Vectorization for Intel® C++ & Fortran Compiler (Linux*)

_____

Lab

Disclaimer

The information contained in this document is provided for informational purposes only and represents the current view of Intel Corporation ("Intel") and its contributors ("Contributors") on, as of the date of publication. Intel and the Contributors make no commitment to update the information contained in this document, and Intel reserves the right to make changes at any time, without notice.

DISCLAIMER. THIS DOCUMENT, IS PROVIDED "AS IS." NEITHER INTEL, NOR THE CONTRIBUTORS MAKE ANY REPRESENTATIONS OF ANY KIND WITH RESPECT TO PRODUCTS REFERENCED HEREIN, WHETHER SUCH PRODUCTS ARE THOSE OF INTEL, THE CONTRIBUTORS, OR THIRD PARTIES. INTEL, AND ITS CONTRIBUTORS EXPRESSLY DISCLAIM ANY AND ALL WARRANTIES, IMPLIED OR EXPRESS, INCLUDING WITHOUT LIMITATION, ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR ANY PARTICULAR PURPOSE, NON-INFRINGEMENT, AND ANY WARRANTY ARISING OUT OF THE INFORMATION CONTAINED HEREIN, INCLUDING WITHOUT LIMITATION, ANY PRODUCTS, SPECIFICATIONS, OR OTHER MATERIALS REFERENCED HEREIN. INTEL, AND ITS CONTRIBUTORS DO NOT WARRANT THAT THIS DOCUMENT IS FREE FROM ERRORS, OR THAT ANY PRODUCTS OR OTHER TECHNOLOGY DEVELOPED IN CONFORMANCE WITH THIS DOCUMENT WILL PERFORM IN THE INTENDED MANNER, OR WILL BE FREE FROM INFRINGEMENT OF THIRD PARTY PROPRIETARY RIGHTS, AND INTEL, AND ITS CONTRIBUTORS DISCLAIM ALL LIABILITY THEREFOR. INTEL, AND ITS CONTRIBUTORS DO NOT WARRANT THAT ANY PRODUCT REFERENCED HEREIN OR ANY PRODUCT OR TECHNOLOGY DEVELOPED IN RELIANCE UPON THIS DOCUMENT, IN WHOLE OR IN PART, WILL BE SUFFICIENT, ACCURATE, RELIABLE, COMPLETE, FREE FROM DEFECTS OR SAFE FOR ITS INTENDED PURPOSE, AND HEREBY DISCLAIM ALL LIABILITIES THEREFOR. ANY PERSON MAKING, USING OR SELLING SUCH PRODUCT OR TECHNOLOGY DOES SO AT HIS OR HER OWN RISK.

Licenses may be required. Intel, its contributors and others may have patents or pending patent applications, trademarks, copyrights or other intellectual proprietary rights covering subject matter contained or described in this document. No license, express, implied, by estoppels or otherwise, to any intellectual property rights of Intel or any other party is granted herein. It is your responsibility to seek licenses for such intellectual property rights from Intel and others where appropriate. Limited License Grant. Intel hereby grants you a limited copyright license to copy this document for your use and internal distribution only. You may not distribute this document externally, in whole or in part, to any other person or entity. LIMITED LIABILITY. IN NO EVENT SHALL INTEL, OR ITS CONTRIBUTORS HAVE ANY LIABILITY TO YOU OR TO ANY OTHER THIRD PARTY, FOR ANY LOST PROFITS, LOST DATA, LOSS OF USE OR COSTS OF PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES, OR FOR ANY DIRECT, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF YOUR USE OF THIS DOCUMENT OR RELIANCE UPON THE INFORMATION CONTAINED HEREIN, UNDER ANY CAUSE OF ACTION OR THEORY OF LIABILITY, AND IRRESPECTIVE OF WHETHER INTEL, OR ANY CONTRIBUTOR HAS ADVANCE NOTICE OF THE POSSIBILITY OF SUCH DAMAGES. THESE LIMITATIONS SHALL APPLY NOTWITHSTANDING THE FAILURE OF THE ESSENTIAL PURPOSE OF ANY LIMITED REMEDY.

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

See: http://software.intel.com/en-us/articles/optimization-notice/

Requirements	**Intel® oneAPI HPC Toolkit** with **Intel® C++/Fortran Compiler Classic** **Linux*** host OS supported by Intel® C++/Fortran Compiler Classic At least **2nd generation Intel® Core™ processor** (with Intel® AVX). For stable measurements it is recommended to turn off Intel® Hyper-Threading, Intel SpeedStep® and Intel® Turbo Boost if available.
Objective	In this lab session, you will use the Intel® C++/Fortran Compiler Classic to become familiar with vectorization. After successfully completing this lab's activities, you will be able to: • To select the right instruction set extension switch for compiling • Apply and analyze vectorization • Deal with memory aliasing and alignment issues potentially preventing vectorization • Improve vectorization by using the pragma IVDEP • Understand Intel specific techniques for vectorization

Notes:

• In these instructions we will use some switches like **/O2** explicitly when invoking the compiler although these switches might be set by default already. Doing it this explicit way will make it more obvious what we are doing.

• Older product suites are also supported.

• Processors only supporting SSE (pre 2nd generation Intel® Core™ processor) can be used as well but differences in terms of performance advantages might be smaller due to smaller vector length. On 2nd generation Intel® Core™ processors and later, SSE can be used instead of AVX to compare results in terms of different vector lengths.

• Time (for each, C/C++ & Fortran):
Activity 1: 45 min
Activity 2: 15 min (optional)

Preface

The following activities can be used for 2nd generation Intel® Core™ processor and later (with Intel® AVX/AVX-512 support, Intel® Xeon Phi™ processor). Please regard the following for the different targets:

- **2nd generation Intel® Core™ processor and later:**

  - As **$SIMD** use **–xavx** (alternatively you can also use **–mavx** or **–axavx**)

  - Please keep in mind: An AVX **vector contains 4 double precision FP elements** or 8 single precision FP elements. The activities below use double precision FP if not noted otherwise, but can be also changed to single precision FP.

- **Intel® Xeon® Processor Scalable Family (Skylake Server or later):**

  - As **$SIMD** use **–xCORE-AVX512**

  - Please keep in mind: An AVX-512 **vector contains 8 double precision FP elements** or 16 single precision FP elements if 512 bit zmm registers are used. By default the compiler avoid zmm code generation with 18.0 compiler version for SKX with –xCORE-AVX512 option and generate AVX-512 instructions with 256 bit ymm registers in most cases. Use -qopt-zmm-usage=high to force the compiler generate zmm code.

Important note:

Please don't forget to set the environment via script shipped with oneAPI toolkit to get access to the compiler and libraries. All exercises require that if not noted otherwise below or by the instructor. For default installation use the following command:

```
$ source /intel/oneapi/setvars.sh
```

On some machines the environment to use Intel tools may be present or may require the usage of module commands. Please check with your system administrator required steps to set the environment properly.

C/C++

Activity 1.1 – Why Vectorization Failed

In this activity, you enable the compiler to generate diagnostic information on sample code that does not vectorize initially but can be vectorized, as will be seen in the very end. We start with a first analysis of what is keeping the compiler from vectorization.

Download sources using following command:

git clone https://github.com/ivorobts/compiler-optimization.git

1. Navigate to the `matvector/c` folder. This example calculates the product of a matrix and a vector. It is simple enough to comprehend but yet contains the most important stumbling stones in terms of vectorization we're going to solve throughout the following lab activities.

2. Initially, we compile our example without any vectorization carried out by the compiler. We use the `-no-vec` switch to turn off vectorization that's implicitly activated with optimization levels of 2 and higher (note that level 2 [`-O2`] is already the default):

    ```
    $ icc -O2 $SIMD -no-vec multiply.c driver.c -o matvector
    $ ./matvector
    ```

    Record execution time _____.

3. Next, we enable vectorization:

    ```
    $ icc -O2 $SIMD multiply.c driver.c -o matvector
    $ ./matvector
    ```

    Record execution time _____.

4. The source file `multiply.c` contains the computation we're measuring. Please take a closer look at it. Before addressing vectorization, it is crucial to understand that the underlying algorithms can take benefit from it. In the given example, we're using a pattern that symbolically acts as a first (algorithmic!) stumbling stone to vectorization. Which is it? Can it be fixed and how?

    Hint: Non-unit stride access
    Solution: `solutions/unit-stride`

    5.   The above example is just one of many kinds of algorithmic problems hindering vectorization. They typically occur if the design of the algorithm either does not allow combined computation of a (consecutive) set of data and/or if the algorithm contains (too many) parameters that only can be resolved at runtime. The above example (trivially!) used the latter, to demonstrate the effects. Your own algorithms for sure require different approaches, which are rather a subject of research then a generic guideline. It is important to solve any such fundamental stumbling stones first before continuing with the next steps. When applying a possible solution to the above problem we notice an improvement of the performance.

    Record execution time _____.

    We use the optimization reports feature from the compiler to provide more information on vectorization details:

```
$ icc -O2 $SIMD multiply.c driver.c -o matvector -qopt-report=5
      -qopt-report-file=stdout
...
```

This will provide additional information which loops have been vectorized and how, including information why vectorization was not possible. Find out which loops in **driver.c** and **multiply.c** vectorized and which loops didn't vectorize.

Note that the inner loop in **multiply.c** was multiversioned. Try to add extra argument to matvec function and use it in the loop. Recompile and check the optimization report. Is the compiler still able to multiversion this loop?

Hint: Adding extra array to the loop prevent the compiler to do a multiversioning.
Solution: **solutions\multiversioning_off**

The compiler can multiversion the loop in trivial cases. There is no mutiversioning in case we add extra function argument and use it in the loop. We will use this modified test case with extra argument further in order to learn solutions we may apply in case multiversioning not used by compiler.

Activity 1.2 – Vectorization of Inner Loop

In this activity, you will make huge progress with vectorization by remedying the reported problems from the optimization reports.

1. Once more, look at the source code of `multiply.c`. With closer analysis we notice that the reported dependencies are not applicable for the semantics of our example. The compiler needs additional information to recognize the assumed dependencies (arrays are allocated in another compilation unit) as false dependencies. You have learned about multiple ways to do so:

    - `-fargument-noalias` option and
    - `#pragma ivdep`
    - keyword `restrict`

    All of those can help to break up the assumed dependencies. This results in better vectorization for our example. In the following, use the last version from the previous activity as basis.

2. First, we start by turning off aliasing for all function arguments throughout a whole compilation unit. This is meaningful if we can guarantee that no pointers in all the function arguments overlap (even if they're of the same type, which still is allowed by strict ANSI aliasing). For our example this is the case because all arrays have disjunctive memory locations. Hence, we can compile the code as is without any modifications, only by adding one single compiler option:

    ```
    $ icc -O2 $SIMD multiply.c driver.c -o matvector -fargument-noalias
    $ ./matvector
    ```

    Record execution time _____.

    Take a look at the vectorization reports for verification.

3. Instead of globally addressing assumed dependencies, we can also break them up on a per-loop basis by using `#pragma ivdep`. Apply it to the proper loop in the code, compare the performance and consult the optimization reports.

    This time, no additional compiler options are needed:

    ```
    $ icc -O2 $SIMD multiply.c driver.c -o matvector
    $ ./matvector
    ```

    Record execution time _____.

    Hint: It's only needed for one loop
    Solution: `solutions/ivdep`

4. Finally, we try yet another approach which is even finer grained: We apply the keyword `restrict`. Can you find the correct location where to apply it to?
    Be aware that we either have to assert C99 conforming code (`-std=c99`) or apply the compiler option `-restrict`. The latter is compiler implementation specific and not standardized. However, most compilers allow that for pre-C99 or even C++ (only for pointers, not references).

    When applied correctly the effective performance should be similar to the previous two solutions:

```
$ icc -O2 $SIMD multiply.c driver.c -o matvector -restrict
$ ./matvector
```

Record execution time _____.

Consult the optimization reports once more for validation.

Hint: It's only needed for one function argument
Solution: **solutions/restrict**

5. As preparation for the next activity, select any of the three solutions from this activity above and generate the assembly (**-S**) of **multiply.c**. You can see that the compiler generated multiple versions and tests right in the beginning of the function. Those are caused by unknown alignment of the array elements. In the following activity we continue with the solution for **#pragma ivdep** but the others work as well, provided that the required compiler options are specified in addition.

Activity 1.3 – Alignment Improvements

In this activity, you will improve the performance of the code generated by asserting alignment of data.

1. The only data that is used throughout the loops are arrays `a`, `b` and `x`. Looking at their allocation in `driver.c`, it turns out that the arrays are not necessarily aligned. By using a simple attribute you can guarantee alignment of all three arrays. Which one would be best here? Change the code accordingly.

   The execution time won't change yet:

   ```
   $ icc -O2 $SIMD multiply.c driver.c -o matvector
   $ ./matvector
   ```

   Record execution time _____.

   Hint: `__declspcec(align(…))` or `__attribute__((aligned(…)))`
   Solution: `solutions/align`

   This change was mostly a preparation for the next step, where we are going to use that alignment.

2. Next, we look into `multiply.c`. As it is compiled separately (like any compilation unit) it does not have knowledge about the alignment. Thus it has to assume unaligned data accesses. The compiler can generate multiple versions (e.g. for aligned and unaligned access) and select the proper execution path during run-time. This involves some overhead (and increases code size). Since we guaranteed proper alignment for the memory locations, to which all the pointers in the function arguments refer to, we can safely assert this for the compiler. There is a simple way to do so, which one?

   The execution time once more is reduced:

   ```
   $ icc -O2 $SIMD multiply.c driver.c -o matvector
   $ ./matvector
   ```

   Record execution time _____.

   Hint: `__assume_aligned(…)`
   Solution: `solutions/assume_aligned`

3. Moving our focus away from sole compiler optimizations there is more room for improvement:
   The amount of elements per row is not multiple of what a SIMD vector (or multiple thereof) can keep. Hence each row needs remainder handling. This remainder handling causes additional overhead at the end of each row. To avoid this overhead, pad the row to a multiple of the SIMD vector length. Padding is controlled via `COLBUF` in our example. Understand how it works and apply a correct value to it so it is done correctly. Once `COLBUF` is set correctly another improvement should become visible:

   ```
   $ icc -O2 $SIMD multiply.c driver.c -o matvector
   $ ./matvector
   ```

   Record execution time _____.

   Note:
   We increased the size of the matrix (2-dimensional array) and yet the performance became much better!

   Solution: `solutions/padding`

4. Please ensure that all elements per row are now multiple of the SIMD vector length. This allows a more aggressive optimization regarding alignment: **#pragma vector aligned**
In addition we can tell the compiler that the rows are multiple of at least the vector length by using **__assume(<variable> % <multiple_vl> == 0)**. The value of **<multiple_vl>** can be any multiple of the vector length.
Apply both to correct locations in **multiply.c** and measure the execution time once more:

```
$ icc -O2 $SIMD multiply.c driver.c -o matvector
$ ./matvector
```

Record execution time _____.

Hint: Only one loop needs the pragma; assertion of elements per row should be as large as possible
Solution: **solutions/vector_aligned**

Note:
Using the pragma and enforcing aligned accesses, unconditionally applies to all accesses inside the loop. If we had <u>not</u> padded the arrays before and used the pragma, the compiler would create incorrect code. The reason is that the first row in the array **a** starts at a vector aligned address but the second one does not (elements per row are not multiple of the vector length).
We won't always see a crash for our example because the compiler tends to use unaligned moves which work for both aligned and unaligned data (not for the coprocessor!). 2nd generation Intel® Core™ processors and later can figure out actual alignment during run-time and, for the case of actually aligned data, use the much faster aligned accesses internally instead. However, there is no guarantee for that and it has to be expected to face SEGVs when the pragma is used incorrectly. The SEGVs result from the GP faults of instructions that require aligned data but have been provided with unaligned memory references.

Note:
When using the official solution, you will notice that **__assume(cols % 64 == 0)** is used. This is not a multiple of 64 byte but a multiple of 64 elements of double precision FP and hence 64 * 8 byte = 512 byte per row. This is also a multiple of the possible vector lengths. The larger the guaranteed elements per row can be asserted, the more flexibility is granted for the compiler. You could also try to just use the vector length itself. Then you might notice good but still not optimal results.

Congratulation! You've now reached the optimal version of this exercise. In the following activity you are going to use an alternative technique to vectorize the same code. The results are the same but this technique might be more applicable for more complex code.

Activity 2 – Using Inter-Procedural Optimization

In this activity you will see that inter-procedural optimization (IPO) can improve vectorization considerably. When crucial information, required for vectorization, is scattered across different compilation units the compiler usually cannot retrieve it because each unit is compiled independently. IPO, more precisely multi-file IPO, is a way to address this problem.
In the following we're using a different application that has multiple compilation units to demonstrate the problem much better. The used application computes electrostatic potential due to a uniform distribution over a square.

1. Navigate to the `square_charge/c` folder.

2. First, compile the whole application without multi-file IPO:

   ```
   $ icc –O3 $SIMD rabs.c square_charge.c trap.c twod.c –o sq
   $ ./sq
   ```

   Record execution time _____.

3. Now, compile the whole application with multi-file IPO turned on:

   ```
   $ icc –O3 $SIMD rabs.c square_charge.c trap.c twod.c –o sq –ipo
   $ ./sq
   ```

   Record execution time _____.

4. Compare both versions and understand how the compiler was able to optimize.
   Hint: Use optimization reports via `–opt-report`, esp. `–opt-report-phase=ipo`

   Optionally:
   How different are the executables for the two versions that the compiler has generated?
   Hint: Take a look at the assembly output [`–S`].

Note:
In this activity you might see that the values of the "Potentials" are slightly different. The reason is that the FP computations are done with the "fast" FP model (compiler default), which can make best use of optimization, including vectorization. For highly accurate algorithms, however, it is required to change the FP model towards a stricter IEEE 754 interpretation (`–fp-model precise`, `–fp-model strict`, etc.). The downside of this is that optimization, and hence vectorization, becomes limited (e.g. by accuracy, order of operations, etc.). Even slower x87 instructions might be used (`–fp-model strict`). The optimization reports tell which loops were not vectorized because of the selected FP model.
As a compromise between speed and numerical stability (accuracy & reproducibility) we recommend to use the option set `–fp-model precise –fp-model source` instead of the strictest IEEE 754 interpretation via `–fp-model strict`.

Fortran

Activity 1.1 – Why Vectorization Failed

In this activity, you enable the compiler to generate diagnostic information on sample code that does not vectorize initially but can be vectorized, as will be seen in the very end. We start with a first analysis of what is keeping the compiler from vectorization.

Download sources using following command:

git clone https://github.com/ivorobts/compiler-optimization.git

1. Navigate to the **matvector/fortran** folder. This example calculates the product of a matrix and a vector. It is simple enough to comprehend but yet contains the most important stumbling stones in terms of vectorization we're going to solve throughout the following lab activities.

2. Initially, we compile our example without any vectorization carried out by the compiler. We use the **-no-vec** switch to turn off vectorization that's implicitly activated with optimization levels of 2 and higher (note that level 2 [**–O2**] is already the default):

   ```
   $ ifort -fpp –O2 $SIMD –no-vec driver.f90 multiply.f90 -o matvector
   $ ./matvector
   ```

   Record execution time _____.

3. Next, we enable vectorization:

   ```
   $ ifort -fpp –O2 $SIMD driver.f90 multiply.f90 -o matvector
   $ ./matvector
   ```

   Record execution time _____.

4. The source file **multiply.f90** contains the computation we're measuring. Please take a closer look at it. Before addressing vectorization, it is crucial to understand that the underlying algorithms can take benefit from it. In the given example, we're using a pattern that symbolically acts as a first (algorithmic!) stumbling stone to vectorization. Which is it? Can it be fixed and how?

   Hint: Non-unit stride access
   Solution: **solutions/unit-stride**

5. The above example is just one of many kinds of algorithmic problems hindering vectorization. They typically occur if the design of the algorithm either does not allow combined computation of a (consecutive) set of data and/or if the algorithm contains (too many) parameters that only can be resolved at runtime. The above example (trivially!) used the latter, to demonstrate the effects. Your own algorithms for sure require different approaches, which are rather a subject of research then a generic guideline. It is important to solve any such fundamental stumbling stones first before continuing with the next steps. When applying a possible solution to the above problem we notice an improvement of the performance.

Record execution time _____.

However, vectorization is still not optimal. We use the optimization reports feature from the compiler to provide more information on why:

```
$ ifort –fpp -O2 $SIMD driver.f90 multiply.f90 -o matvector –qopt-report=5 ↵
        –qopt-report-file=stdout
...
```

This will provide additional information which loops have been vectorized and how, including information why vectorization was not possible. Find out which loops in **driver.f90** and **multiply.f90** vectorized and which loops didn't vectorize.

It seems that the inner loop found in **multiply.f90** has cases of "vector dependence" which limit vectorization. Try to understand which dependencies there are.

Activity 1.2 – Vectorization of Inner Loop

In this activity, you will make huge progress with vectorization by remedying the reported problems from the optimization reports.

1. Once more, look at the source code of **multiply.f90**. With closer analysis we notice that the reported dependencies are not applicable for the semantics of our example. The compiler needs additional information to recognize the assumed dependencies (arrays are allocated in another compilation unit) as false dependencies. You have learned about multiple ways to do so:

   - **-fno-fnalias** option and
   - **!DIR$ IVDEP**
   - keyword **contiguous**

   All of those can help to break up the assumed dependencies or avoid unnecessary inherent overhead of Fortran array subset handling (non-contiguous subset with non-unit stride). This results in better vectorization for our example. In the following, use the last version from the previous activity as basis.

2. First, we start by turning off aliasing for all functions throughout a whole compilation unit. This is meaningful if we can guarantee that no pointers in all the functions overlap. For our example this is the case because all arrays have disjunctive memory locations. Hence, we can compile the code as is without any modifications, only by adding one single compiler option:

   ```
   $ ifort –fpp -O2 $SIMD driver.f90 multiply.f90 -o matvector –fno-fnalias
   $ ./matvector
   ```

   Record execution time _____.

   Take a look at the vectorization reports for verification. The global alternative **-fno-alias** could also be used (but with more care!).

3. Instead of globally addressing assumed dependencies, we can also break them up on a per-loop basis by using **!DIR$ IVDEP**. Apply it to the proper loop in the code, compare the performance and consult the optimization reports.

   This time, no additional compiler options are needed:

   ```
   $ ifort –fpp -O2 $SIMD driver.f90 multiply.f90 -o matvector
   $ ./matvector
   ```

   Record execution time _____.

   Hint: It's only needed for one loop
   Solution: **solutions/ivdep**

4. Finally, we can add another optimization to the two previous versions by using an additional keyword. It tells the compiler that all array elements are compact in memory with unit stride. The Fortran standard allows creating subsets of arrays on the fly which can even have non-unit stride. As we learned earlier, non-unit stride access is bad in terms of performance. For our case we also don't need this "flexibility" of the language. Hence, we can tell the compiler that an array is expected to have unit stride by adding the **contiguous** keyword.

   When applied correctly to either of the two previous versions, the effective performance should increase another time. We're using the **!DIR$ IVDEP** version, but you can also try the **–fno-fnalias** version:

```
$ ifort —fpp -O2 $SIMD driver.f90 multiply.f90 -o matvector
$ ./matvector
```

Record execution time _____.

Consult the optimization reports once more for validation.

Hint: It's only needed for one function argument
Solution: **solutions/contiguous**

5. For Fortran, control of aligning targets of pointers is limited. To show a more detailed tuning of alignment, we're going to use a version with passing arrays directly instead of pointers to arrays. Some of the techniques can also be applied to the pointer version used previously, though.
    As a positive side-effect of the array version we also get rid of assumed data dependencies. All the other aspects learned above still apply for it. Please use the version found in **solutions/array_version** for the following activities. Compile it and verify it's performance:

```
$ ifort —fpp -O2 $SIMD driver.f90 multiply.f90 -o matvector
$ ./matvector
```

Record execution time _____.

6. Generate the assembly (**-S**) of **multiply.f90**. You can see that the compiler generated multiple versions and tests right in the beginning of the function. Those are caused by unknown alignment of the array elements.

Activity 1.3 – Alignment Improvements

In this activity, you will improve the performance of the code generated by asserting alignment of data.

1. The only data that is used throughout the loops are arrays **a**, **b** and **x**. Looking at their allocation in **driver.f90**, it turns out that the arrays are not necessarily aligned. By using a simple attribute you can guarantee alignment of all three arrays. Which one would be best here? Change the code accordingly.

   The execution time won't change yet:

   ```
   $ ifort -fpp -O2 $SIMD driver.f90 multiply.f90 -o matvector
   $ ./matvector
   ```

   Record execution time _____.

   Hint: **!DIR$ ATTRIBUTES ALIGN:**
   Solution: **solutions/align**

   This change was mostly a preparation for the next step, where we are going to use that alignment.

2. Next, we look into **multiply.f90**. As it is compiled separately (like any compilation unit) it does not have knowledge about the alignment. Thus it has to assume unaligned data accesses. The compiler can generate multiple versions (e.g. for aligned and unaligned access) and select the proper execution path during run-time. This involves some overhead (and increases code size). Since we guaranteed proper alignment for the arrays in the function arguments we can safely assert this for the compiler. There is a simple way to do so, which one?

   The execution time once more is reduced:

   ```
   $ ifort -fpp -O2 $SIMD driver.f90 multiply.f90 -o matvector
   $ ./matvector
   ```

   Record execution time _____.

   Hint: **!DIR$ ASSUME_ALIGNED**
   Solution: **solutions/assume_aligned**

3. Moving our focus away from sole compiler optimizations there is more room for improvement:
   The amount of elements per column is not multiple of what a SIMD vector (or multiple thereof) can keep. Hence each column needs remainder handling. This remainder handling causes additional overhead at the end of each column. To avoid this overhead, pad the column to a multiple of the SIMD vector length. Padding is controlled via **ROWBUF** in our example. Understand how it works and apply a correct value to it so it is done correctly. Once **ROWBUF** is set correctly another improvement should become visible:

   ```
   $ ifort -fpp -O2 $SIMD driver.f90 multiply.f90 -o matvector
   $ ./matvector
   ```

   Record execution time _____.

   Note:
   We increased the size of the matrix (2-dimensional array) and yet the performance became much better!

   Solution: **solutions/padding**

4. Please ensure that all elements per column are now multiple of the SIMD vector length. This allows a more aggressive optimization regarding alignment: `!DIR$ VECTOR ALIGNED`
In addition we can tell the compiler that the columns are multiple of at least the vector length by using `!DIR$ ASSUME (MOD(<variable>, <multiple_vl>) .EQ. 0)`. The value of `<multiple_vl>` can be any multiple of the vector length.
Apply both to correct locations in `multiply.c` and measure the execution time once more:

```
$ ifort -fpp -O2 $SIMD driver.f90 multiply.f90 -o matvector
$ ./matvector
```

Record execution time _____ .

Hint: Only one loop needs that directive; assertion of elements per column should be as large as possible
Solution: `solutions/vector_aligned`

Note:
Using this directive and enforcing aligned accesses, unconditionally applies to all accesses inside the loop. If we had <u>not</u> padded the arrays before and used the directive, the compiler would create incorrect code. The reason is that the first column in the array `a` starts at a vector aligned address but the second one does not (elements per column are not multiple of the vector length).
We won't always see a crash for our example because the compiler tends to use unaligned moves which work for both aligned and unaligned data (not for the coprocessor!). 2nd generation Intel® Core™ processors and later can figure out actual alignment during run-time and, for the case of actually aligned data, use the much faster aligned accesses internally instead. However, there is no guarantee for that and it has to be expected to face SEGVs when the directive is used incorrectly. The SEGVs result from the GP faults of instructions that require aligned data but have been provided with unaligned memory references.

Note:
When using the official solution, you will notice that `!DIR$ ASSUME (MOD(rows, 64) .EQ. 0)` is used. This is not a multiple of 64 byte but a multiple of 64 elements of double precision FP and hence 64 * 8 byte = 512 byte per column. This is also a multiple of the possible vector lengths. The larger the guaranteed elements per column can be asserted, the more flexibility is granted for the compiler. You could also try to just use the vector length itself. Then you might notice good but still not optimal results.

Congratulation! You've now reached the optimal version of this exercise. In the fowling two activities you are going to use two alternative techniques to vectorize the same code. The results are the same but those techniques might be more applicable for more complex code.

1.4 Fortran Array Notation

In this activity, you will apply Fortran array notation to the original example.

1. Take the original version from activity 1.1 and change the code in `multiply.f90` to use Fortran array notation.

   ```
   $ ifort –fpp -O2 $SIMD driver.f90 multiply.f90 -o matvector
   $ ./matvector
   ```

   Record execution time _____.

   Hint: Transform inner loop; don't forget to take the non-unit stride into account (not necessary though)
   Solution: `solutions/arraynotation`

2. Now re-apply the steps from activity 1.1 to 1.3 to the current version. What is the best combination?

   Record execution time _____.

   Note:
   For the example we use throughout this activity there should be almost no difference compared to the auto-vectorized version. Fortran array notation might become more powerful for innermost loops. However, there can be side effects because of temporaries created or if used across outer loops.

   Hint: Apply the same steps as in previous activities. Not all might be needed, though.
   Solution: `solutions/arraynotation_best`

Activity 2 – Using Inter-Procedural Optimization

In this activity you will see that inter-procedural optimization (IPO) can improve vectorization considerably. When crucial information, required for vectorization, is scattered across different compilation units the compiler usually cannot retrieve it because each unit is compiled independently. IPO, more precisely multi-file IPO, is a way to address this problem.
In the following we're using a different application that has multiple compilation units to demonstrate the problem much better. The used application computes electrostatic potential due to a uniform distribution over a square.

1. Navigate to the **`square_charge/fortran`** folder.

2. First, compile the whole application without multi-file IPO:

   ```
   $ ifort -fpp –O3 $SIMD rabs.f90 square_charge.f90 trap.f90 twod.f90 –o sq
   $ ./sq
   ```

   Record execution time _____.

3. Now, compile the whole application with multi-file IPO turned on:

   ```
   $ ifort -fpp –O3 $SIMD rabs.f90 square_charge.f90 trap.f90 twod.f90 –o sq –ipo
   $ ./sq
   ```

   Record execution time _____.

4. Compare both versions and understand how the compiler was able to optimize.
   Hint: Use optimization reports via **`–qopt-report`**, esp. **`–qopt-report-phase=ipo`**

   Optionally:
   How different are the executables for the two versions that the compiler has generated?
   Hint: Take a look at the assembly output [**`–S`**].

Note:
In this activity you might see that the values of the "Potentials" are slightly different. The reason is that the FP computations are done with the "fast" FP model (compiler default), which can make best use of optimization, including vectorization. For highly accurate algorithms, however, it is required to change the FP model towards a stricter IEEE 754 interpretation (**`–fp-model precise`**, **`–fp-model strict`**, etc.). The downside of this is that optimization, and hence vectorization, becomes limited (e.g. by accuracy, order of operations, etc.). Even slower x87 instructions might be used (**`-fp-model strict`**). The optimization reports tell which loops were not vectorized because of the selected FP model.
As a compromise between speed and numerical stability (accuracy & reproducibility) we recommend to use the option set **`–fp-model precise`** instead of the strictest IEEE 754 interpretation via **`–fp-model strict`**.