Basics of Vectorization for Fortran Applications

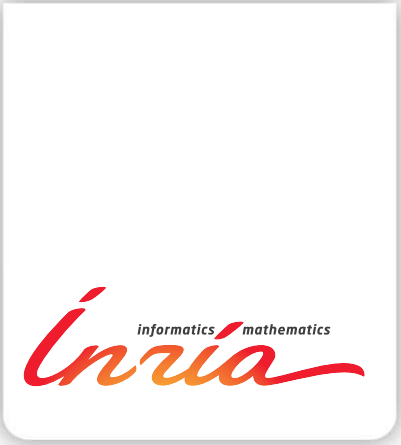
Laércio Lima Pilla

Basics of Vectorization for Fortran Applications

Laércio Lima Pilla

RESEARCH REPORT
N° 9147
January 2018
Project-Team CORSE

ISSN 0249-6399 ISRN INRIA/RR--9147--FR+ENG



Basics of Vectorization for Fortran Applications

Laércio Lima Pilla*

Project-Team CORSE

Research Report n° 9147 — January 2018 — 9 pages

Abstract: This document presents a general view of vectorization (use of vector/SIMD instructions) for Fortran applications. The vectorization of code becomes increasingly important as most of the performance in current and future processor (in floating-point operations per second, FLOPS) depends on its use. Still, the automatic vectorization done by the compiler may not be an option in all cases due to dependencies, ambiguity, or sparse data access.
In order to cover the basics of vectorization, this document explains the operation of vector instructions for different architectures, how code vectorization can be done, and how to test if your code has vectorized well.
This document is intended mainly for use by developers and engineers with basic knowledge of computer architecture and programming in Fortran. It was designed to serve as a starting point for people working on the vectorization of applications, and does not address the subject in all its details.

Key-words: Fortran, vectorization, parallelization, compiler

* Contact: laercio.lima@inria.fr

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Notions de base pour la vectorisation des applications Fortran

Résumé : Ce document présente une vue générale de la vectorisation (utilisation d'instructions vectorielles / SIMD) pour les applications Fortran. La vectorisation du code devient de plus en plus importante car elle permet d'augmenter fortement le nombre d'opération flottantes par seconde (FLOPS) avec les processeurs actuels et futurs. Cependant, la vectorisation automatique effectuée par le compilateur peut ne pas être une option dans tous les cas en raison de dépendances, d'ambiguïtés ou d'accès aux données éparses.

Afin de couvrir les bases de la vectorisation, ce document explique explique le fonctionnement des instructions vectorielles pour différentes architectures, comment le code peut être vectorisé, et comment tester si votre code a bien été vectorisé.

Ce document est principalement destiné aux développeur·euse·s et aux ingénieur·e·s ayant une connaissance de base de l'architecture informatique et de la programmation en Fortran. Il a été conçu pour servir de point de départ aux personnes travaillant sur la vectorisation des applications et n'aborde pas le sujet dans tous ses détails.

Mots-clés : Fortran, vectorisation, parallélisation, compilateur

Contents

1 Introduction

Current and future processor architectures use parallelism at different levels to achieve high performance while respecting power limitations. We can cite three different levels of parallelism and their characteristics as follows:

- **Instruction-Level Parallelism (ILP)** happens when multiple instructions of the same thread are executed concurrently. This is done automatically by complex features present in the organization of a processor, such as pipelining and the dispatch of multiple instructions per cycle in a dynamic manner (called superscalar processors) or a static manner (called very long instruction word processors, or VLIW). Instruction-level parallelism is achieved by a combination of efforts from the processor with its features, and the compiler with optimization passes that accentuate the parallelism between instructions.

- **Thread-Level Parallelism (TLP)** regards the execution of different instruction streams (threads) in a processor. These threads can execute in parallel in different processor cores in a multicore or manycore processor, and also in the same core in what is called *multithreading*. These threads can come from the same application or different application running concurrently in the same processor. Thread-level parallelism is usually explored in one application by using parallel programming APIs like OpenMP and MPI.

- **Data-Level Parallelism**, in this context, is related to the presence of vector instructions, which are also called **SIMD** (single instruction, multiple data) instructions based on Flynn's Taxonomy. These vector instructions apply the same operation over multiple data items (like integers and floating-point values) at the time, given that these items are stored contiguously in vector/SIMD registers. Data-level parallelism carries the bulk of the FLOPS (floating-point operations per second) performance of current processors. Nevertheless, using it requires the *vectorization* of parts of the application involving combined efforts from developers and the compiler.

While ILP is mostly hidden from the developers and TLP is fairly well understood and evident after over fifteen of its presence in commercial processors, vector instructions in general-purpose processors have been comparatively less explored by applications and their developers. However, as the importance of SIMD hardware increases in current and future processors, so do the efforts

to fully benefit from it. In this context, this document provides some basic information and guidelines to help Fortran developers vectorize their applications. We organize this information in three main parts: first we discuss the hardware available in Section 2, then we discuss the ways we may vectorize our code in Section 3, and finally we explore some of the things that may make it difficult to vectorize code and some ways to see if it really happened in Section 4.

2 SIMD hardware

The SIMD features in processors have been increasing in recent years. These changes are related to the size (number of elements or bits) of the SIMD registers, the availability of basic and complex SIMD operations, and some other features that can enable more vectorization (like predication). Since these features are different among architectures (and even extensions of the same architecture), we will shortly discuss the SIMD hardware present in Intel/AMD x86 processors and in ARM processors. Nonetheless, since the use of Fortran applications in ARM processors is still limited, the remaining sections of this document will focus in x86 processors only.

2.1 Intel/AMD x86

The x86 (and x86-64) Instruction Set Architecture (ISA) is the standard architecture present in processors made by Intel and AMD. This ISA by itself does not include SIMD instructions and registers, so the companies came with different ISA extensions through the years. A relevant subset of these extensions is as follows:

- MultiMedia eXtensions (MMX): made available in 1996, it includes 64-bit registers that can operate on two 32-bit integers at the same time. It also supports larger numbers of smaller values (like four 16-bit values).
- Streaming SIMD Extensions (SSE): made available in 1999, it includes 128-bit registers that can operate on four 32-bit floating-point values at the same time. Its later versions (up to SSE4.2) can split the 128 bits into different numbers of values.
- Advanced Vector eXtensions (AVX): made available in 2008, it includes 256-bit registers that can operate on eight 32-bit or four 64-bit floating-point values at the same time. Its later version (AVX2) can also work with integers of different sizes.
- Advanced Vector eXtensions 512 (AVX-512): made available in 2015, extends AVX to work with 512-bit registers and double the values.

As the evolved ISA extensions usually support their previous versions, applications compiled for previous processor generations are still able to partly benefit from the SIMD hardware present in modern processors. Nevertheless, when an application is compiled to use the most modern SIMD features, it may be able to achieve higher performance but pays the cost of losing backwards compatibility.

2.2 ARM

The ARMv8 architecture includes the Advanced SIMD extension, commonly known as NEON. NEON includes registers with up to 128 bits, enabling SIMD operations over two 64-bit or four 32-bit values, among others. Although this may seem small compared to the x86 extensions

mentioned before, it is important to remember that ARM processors were mainly developed for embedded systems with power restrictions.

A recent change in processor design for High-Performance Computing by ARM was announced in 2016 with the development of Scalable Vector Extension (SVE) [1]. While previous SIMD extensions had fixed register sizes, SVE will include registers starting from 128 bits up to 2048 bits (in multiples of 128). By adding this flexibility directly into the ISA, future ARM processors will be able to use SIMD features in different levels while keeping compatibility and reusing the same compiled code. Larger SIMD capabilities also means that more of the processors' performance may depend in the correct vectorization of code.

3 Vectorization approaches

In order to vectorize an application or its parts, there are three different approaches in general. We can cite them in increasing order of complexity as follows:

- **Auto-vectorization by the compiler** is the most effortless way one may vectorize code, as it is not require any changes in the application's code. Although modern compilers are able to optimize applications in many ways, sometimes they are unable to handle some ambiguous, sparse, or complex data access patterns.
- **Guided vectorization with notations** requires the addition of special comments in the application's code by the developer in order to inform and help the compiler. In addition, this can also be accompanied by changes in the code's data layout, loops, and functions to help highlight vectorization opportunities.
- **Low-level vectorization with intrinsics and assembly** require drastic changes to the application's code in order to use special low-level functions and data types that are directly mapped to assembly code. Besides affecting portability, this is not available in Fortran, although some work to provide something similar to intrinsics has already been done [1]

One way to approach this challenge would be to start by using auto-vectorization and following it with notation additions wherever needed. This idea is developed in the next subsections, while some issues related to vectorization and proposed code changes are left to Section 4.

3.1 Auto-vectorization by the compiler

When instructed to optimize code, compilers are sometimes able to find vectorization opportunities with no additional clues from developers. In order to request this kind of effort from a compiler, developers add optimization flags to the compiler call.

While some optimization flags are fairly standard among compilers, others are not. In order to illustrate the use of optimization flags, we will explore their use with the Intel and GCC Fortran compilers, respectively named `ifort` (version 17.0.3) and `gfortran` (version 5.4.0). Let us consider the resumed dot product function below:

```
function dot_p (a, b) result(res)
    ! Input information ommited ...
    array_size = min(size(a), size(b))   ! Initialize values
    res = 0.0d+0
```

[1] Presentation "Enabling Manual Vectorization of Complex Code Patterns in Fortran" by Florian Wende: `https://www.ixpug.org/docs/sc14_ixpug_bof_Wende.pdf`

```
    do i = 1, array_size                    ! Main loop
        res = res + a(i) * b(i)
    enddo
end function dot_p
```

The aforementioned function contains a simple loop with no dependencies between iterations, which provides plenty of opportunity for vectorization. In this situation, the effects of optimization flags are as follows:

- **Optimization levels**: it is usual for compilers to group several optimization passes into optimization levels. These optimization levels are set using flags starting by `-O` (minus followed by the letter O), like `-O2` and `-Ofast`.
  - `-O0`: this flag explicitly removes all optimizations, so no vectorization is done.
  - `-O1`: this optimization level only includes optimizations that do not increase the size of the code. This vectorization is usually accompanied with size-increasing optimizations like loop unrolling, no vectorization is done.
  - `-O2`: this level enables loop unrolling and vectorization. In our case, `ifort` was able to vectorize this code at this optimization level, while `gfortran` was not. Additionally, if no optimization level is set, the standard level for `ifort` is `-O2`.
  - `-O3`: this level includes all optimizations from `-O2` and also adds some more complex ones, like loop fusion. At this level, `gfortran` also vectorizes the dot product loop.
- **Architecture flags**: during the compilation process, the machine code to be generated usually includes only the standard ISA of the machine where the compiler is running or the target architecture (if set). Nevertheless, modern processors usually include different ISA extensions with SIMD instructions and registers (as mentioned in Section 2). In order to fully utilize the vector features of a processor, the flags below can be used. Nevertheless, it is important to emphasize that using these flags will disrupt backward compatibility with older processors that do not include some of the ISA extensions.
  - In `gfortran`: the flag `-march=native` can be used to inform the compiler that it can use the whole ISA from its current machine. To inform it to use specific ISA extensions, the `-m` flag can be used, e.g., `-msse2` and `-mavx512f`. The optimization level `-Ofast` includes both `-O3` and `-march=native`.
  - In `ifort`: a synonym to `-march=native` in this compiler is `-xHost`. It can also be used to specify other ISA, such as `-xCORE-AVX2` and `-xMIC-AVX512`.
- **Memory alignment**: in `ifort`, the flag `-align` can be used to try to enforce that addresses start at a multiple of a value. Its general use for vectorization is `-align array64byte`, which indicates that arrays should start at multiples of 64 Bytes (512 bits), which is the size of a block in common cache levels.

3.2 Guided vectorization with notations

When the compiler is unable to resolve all issues related to the vectorization of loops and functions, the developers can add some special kinds of comments that serve to guide the compiler. The comment line directives, also known as pragmas in the context of the C language, are evaluated during the pre-compilation step. They follow the same behavior than the ones used for parallel programming with OpenMP.

One of the first comment line directives employed to help in the vectorization of code is **IVDEP**. It serves to inform the compiler that it should ignore assumed vector dependencies. The compiler assumes a dependency when, due to missing information, it cannot tell if loop iterations are dependent on one another or not. For instance, IVDEP can be used in the example below to enable vectorization, else the compiler would assume that the value of `j` could be negative, which would make vectorization illegal. Nevertheless, IVDEP does not override real dependencies detected by the compiler.

```
!dir$ ivdep
do i = 5, 20
    a(i) = a(i + j) * k
done
```

Another comment line directive is **SIMD**. It is employed to inform the compiler that a certain part of the code should be vectorized. Nevertheless, the compiler may still decide to avoid doing it in the case of ambiguity or probable performance losses. The more portable way of using SIMD is using the directive available in OpenMP since its version 4.0. In this case, the compiler must be informed to use OpenMP (flag `-fopenmp`). If one is using `ifort` only, the directive `!dir$ simd` can be used instead. This can also help the compiler in situations with some ambiguity where it would generate different versions of the loop (some vectorized, some not).

```
!$omp simd
do i = 1, array_size
    res = res + a(i) * b(i)
enddo
!$omp end simd
```

If the application code already uses OpenMP to divide loops iterations among threads, the SIMD directive can be added to the same lines to try to vectorize the code.

In the case where a function is called for the items of a vectorized loop, the **Declare SIMD** directive, available in OpenMP version 4.5, can be used to inform the compiler that it should generate a vector version of the function. In this context, the **uniform** clause can be used to inform the compiler of any constant/invariant values passed to the function in all iterations. An example of its use is presented below.

```
function add_two(a,b,my_const) result(c)
!$omp declare simd(add_two) uniform(my_const)
...
```

For more examples with the SIMD directive, please refer to [2, Chapter 5].

4 Vectorization issues

Many issues may harm the ability of the compiler to vectorize some code. We discuss some of these issues in the next subsection, followed by ways to verify if vectorization was successfully done or not by the compiler. For a detailed list of vectorization inhibitors and possible solutions, please refer to [3, Chapter 7].

4.1 Inhibitors

A short list of issues that may prevent the compiler from vectorizing code is presented below.

- *Conditional termination of the loop*: if a condition inside the loop may make it terminate before executing all its iterations, the code may not be vectorized because the compiler usually does not vectorize loops with multiple exits. The example below illustrates a situation where the loop may terminate prematurely.

```
do i = 1, array_size
    if (a(i) > 100) exit
    res = res + a(i) * b(i)
enddo
```

- *Function calls within the loop*: if the loop iteration includes function calls that are not vectorized or inlined, then the compiler will not vectorized the loop. An example is illustrated below.

```
do i = 1, array_size
    res = res + non_vect_fun(a(i)) * b(i)
enddo
```

- *[Ambiguous] overlap between indexes*: as mentioned before for IVDEP, the compiler will avoid vectorizing code if it assumes that there is a dependency between iterations. Additionally, it will not vectorize code in the case of a real dependency (where the value read in one iteration is written during a previous iterations). An ambiguous overlap and a real dependency are shown in the example below.

```
do i = 2, array_size
    c(i) = c(d(i))        ! ambiguous about the dependency
    a(i) = a(i-1) * b(i) ! real dependency, cannot be vectorized
enddo
```

- *Lack of spatial locality*: vectorization benefits from contiguous memory accesses. If the data access pattern is too sparse, then the cost of gathering and scattering data may overcome the benefits from vectorization, so the compiler will avoid doing it. The example below illustrates two scenarios of sparse access: one due to indirect access and the other due to strided access.

```
do i = 1, array_size
    res = res + a(c(i)) * b(d(i))                          !indirect
    e(array_size+1-i) = a(1 + modulo(4*i, array_size))    !strided
enddo
```

4.2 Checking if vectorization happened

One important part of vectorizing an application is to identify which parts of the code can/were successfully vectorized and which were not, so the developers can focus their efforts in solving the remaining issues. This can be done in different ways, some of which are presented below.

- *Compilation report*: one of the most direct ways to gather information on which parts of the were vectorized is to make the compiler report its optimization process. One requests this information from the compiler by using compiler-dependent flags. Not only this usually provides information on how a loop was parallelized, it can also provide hints to fix issues in parts which were not.

  - In `gfortran`: `-fopt-info` is the general flag to gather information on optimization passes. Some variations of this flag are `-fopt-info-missed`, `-fopt-info-vec` and `-fopt-info-loop`.
  - In `ifort`: the compilation report can be gathered using the `-qopt-report=N` flag, where `N` is a natural number. Values bigger than '2' usually provide valuable information related to vectorization.

- *Assembly code*: if one only wants to know if some code contains SIMD instructions, one way to do it is to check its assembly code. For instance, in x86 processors, a vectorized application uses SIMD registers (e.g., `%xmm`) and SIMD instructions (which usually include additional letters like "p" for parallel, as in `addpd`). This information can be obtained in different ways depending if one has access to the application's code or not.
  - *With code*: the compiler flag `-S` can be used to force the compiler to save the generated assembly code in a file.
  - *Without code*: there are tools to check the machine code from a program. For instance, in Unix environments, the command `objdump -d` can be used to disassemble an application and display its equivalent code in assembly.

- *Support tools*: some of the tools used to profile applications can also be used to study specific parts of them. For instance, in Intel environments, Intel VTune [2] can be used to check for hotspots in the code, and to check if these parts are vectorized or not. The same can be said for the MAQAO tool [3], which is made available under LGPL.

Acknowledgments

The author would like to thank Fabian Gruber for his help with questions related to compilers, Fabrice Rastello for his correction on issues related to compilers and suggestions, Raphaël Jakse for his review of the abstract, and Phillippe Virouleau for his general comments.

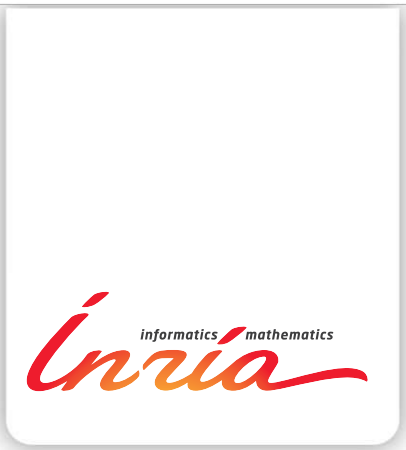
References

[1] Nigel Stephens. ARMv8-A Next-Generation Vector Architecture for HPC. `https://community.arm.com/cfs-file/__key/telligent-evolution-components-attachments/01-2142-00-00` 2016.

[2] OpenMP Architecture Review Board. OpenMP Application Programming Interface Examples - version 4.5.0. `http://www.openmp.org/wp-content/uploads/openmp-examples-4.5.0.pdf`, 2016.

[3] J. Levesque and A. Vose. *Programming for Hybrid Multi/Manycore MPP Systems*. Chapman & Hall/CRC Computational Science. CRC Press, 2017.

[2] `https://software.intel.com/en-us/intel-vtune-amplifier-xe`
[3] `http://www.maqao.org/`



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399