMD form could also make use of a compiler's auto-vectorization capabilities, array notation, SIMD intrinsics or inline assembly.

Setup

Building the Application

Begin by loading the Intel Compiler 15.0 Beta on Beacon:

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

Decide whether you would like to work with C++ or Fortran. The techniques covered in this lab are applicable to both, so you should choose whichever language is most familiar.

For C++ (you can omit the target "`c`"):

```
make c
```

For Fortran:

```
make fortran
```

Initially the application will be compiled without vectorization according to the flag `-no-vec` specified in the Makefile.

The optimization (and later vectorization) report will be redirected to the file `nbody.optrpt`. Please notice the new structure of the report!

In order to build for an Intel® Xeon Phi™ coprocessor, type:

```
make c_mic
make fortran_mic
```

Configuring and Running the Application

Please be patient when doing the first runs. Because of the included warm-up calculations and the deactivated vectorization the total execution time will be in the range of 1.5-2.5 minutes.

Host

Before running the application, be sure to set the OpenMP thread affinity as below. The optional `verbose` flag will show the binding of threads to logical processors:

```
export KMP_AFFINITY=compact,granularity=fine[,verbose]
```

Then, to execute the application, run:

```
./nbody 262144
```

Coprocessor

The easiest way to execute the application built for Xeon Phi™ on the coprocessor is to run the script which will be used also in later investigations with the Intel® VTune™ Amplifier XE:

```
./nbody_vtune.sh
```

Background (optional):

> Have a look into the script. The application is started from the host on the coprocessor by using the tool `micnativeloadex` which will also resolve all required libraries (`micnativeloadex -h` will show the options). The tool `micnativeloadex` will not use the last core on the coprocessor by default.
>
> If the NFS export is enabled for your current working directory and the compiler paths you can issue an alternative command using `ssh` to start the application from the host on the coprocessor. It will exploit all cores on Xeon Phi™ including the last one which often works well for running native codes as in the present lab. You will find the `ssh` command commented out in the script `nbody_vtune.sh` as well.
>
> The third alternative is to login with `ssh` directly onto the coprocessor and execute the application as in the Xeon case. Without NFS you have to copy the application and required libraries over before; with NFS you have to setup the environment, in particular the paths to the libraries.

Run the application now, and record the checksum and run time:

Checksum:	
Run time:	

Hotspots Analysis

Analysis with Intel® VTune™ Amplifier

Perform a hotspots analysis from the command-line:

Host

```
amplxe-cl -collect hotspots -r nbody_hs -- ./nbody 262144
```

Coprocessor

```
amplxe-cl -collect knc-hotspots -r nbody_hs -- ./nbody_vtune.sh
```

And examine the output:

```
amplxe-cl -report hotspots -r nbody_hs
```

Which function takes the most time? Record it below:

Hotspot:	

Selecting Floating-Point Precision

Many applications use double precision by default, even though single precision may be adequate. Our application has been configured to allow the floating-point precision to be changed at compile time, allowing us to investigate the impact of this decision on performance and accuracy. Locate the #define statements at the top of the main file, and change the precision from double to single:

For C++:

```
#define FPTYPE float
#define SQRT sqrtf
```

For Fortran:

```
#define FPTYPE real
#define SQRT sqrt
```

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

Checksum:	
Run time:	

Do the checksums match? How has the run time changed?

The impact of floating-point precision can be significant: double precision numbers are twice as large (in bytes) as single precision numbers, affecting both memory bandwidth requirements and the number of values that can be packed into SIMD units of a fixed bit-width. A speed-up of 2x is not unusual.

IMPORTANT:
Before changing the floating-point precision of your own applications, you should analyze the numerical stability of your algorithm and assess the correctness of this optimization.

Vectorization

Edit the `Makefile` and comment the variable holding the `-no-vec` flag. This will enable the auto-vectorization by the compiler. Re-build the application either for the host or the coprocessor. Check the vectorization report in `nbody.optrpt`.

Re-run the application now, and record the checksum and run time:

Checksum:	
Run time:	

How much did the vectorization improve the performance; is this what you expected?

Re-arrange Data for SIMD Efficiency

The code uses an Array-of-Structures (AoS) to store its position and acceleration data – that is, the Position and Acceleration arrays store the 3D quantities contiguously in memory, as shown below:

X	Y	Z	X	Y	Z	X	Y	Z	X	Y	Z

Do you think this layout is efficient for scalar execution? What about for SIMD execution?

To help with our investigation, our application produces copies of the input data arrays in both AoS and SoA formats (e.g. Position is split into Position_X, Position_Y and Position_Z). If the *j* loop in Perform_NBody() is vectorized, which form of the arrays will be better?

Change the references to Position and/or Acceleration in Perform_NBody(). If you do decide to make any changes to the Acceleration array, remember to update the Checking() function as well!

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

Checksum:	
Run time:	

How has the run time changed?

If the application runs slower, don't panic! The best data structure for scalar and SIMD execution are not always the same. As a rule of thumb:

Data Accessed by Scalar Code

- Use AoS to improve cache locality.

Data Accessed by SIMD Code with Linear Index

- SIMD lanes will access contiguous data elements.
- Use SoA (or AoSoA), to avoid gathers/scatters.

Data Accessed by SIMD Code via Indirection

- SIMD lanes may access non-contiguous data elements.
- Use a new algorithm (to avoid indirection).
  OR
- Use AoS to improve cache locality of gathers/scatters.

Applying this rule to our application (assuming that the *j* loop will be vectorized), only the layout of the Position array needs to be adjusted for SIMD efficiency – the Acceleration array is only accessed by scalar code, and can stay in AoS.

Align Data Structures

There are three steps to ensuring aligned data accesses in your SIMD applications:

1. Align memory.
2. Access memory in an aligned way.
3. Tell the compiler.

Align Memory

Look at the *j* loop in Perform_NBody(). Which arrays do you think need to be aligned?

When you have decided, update the definitions of these arrays to align them to a 64-byte boundary. We use a 64-byte boundary here because current-generation Intel® Xeon Phi™ coprocessors have a 512-bit SIMD width (512 / 8 = 64); 16- and 32-byte boundaries are sufficient when aligning arrays for the 128- and 256-bit instructions in SSE and AVX respectively.

For C++:

```
FPTYPE* array = (FPTYPE*)_mm_malloc(elements * sizeof(FPTYPE), 64);
```

For Fortran:

```
FPTYPE, allocatable :: array(:)
!dir$ attributes align:64 :: array
```

Access Memory in an Aligned Way

An array will be "accessed in an aligned way" by a loop if:

- The base address of the array is appropriately aligned; and
- Every index used to access the array is a multiple of the SIMD width.

Is this true of the *j* loop in Perform_NBody()?

Tell the Compiler

Even when an array has been aligned using a function call or attribute, the compiler is still often unable to determine whether all accesses to that array will be aligned.

For best results, the programmer should tell the compiler all accesses are aligned when converting the code to SIMD form, as discussed in the next section.

Convert Code to SIMD Form

We are finally ready to convert our application to SIMD form. The OpenMP* 4.0 standard supports two different ways to do this:

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

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

The first of these methods is most applicable here, so go ahead and mark the *j* loop in Perform_NBody() as being a SIMD loop. As a reminder, the syntax is as follows:

For C:

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

For Fortran:

```
!$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.

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

Checksum:	
Run time:	

Do the checksums match? How has the run time changed?

If your code produces different answers to what you expected, then you may have forgotten a clause. Repeat this exercise until you are happy with the checksum.

Hint: Make sure that your variables are declared as private/reduction where appropriate.

Optimize Memory Access Patterns

So far, this lab has focused upon converting code to SIMD form (and doing so efficiently). The application speed-up we have achieved is (hopefully!) impressive, but we should not stop here – we can tune and optimize SIMD code in exactly the same way as we can tune and optimize scalar code.

Consider the memory-access pattern of our N-body application. Since we use a naïve all-pairs O(N^2) implementation, each body *i* is interacted with each body *j* and, for sufficiently large values of N, there are too many bodies to store in cache.

Cache-blocking is a well-known optimization that targets situations such as these, and remains applicable for SIMD codes. Block the *j* loop in Perform_NBody(), following a similar code structure to that below:

```
#define CHUNK_SIZE 8192
#pragma omp parallel
for (int jj = 0; jj < number_of_bodies; jj += CHUNK_SIZE)
{
      #pragma omp for
      for (int i = 0; i < number_of_bodies; i++)
      {
            #pragma omp simd …
            for (int j = jj; j < jj + CHUNK_SIZE; j++)
            {
                  …
            }
      }
}
```

After applying this optimization, the code is able to keep a number of bodies (CHUNK_SIZE) in cache, and re-use this data for multiple iterations of the *i* loop.

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

Checksum:	
Run time:	

How has the runtime changed? Play with the block size until you are happy that you are seeing a performance improvement – the default size may not be optimal on all systems.

That cache blocking improves performance here highlights the importance of designing SIMD-friendly algorithms and converting code to SIMD form *before* spending time exploring other optimization opportunities. Codes that are compute-bound in scalar may be memory-bound in SIMD, and design decisions that are beneficial to scalar code may even be detrimental to SIMD code.

Further Optimization

In applications where high-latency math operations are a performance bottleneck, significant performance gains can be realised by reducing the accuracy of the Intel® math library.

To reduce the accuracy of the divide and sqrt operations in our N-body code, add the following two flags to the Makefile:

```
-fimf-domain-exclusion=15 -fimf-accuracy-bits=12
```

The first of these flags informs the compiler that we do not require math functions to return correct results for "uncommon" inputs (i.e., extremes, NaNs, infinities and denormals), and the second informs the compiler exactly how much precision we need for the remaining "common" inputs. The reader is referred to the Intel® Compiler documentation for an in-depth explanation of the operation of these flags.

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

Checksum:	
Run time:	

How has the runtime changed?

People often fear that adjusting FP accuracy (as suggested above) is beyond what they want. Two examples: (1) processing de-normalized FP numbers (rather than flushing them to zero) is often not supported in "real hardware" or may be unsupported on some non-x86 architectures, and (2) consider calculating EXP(800) – did you know that the result is out of range with respect to double-precision? Therefore, it makes sense to eliminate such kind of input from the processing pipeline rather than processing it (normalized numbers, etc.).

Homework: Outer-loop Vectorization, Array Notation and SIMD Intrinsics

Assignment 1

OpenMP 4.0 supports the application of its simd pragmas to outer- as well as inner-loops. Move the pragma from the *j* loop to the *i* loop, paying close attention to the clauses you supply.

How does this affect performance? Why?

Assignment 2

Re-write the *j* loop using array notation.

Assignment 3

Re-write the *j* loop using SIMD intrinsics. You may find the Intel® Intrinsics Guide helpful: http://software.intel.com/sites/landingpage/IntrinsicsGuide/