

Vectorization using Intel® Composer XE 2013 for 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/

Intel and Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

*Other names and brands may be claimed as the property of others.

Copyright © 2013, Intel Corporation. All Rights Reserved.

Requirements	Intel® Composer XE 2013 Initial Release (following updates might show other results) At least 2 nd generation Intel® Core™ processor (with Intel® AVX) For stable measurement it is recommended to turn off Intel® Hyper-Threading, Intel SpeedStep® and Intel® Turbo Boost
Objective	In this lab session, you will use Intel® Composer XE 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 Generate and analyze vectorization and optimization reports Deal with memory aliasing and alignment issues potentially preventing vectorization Understand the benefit of inter-procedural optimization for best vectorization results To improve vectorization by using the pragma/directive SIMD To use Guided Auto Parallelism (GAP) – in particular 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.
- Throughout the exercises we use strict ANSI conforming code. Hence we use the compiler option <code>-ansi-alias</code> to turn on strict ANSI aliasing. This option is highly recommended for any ANSI conforming code. It won't be set by default for Intel C/C++ compilers because of historical reasons.

For the Fortran compiler, -ansi-alias is the default.

- Time (C/C++ & Fortran separate):
 - Activity 1: 45 min (60 min recommended)
 - Activity 2: 15 min
 - Activity 3: 30 min (45 min recommended)
 - Activity 4: 15 min

C/C++

Activity 1.1 – Why Vectorization Failed

In this activity, you enable the compiler to generate diagnostic information on sample code that cannot be vectorized. With that we already are faced with a common but trivial problem that hinders vectorization.

- Navigate to the matvector/c folder. This example calculates the product of a matrix and a vector.
- 2. Initially we compile our example without any vectorization done by the compiler. For this 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 -ansi-alias -O2 -no-vec multiply.c driver.c -o matvector \ ./matvector
```

Record execution time 48.045...

- 3. Next, we enable vectorization for AVX:
 - \$ icc -ansi-alias -O2 -xavx multiply.c driver.c -o matvector
 \$./matvector

Record execution time 45.395 .

4. In multiply.c, which contains the computation we're measuring, we use a pattern that's common to limit performance. Which is it? And, can it be fixed and how?

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

5. When applying a possible solution we notice a slight improvement of the performance.

```
Record execution time 44.249 .
```

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

```
$ icc -ansi-alias -02 -xavx multiply.c driver.c -o matvector -vec-report3
...
```

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.

It seems that the inner loop 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 vectorization reports.

- 1. Once more, look at the source code of multiply.c. The compiler needs additional information to see that the assumed dependencies (arrays are allocated in another compilation unit) are 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 -ansi-alias -O2 -xavx multiply.c driver.c -o matvector -fargument-noalias \$ ./matvector
```

Record execution time 44.195.

Also take a look at the vectorization reports.

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

This time no additional compiler options are needed:

```
$ icc -ansi-alias -O2 -xavx multiply.c driver.c -o matvector
$ ./matvector
```

Record execution time 21.044 ...

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.

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

```
$ icc -ansi-alias -O2 -xavx multiply.c driver.c -o matvector -restrict
$ ./matvector
```

Record execution time 20.547 .

Also take a look at the vectorization reports.

```
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 above and generate the assembly (-s). 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 use the solution with #pragma ivdep but the other 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 aligning the data.

1. The only data that is used throughout the loop 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 much:

```
$ icc -ansi-alias -02 -xavx multiply.c driver.c -o matvector
$ ./matvector

Record execution time 20.715 .

Hint: __declspcec(align(...)) Or __attribute__((aligned(...)))
Solution: solutions/align
```

This change was mostly a preparation for the next step, where we really benefit from proper 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 all the pointers in the function arguments refer to we can safely tell this to the compiler. There is a simple way to do so, which one?

The execution time once more is reduced:

```
\ icc -ansi-alias -O2 -xavx multiply.c driver.c -o matvector \ ./matvector
```

Record execution time 19.103 .

Hint: __assume_aligned(...)
Solution: solutions/assume_aligned

3. Moving our focus away from sole compiler optimization there is more room for improvement: The amount of elements per row is not multiple of what an AVX vector (or multiple thereof) can keep. Hence each row needs remainder handling.

The solution is padding, which 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 big improvement should become visible:

```
\ icc -ansi-alias -O2 -xavx multiply.c driver.c -o matvector \ ./matvector
```

Record execution time <u>11.388</u>.

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

Solution: solutions/padding

4. Please ensure that all elements per row are now multiple of elements a SIMD vector can take (for AVX: four 64 bit FP values [double]). This allows a more aggressive optimization regarding alignment: #pragma vector aligned

Apply it to the correct loop in multiply.c and measure the execution time once more:

```
\ icc -ansi-alias -O2 -xavx multiply.c driver.c -o matvector \ ./matvector
```

Record execution time 11.028.

Hint: Only one loop needs that pragma Solution: solutions/vector aligned

Note: Using this pragma and enforcing aligned accesses, unconditionally applies to all accesses inside the loop. If we had <u>not</u> padded the arrays before and use the pragma the compiler would create incorrect code. The reason is that the first row in the array a starts at an aligned address but the second one does not (elements per row are not multiple of the vector length). We won't see any crash for our example because the compiler tends to use unaligned moves which work for both aligned and unaligned data. 2nd generation Core™ processors, for example, 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 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.

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	n without	multi-	file	TPO
	,	complic	CIIC	*****	application	· wiciioac	minarci	1110	11 0

<pre>\$ icc -ansi-alias \$./sq</pre>	-03	-xavx	rabs.c	square_	_charge.c	trap.c	twod.c	-0	sq
Record execution tim	e								

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

```
$ icc -ansi-alias -03 -xavx 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-phase ipo inl

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 exercise 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 precision, order of operations, etc.). Even slower x87 instructions might be used. The vectorization reports tell which loops were not vectorized because of the selected FP model. As a compromise between speed and numerical stability (precision & 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.

Activity 3 - Pragma SIMD

In this activity, you become familiar with the pragma SIMD to vectorize code the compiler won't by default.

1.	Nav	/inata	tο	tha	a i ma	/~	folder.
Ι.	ING	viuate	ιU	uic	SIMO	···	ioiuei.

Record result .

Solution: solutions/simd

2.	Initially we compile our example without any modification and record the execution t	ime	&
	result:		

\$ icc -ansi-alias -O2 -xavx r \$./simd	main.c	simd.c	-0	simd
Record execution time				
Record result				

3. It is quite slow and can be much faster. Hence, this is a great opportunity to take a look at the vectorization reports:

```
$ icc -ansi-alias -02 -xavx main.c simd.c -o simd -vec-report3
...
```

Look at the implementation to understand which problems are reported. Note, that it's not important to understand what's computed but more, how it is done. Can you see we're having potential for vectorization? Use #pragma simd to enforce

- 1	icc ./si	-ansi-ali md	ias ·	-02	-xavx	main.c	simd.c	-0	simd
Re	ecord	execution	time	e					

vectorization here. What do you see and why?

Hint: Compare result to previous run; AVX uses vectors of eight 32 bit FP values [float]

4. What (obviously) needs to be done to safely vectorize the loop in simd.c in first place? Apply this change:

```
$ icc -ansi-alias -O2 -xavx main.c simd.c -o simd
$ ./simd

Record execution time _____.
Record result _____.
```

Also take a look at the vectorization reports. Why aren't they providing more information about the loop enhanced with the pragma?

Hint: What is the largest vector length possible for a correct result? Solution: solutions/vectorlength

Vectorization

- 5. The result is still not correct. There are two other properties of the loop body that need to be provided to the pragma:
 - Reduction taking place and
 - Access via a pointer that's linearly incremented

In the following two additional clauses are applied to the pragma to retrieve the correct result and clearly benefit from vectorization.

6. Can you find the local variable the reduction is applied to? Which operation defines the reduction?

Apply the corresponding clause to the pragma and compare execution time & result once more:

\$ icc \$./si	-ansi-alias md	-02 -xav	k main.c	simd.c	-0	simd
Record	execution tim	ne				
Record	result	·				
	ee comment n: solutions	/reductio	n			

7. Finally, identify the pointer which is linearly incremented. By which value?

Apply the corresponding clause to the pragma and compare execution time & result the last time:

-	cc -ansi /simd	-alias	-02 -xa	vx mai	in.c s	simd.c	-0	simd
Rec	ord execu	ition time	e	·				
Rec	ord result	t						

The result is correct now. What is the speedup compared to the initial (compiler only) version?

Hint: See comment

Solution: solutions/linear

Activity 4 – Guided Auto Parallelism (GAP)

In this activity, you will use the Guided Auto Parallelism (GAP) feature of the compiler to learn how to improve vectorization.

- 1. Navigate to the gap/c folder.
- 2. First, look into gap.c. How many problems regarding vectorization can you spot?
- 3. Use the GAP feature and let the compiler report instead of manually spotting possible problems:

```
$ icc -ansi-alias -std=c99 -O2 -xavx -c gap.c -guide
...
```

Vectorization

Not related to vectorization but for completeness rerun with -parallel option set:

```
$ icc -ansi-alias -std=c99 -O2 -xavx -c gap.c -guide -parallel
...
```

Note: Option -parallel is required for GAP's (auto) parallelism reports (see below).

In all cases no object was ever created. Verify that GAP does not produce any object/library/executable at all.

- 4. There are three GAP categories:
 - (auto) parallelization [-guide-par]; requires option -parallel
 - (auto) vectorization [-guide-vec]
 - data transformation [-guide-data-trans]

Rerun GAP with either of such options and compare the messages:

```
$ icc -ansi-alias -std=c99 -O2 -xavx -c gap.c -parallel -guide-par
...
$ icc -ansi-alias -std=c99 -O2 -xavx -c gap.c -guide-vec
...
$ icc -ansi-alias -std=c99 -O2 -xavx -c gap.c -guide-data-trans
```

5. In case of big source files it makes sense to restrict GAP to selected functions by using the compiler option <code>-guide-opts="..."</code>.

Rerun GAP and restrict analysis to either function "transform(...)" or "mult(...)".

Hint: Enclose the function name in ticks (') and the overall option value in double quotes (")

6. Apply the proposed fix(es) related to vectorization and rerun GAP.

Fortran

Activity 1.1 – Why Vectorization Failed

In this activity, you enable the compiler to generate diagnostic information on sample code that cannot be vectorized. With that we already are faced with a common but trivial problem that hinders vectorization.

- 1. Navigate to the matvector/fortran folder. This example calculates the product of a matrix and a vector.
- 2. Initially we compile our example without any vectorization done by the compiler. For this 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):

	<pre>\$ ifort -02 -no-vec driver.f90 multiply.f90 -o matvector \$./matvector</pre>
	Record execution time
3.	Next, we enable vectorization for AVX:
	<pre>\$ ifort -02 -xavx driver.f90 multiply.f90 -o matvector \$./matvector</pre>
	Record execution time
4.	In multiply.f90, which contains the computation we're measuring, we use a pattern that's common to limit performance. Which is it? And, can it be fixed and how?
	Hint: Non-unit stride access Solution: solutions/unit-stride
5.	When applying a possible solution we notice a slight improvement of the performance.
	Record execution time
	However, vectorization is still not optimal. We use the vectorization reports feature from the compiler to provide more information on why:
	\$ ifort -O2 -xavx driver.f90 multiply.f90 -o matvector -vec-report3

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 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 vectorization reports.

- 1. Once more, look at the source code of multiply.f90. The compiler needs additional information to see that the assumed dependencies (arrays are allocated in another compilation unit) are false dependencies. Those typically not occur in Fortran applications but when using Cray* pointers they are induced.
- 2. There are two options which can help us here: -fno-alias & -fno-fnalias

 Both ignore the aliasing induced by the Cray* pointers. Adding either of them clearly shows
 much better vectorization (e.g. -fno-alias):

•	ifort -O2 -xavx driver.f90 multiply.f90 -o matvector -fno-alias ./matvector
Re	cord execution time
	oking at the vectorization reports we get confirmation about successful vectorization of the

\$ ifort -02 -xavx driver.f90 multiply.f90 -o matvector -fno-alias -vec-report3

Note: It is crucial to understand that this solution is only safe to apply if the Cray* pointer pointees indeed are non-aliased memory locations. If this is not true the applied optimizations might change the semantics of the code and hence the results are different/incorrect. Fortunately our Cray* pointer pointees are separate memory locations, so this solution can be applied safely.

3. This was one of the rare examples (for Fortran) which shows dependencies as showstopper for vectorization. We're going to use another version now which passes the arrays directly. Please use the version in sub-directory array_version from now on and ensure that vectorization works well here, even without the option -fno-alias Or -fno-fnalias:

\$ ifort	-02	-xavx	driver	_array.f90	multiply_	_array.	£90	-0	matvector	-vec-report3
• • •										
Even the	e exec	ution t	ime is si	milar.						
Record	execut	ion tim	ie	·						

4. As preparation for the next activity generate the assembly (-s). 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 aligning the data.

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

The execution time won't change much:

	<pre>ifort -O2 -xavx driver_array.f90 multiply_array.f90 -o matvector ./matvector</pre>
Re	ecord execution time
	nt: !DIR\$ ATTRIBUTES ALIGN:

This change was mostly a preparation for the next step, where we really benefit from proper alignment.

2. Next, we look into multiply_array.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 passed in via the function arguments we can safely tell this to the compiler. There is a simple way to do so, which one?

The execution time once more is reduced:

<pre>\$ ifort -02 -xavx driver_array.f90 \$./matvector</pre>	multiply_array.f90	-o matvector
Record execution time		
Hint: !DIR\$ ASSUME_ALIGNED Solution: solutions/assume aligned		

3. Moving our focus away from sole compiler optimization there is more room for improvement: The amount of elements per column is not multiple of what an AVX vector (or multiple thereof) can keep. Hence each column needs remainder handling. The solution is padding, which is controlled via ROWBUF in our example. Understand how it

The solution is padding, which is controlled via ROWBUF in our example. Understand how it works and apply a correct value to it so it is done correctly. When ROWBUF is set correctly another big improvement should become visible:

- 1	ifort ./matv			driver_	_array.f90	multiply_	_array.f9	0 -0	matvector
R	ecord ex	xecut	ion tim	ne					

Vectorization

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

Solution: solutions/padding

4. Please ensure that all elements per row are now multiple of elements a SIMD vector can take (for AVX: four 64 bit FP values [real*8]). This allows a more aggressive optimization regarding alignment: !DIR\$ VECTOR ALIGNED

Apply it to the correct loop in multiply_array.f90 and measure the execution time once more:

	ifort ./matv			driver_	_array.f90	multiply	_array.f	:90 -o	matvecto	1
R	ecord e	xecut	tion tim	ne						

Hint: Only one loop needs that directive 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 use the directive the compiler would create incorrect code. The reason is that the first column in the array a starts at an aligned address but the second one does not (elements per column are not multiple of the vector length). We won't see any crash for our example because the compiler tends to use unaligned moves which work for both aligned and unaligned data. 2^{nd} generation $Core^{TM}$ processors, for example, 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 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.

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. Na	vigate to	the squar	e_charge	/fortran	folder.
-------	-----------	-----------	----------	----------	---------

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

```
$ ifort -03 -xavx 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 -03 -xavx 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 -opt-report-phase ipo inl

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 exercise 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 precision, order of operations, etc.). Even slower x87 instructions might be used. The vectorization reports tell which loops were not vectorized because of the selected FP model. As a compromise between speed and numerical stability (precision & reproducibility) we recommend to use the option -fp-model precise instead of the strictest IEEE 754 interpretation via -fp-model strict.

Activity 3 - Directive SIMD

In this activity, you become familiar with the directive SIMD to vectorize code the compiler won't by default.

- 1. Navigate to the simd/fortran folder.
- Initially we compile our example without any modification and record the execution time & result:

<pre>\$ ifort -02 -xavx main.f90 \$./simd</pre>	simd.f90	-0	simd
Record execution time	_·		
Record result			

3. It is quite slow and can be much faster. Hence, this is a great opportunity to take a look at the vectorization reports:

```
$ ifort -O2 -xavx main.f90 simd.f90 -o simd -vec-report3
...
```

Look at the implementation to understand which problems are reported. Note, that it's not important to understand what's computed but more, how it is done. Can you see were we're having potential for vectorization? Use !DIR\$ SIMD to enforce vectorization here. What do you see and why?

•	./simo		-xavx	main.f90	simd.190	-0	sima
R	ecord e	xecu	tion tim	ne	_·		
_		14					

Hint: Compare result to previous run; AVX uses vectors of eight 32 bit FP values [real] Solution: solutions/simd

4. What (obviously) needs to be done to safely vectorize the loop in simd.f90 in first place? Apply this change:

```
$ ifort -02 -xavx main.f90 simd.f90 -o simd
$ ./simd

Record execution time _____.
Record result _____.
```

Also take a look at the vectorization reports. Why aren't they providing more information about the loop enhanced with the directive?

Hint: What is the largest vector length possible for a correct result? Solution: solutions/vectorlength

Vectorization

- 5. The result is still not correct. There are two other properties of the loop body that need to be provided to the directive:
 - Reduction taking place and
 - Access via separate index that's linearly incremented

In the following two additional clauses are applied to the directive to retrieve the correct result and clearly benefit from vectorization.

Apply the corresponding clause to the directive and compare execution time & result once

6. Can you find the local variable the reduction is applied to? Which operation defines the reduction?

more:
\$ ifort -O2 -xavx main.f90 simd.f90 -o simd
\$./simd

Record execution time ______.

Record result ______.

Hint: See comment
Solution: solutions/reduction

7. Finally, identify the separate index which is linearly incremented. By which value?
Apply the corresponding clause to the directive and compare execution time & result the last time:
\$ ifort -O2 -xavx main.f90 simd.f90 -o simd
\$./simd

Record execution time ______.

Record result ______.

The result is correct now. What is the speedup compared to the initial (compiler only) version?

Hint: See comment

Activity 4 – Guided Auto Parallelism (GAP)

In this activity, you will use the Guided Auto Parallelism (GAP) feature of the compiler to learn how to improve vectorization.

1. Navigate to the gap/fortran folder.

Solution: solutions/linear

- 2. First, look into gap.f90. How many problems regarding vectorization can you spot?
- 3. Use the GAP feature and let the compiler report instead of manually spotting possible problems:

```
$ ifort -O2 -xavx -c gap.f90 -guide ...
```

Vectorization

Not related to vectorization but for completeness rerun with -parallel option set:

```
$ ifort -O2 -xavx -c gap.f90 -guide -parallel
...
```

Note: Option -parallel is required for GAP's (auto) parallelism reports (see below).

In all cases no object was ever created. Verify that GAP does not produce any object/library/executable at all.

- 4. There are three GAP categories:
 - (auto) parallelization [-guide-par]; requires option -parallel
 - (auto) vectorization [-guide-vec]
 - data transformation [-guide-data-trans]

Rerun GAP with either of such options and compare the messages:

```
$ ifort -02 -xavx -c gap.f90 -parallel -guide-par
...
$ ifort -02 -xavx -c gap.f90 -guide-vec
...
$ ifort -02 -xavx -c gap.f90 -guide-data-trans
...
```

5. In case of big source files it makes sense to restrict GAP to selected routines by using the compiler option <code>-guide-opts="..."</code>.

Rerun GAP and restrict analysis to either routine "transform (...)" or "move (...)".

Hint: Use line numbers and enclose the overall option value in double guotes (")

6. Apply the proposed fix(es) related to vectorization and rerun GAP.