Performance Tuning Using Vectorization

Nicolay J. Hammer *

HLST Core Team
Max-Planck-Institut für Plasmaphysik

February 25, 2011

*tel.: +49 (0)89 3299 3529 , email: nicolay.hammer@ipp.mpg.de

Contents

1 Introduction **4**

2 Test Case Definition for the Performance Measurement **7**

3 Hardware Architectures and Vectorization **8**
3.1 Intel Xeon X5570 CPU with Nehalem Architecture 8
3.2 IBM POWER6 CPU and Architecture 9

4 The Intel Fortran Compiler `ifort` and the Intel Performance Libraries **9**
4.1 The Math Kernel (MKL) Library 9
4.2 The Integrated Performance Primitives (IPP) Library 10

5 Vectorization and the Intel Nehalem CPU **11**
5.1 Basic Run Time Behavior 11
5.2 Vectorization using the Intel `ifort` auto-vectorizer unit 11
5.3 Vectorization using the Intel MKL Library 13
5.4 Vectorization using the Intel IPP Library 13
5.5 Vectorization gain on Intel Nehalem 13

6 The IBM XL Fortran (XLF) compiler and the Mathematical Acceleration Subsystem (MASS) Library **15**

7 Vectorization and the IBM POWER6 CPU **16**
7.1 Vectorization using the IBM XLF auto-vectorizer unit and the vector MASS library 16
7.2 Vectorization gain on IBM POWER6 17

8 Intel Nehalem vs. IBM POWER6 **17**

9 Issues of Vectorization **20**
9.1 Impact of the vector length on the execution runtime 20
9.2 Unsupported Formats of the `ifort` Auto-Vectorizer 20
9.3 IBM XL Fortran SIMD Functions on POWER6 20

10 Summary and Conclusions **21**

References **24**

Online References **24**

A Summary of Vector Statement Execution Times **27**

B Double Precision Performance Measurement **28**

C Single Precision Performance Measurement **44**

List of Tables

1 Fastest vector statement execution times. 14
2 Vectorization gain factors. 18
3 Vector statement execution times on an Intel Xeon X5570 CPU. . 27
4 Vector statement execution times on an IBM POWER6 CPU. . . 27

List of Figures

1 SISD vs. SIMD. 5
2 Example of vectorized FORTRAN source code. 6
3 Source code example for testing vectorization behavior. 7
4 Example of an Intel IPP interface. 10
5 Basic structure of runtime vs. vector length measurements. . . . 12
6 Execution runtimes: Intel Nehalem vs. IBM POWER6. 15
7 Rel. gain of exec. runtime: Intel Nehalem vs. IBM POWER6. . . 19
8 Fortran intrinsic MULT operation sampled coarse and fine. 21
9 Fortran intrinsic ABS function. 28
10 Fortran intrinsic ADD and fused MULTIPLY ADD operation. 29
11 Fortran intrinsic ATAN2 function. 30
12 Fortran intrinsic DIVISION operation. 31
13 Fortran intrinsic EXP function. 32
14 Fortran intrinsic INT and NINT function. 33
15 Limiting operation using the Fortran intrinsic MIN function. . . . 34
16 Fortran intrinsic MIN function. 35
17 Fortran intrinsic POW operation. 36
18 Fortran intrinsic POW3O2 operation. 37
19 Fortran intrinsic MODULO function. 37
20 Fortran intrinsic SIN function. 38
21 Fortran intrinsic SQRT function. 39
22 Fortran intrinsic MULT and SQR operation. 40
23 Fortran intrinsic FLOOR and REAL function. 41
24 Intel IPP VECTOR PACKING operations. 42
25 Intel IPP VECTOR PACKING operations. 43
26 Fortran intrinsic MULT and EXP operation (32 bit). 44

1 Introduction

According to Flynn's Taxonomy, a classification of computer architectures [1], there are four classifications defined by the instruction and data streams which are available in a specific architecture. These are Single Instruction, Single Data stream (SISD, Fig. 1 left panel), Single Instruction, Multiple Data streams (SIMD, Fig. 1 right panel), Multiple Instruction, Single Data stream (MISD), and Multiple Instruction, Multiple Data streams (MIMD). For further details please see [A]. This classification of computer architectures can also be viewed as a classification by parallelism. Then SISD architectures correspond to a single core serial (non-parallel) computer, MIMD to a multi core parallel computer, and SIMD to vector computer. MISD are rather rare systems which do not fit really into this picture, e.g. the flight control computer of the Space Shuttle.

In this report we focus on Single Instruction, Multiple Data architectures, where one single instruction is executed on a multiple set of data in parallel. Furthermore, we focus on double precision (64 bit) test cases, since most of the scientific simulation software uses double precision floating point numbers. An advanced question would be whether some scientific numerical codes could partially use single precision (32 bit) floating point numbers instead and how much performance could be gained due to vectorization. However, this question reaches beyond the topics covered in the work presented in this report.

The goal of the work presented here is to investigate the performance gain of intrinsic functions making use of the SIMD capabilities of Intel processors with Nehalem architecture. It was triggered by the EUTERPE vectorization project of HLST which has its impact on the selection of intrinsic functions taken here and on the choice of the vectorization concept.

The Streaming SIMD Extensions (SSE) is a SIMD instruction set extension to the x86 architecture, designed by Intel and introduced in 1999 in their Pentium III series processors. The Nehalem processor supports SIMD instructions of the SSE4.2 instruction set to access small-scale SIMD with 128 bit registers. In this connection, the 128 bits of the SIMD registers can hold four 32 bit, two 64 floats (Fig. 1), eight 16-bit short integers, four 32 bit, two 64 bit integers, and sixteen 8-bit bytes or characters, respectively [B]. The capability of SIMD in Intel and AMD processors will be further extended in the future as Intel has already announced the Advanced Vector Extensions (AVX) which will be the new 256 bit instruction set successor to SSE and is designed for applications that are floating point intensive. In addition, modern Graphics Processing Units (GPUs) are using Single Instruction, Multiple Thread (SIMT) implementations, capable of branches, loads, and stores on 128 or 256 bits at a time. Basically, SIMT is a variant of the SIMD class. If a code has been programmed for SIMD it will automatically run on a SIMT architecture but not the other way round. Hence, it is of key interest to investigate today's Nehalem's SIMD capabilities for realistic double precision (64 bit) problems to be in line with future hardware development.

A necessary requirement for vectorization is the absence of true backward or loop data dependencies across the iterations of a loop statement. Only operations which are completely independent from each other or just depend on results whose calculation has been completed before can be executed in parallel. The vectorization concept discussed in the rest of this section matches the requirement of the EUTERPE vectorization project mentioned above. Therefore

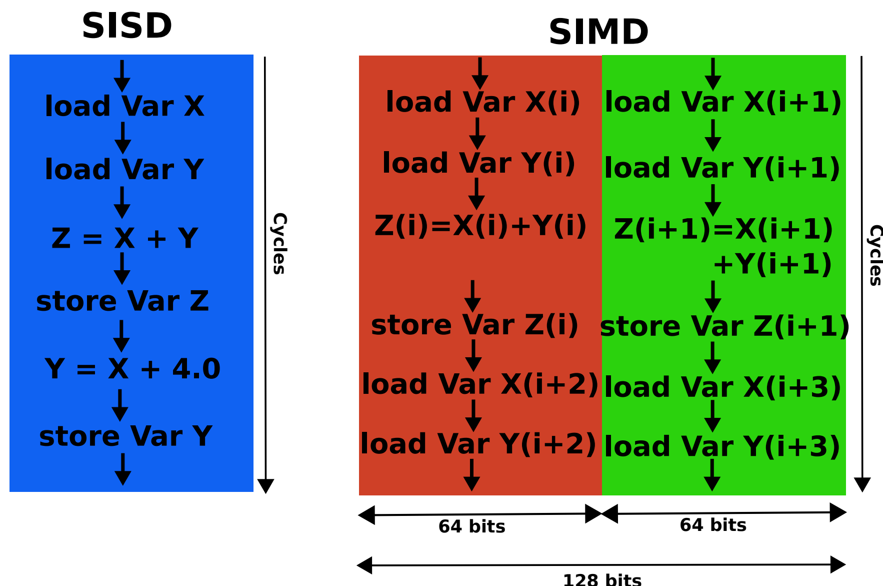

Figure 1: Left Panel: Schematic floating point work flow in the single instruction, single data (SISD) concept which corresponds to the scalar execution. Right Panel: Schematic floating point work flow in the single instruction, multiple data (SIMD) concept which corresponds to the vector execution. In the shown example the 128 bit SIMD or vector unit is divided into 2×64 bit which corresponds to a double precision vector operation.

it cannot be seen as the universal concept of vectorization. In this concept the vectorization of source code follows the subsequent described procedure. In a first step the main loop within the target source code part, e.g. a subroutine, is split into an outer loop providing chunks of vector data and an inner loop working on those chunks. This procedure is commonly known as “strip mining”. The size of those data chunks can be controlled by an according parameter. Thus, the outer loop enables a complete control over the length of work vectors which will be processed by the vector units of a CPU through the inner loop statements which will be vectorized. The basic idea of a vector pipeline unit inside a CPU compared to a scalar pipeline is sketched in Fig. 1.

In a subsequent number of working steps, the inner loop is split into several block loops of same size (Fig. 2, middle panel) but containing less source code statements, usually one to a few. Correspondingly, scalar variables used inside the former scalar main loop are exchanged by vector variables. This is a mandatory step for vectorization, no matter whether done intrinsically by the auto-vectorizer of the compiler, e.g. the auto-vectorizer of `ifort` (Section 4) or done manually by the programmer (Fig. 2, lower panel), using e.g. the Vector Mathematical Functions (VMF) of the Intel Math Kernel Library (MKL) (see Subsection 4.1 and [C]). The purpose is the following. Statements which shall be replaced by a vectorized version manually have to be isolated first within the source code. Moreover, the auto-vectorizer units of the compiler have certain limitations when analysing complex code structures and will simply fail if the source code structure exceeds this complexity limit. However, by splitting big

```
DO ip = 1, npart_loc
   .
   .
   .
   rhog = SQRT(2.0 * mut0 / b_abs) * msdqs(species)

   navg = NINT(rhog * REAL(NAVG_TRM))

   IF (navg < NAVG_TRM) THEN
      navg = NAVG_TRM
   ELSE IF (navg > NAVG_MAX) THEN
      navg = NAVG_MAX
   END IF
   .
   .
   .
END DO
```

```
tmp1_vec(1:newSize) = 2.0 * mut0_vec(1:newSize)
CALL VDDIV ( newSize, tmp1_vec(1:newSize), b_abs_vec(1:newSize), &
                      tmp2_vec(1:newSize) )
CALL VDSQRT( newSize, tmp2_vec(1:newSize), tmp1_vec (1:newSize) )
rhog_vec(1:newSize) = tmp1_vec(1:newSize) * msdqs(species)

! select number of points on the gyro-ring
DO ip = 1, newSize !remark: LOOP WAS VECTORIZED.
   rhog = rhog_vec(ip)
   ! NINT(tmp1) replaced by INT( tmp1 + 0.5 )
   tmp1_vec(ip) = rhog * navg_trm_real + 0.5
END DO
DO ip = 1, newSize !remark: LOOP WAS VECTORIZED.
   tmp1 = tmp1_vec(ip)
   navg_vec(ip) = INT(tmp1)
END DO
DO ip = 1, newSize ! remark: LOOP WAS VECTORIZED.
   navg = navg_vec(ip)
   navg_vec(ip) = MAX( navg, NAVG_TRM )
END DO
DO ip = 1, newSize ! remark: LOOP WAS VECTORIZED.
   navg = navg_vec(ip)
   navg_vec(ip) = MIN( navg, NAVG_MAX )
END DO
```

```
tmp1_vec(1:newSize) = 2.0 * mut0_vec(1:newSize)
CALL VDDIV ( newSize, tmp1_vec(1:newSize), b_abs_vec(1:newSize), &
                      tmp2_vec(1:newSize) )
CALL VDSQRT( newSize, tmp2_vec(1:newSize), tmp1_vec (1:newSize) )
rhog_vec(1:newSize) = tmp1_vec(1:newSize) * msdqs(species)

! select number of points on the gyro-ring
DO ip = 1, newSize !remark: LOOP WAS VECTORIZED.
   rhog = rhog_vec(ip)
   ! NINT(tmp1) replaced by INT( tmp1 + 0.5 )
   tmp1_vec(ip) = rhog * navg_trm_real + 0.5
END DO
DO ip = 1, newSize !remark: LOOP WAS VECTORIZED.
   tmp1 = tmp1_vec(ip)
   navg_vec(ip) = INT(tmp1)
END DO
DO ip = 1, newSize ! remark: LOOP WAS VECTORIZED.
   navg = navg_vec(ip)
   navg_vec(ip) = MAX( navg, NAVG_TRM )
END DO
DO ip = 1, newSize ! remark: LOOP WAS VECTORIZED.
   navg = navg_vec(ip)
   navg_vec(ip) = MIN( navg, NAVG_MAX )
END DO
```

Figure 2: Upper Panel: Example of a FORTRAN source code section showing a loop statement before splitting the statement into single loop blocks. Middle Panel: Same statement as above but after splitting into single loop blocks. Lower Panel: The same source code section as shown in the upper panel after replacing the square root function by the vector square root function provided by the VMF of Intel's MKL.

Please note that, the source code examples shown here are taken from the vectorization of a particle in cell code using manual "strip mining", thus they are not universal (see Section 1).

```
CALL PERFON ('autovec')

DO nb = 0, nBlocks - 1

   nOfs   = nb * nChunk
   nSize  = MIN( nmax - nOfs , nChunk )
   nStart = nOfs + 1
   nStop  = nOfs + nSize

   vec1(1:nSize) = rearr1(nStart:nStop)

   !DEC$ VECTOR ALWAYS
   DO n = 1, nSize
      vec2(n) = EXP( vec1(n) )
   ENDDO

ENDDO ! nb = 0, nBlocks - 1

CALL PERFOFF
```

```
CALL PERFON ('mkl-vmf')

DO nb = 0, nBlocks - 1

   nOfs   = nb * nChunk
   nSize  = MIN( nmax - nOfs , nChunk )
   nStart = nOfs + 1
   nStop  = nOfs + nSize

   vec1(1:nSize) = rearr1(nStart:nStop)

   CALL VDEXP( nSize, vec1(1:nSize), vec2(1:nSize) )

ENDDO ! nb = 0, nBlocks - 1

CALL PERFOFF
```

Figure 3: Source code example from the test code used on the Intel Nehalem CPU to investigate the vectorization behavior on Intel's Nehalem architecture. Left Panel: Double loop statement containing the intrinsic Exponential (EXP) function for vectorization using the Intel `ifort` auto-vectorizer unit. Right Panel: Loop statement containing the double precision (64 bit) vector Exponential (VDEXP) function for vectorization using the vector math function of the Intel MKL.

statement blocks which are rather complex into smaller less complex ones, one can support the compiler to vectorize a given source code statement.

2 Test Case Definition for the Performance Measurement

The test case used in the work presented here, was defined as follows. For the double precision performance measurements (Appendix B) the data vector which serves as a work data reservoir is set to a size of 2^{26} double precision vector elements. For the single precision performance measurements (Appendix C) 2^{26} single precision vector elements are used. The values of those vector elements range from 0 to $2\,\pi$. In those cases where a second input vector is required, e.g. the multiply operation, the values of this second input vector range from $\pi/2$ to $5\,\pi/2$. The size of the input vectors is chosen for two reasons. Firstly, to assure that there is enough work within the source code part for which the timing measurement is performed, so that the overhead of the used performance library PERF [D] is negligible. Secondly, a data array of such a size definitely resides in the main memory and the work data have to be fetched from there like in a data intensive case of a scientific numerical code.

For performance timing measurements we instrumented the used test bed codes by the simple but efficient PERF library (Fig. 3). The PERF library was programmed by the RZG scientist Reinhard Tisma and gives information about the time spent and the Mflop rate achieved in a detected region.

A double loop statement is used to host the intrinsic functions statement which is investigated. This follows the idea which was already sketched in the introduction where the outer loop is providing chunks of vector data and an inner loop working on those chunks (Fig. 3). The size of those data chunks can be controlled by a corresponding parameter which is changed at compile time. For our investigations this parameter is varied in steps of a factor two between 2^4 and 2^{26} to sample the full range from small to large loop sizes. The choice of vector length of 2^n assures that the work vector size is a whole-number multiple of the length of the CPU's vector pipeline (see Fig. 1).

The loop statements shown in Fig. 3 are supposed to be a general example which can be used for all cases described in this report by replacing the corresponding lines of source code. The left panel shows a double loop statement with the intrinsic Exponential (EXP) function prepared for the Intel auto-vectorizer unit by the use of the !DEC$ VECTOR ALWAYS pragma statement. The non-vectorized loop statements are realized by the use of the !DEC$ NOVECTOR and !IBM* NOVECTOR pragma statements, for Intel `ifort` and IBM XLF, respectively. Note that there is no corresponding pragma statement to !DEC$ VECTOR ALWAYS for IBM XLF. In cases of loops having no pragma statement both compilers decide whether vectorization is possible for the statement or not. Of course, the intrinsic Exponential (EXP) function being examined in Fig. 3 can be replaced by any other intrinsic function. When using the vector performance libraries, i.e. Intel MKL (Subsection 4.1), Intel IPP (Subsection 4.2), and IBM vector MASS (Section 6), the inner loop statement is replaced by the corresponding subroutine call to the vector function. In the right panel of Fig. 3 the example is shown for the double precision (64 bit) vector Exponential (VDEXP) of the vector math functions of the Intel MKL.

For the performance measurements, we request a complete quad core CPU of HPC-FF although we only use a single core. This is done to assure exclusive access to the L3 cache of the Nehalem CPU and to prevent that our measurements are influenced by other processes running on the same CPU. In a similar way, we executed the performance measurements on an IBM POWER6 CPU of VIP on a dedicated empty compute node. All measurements are repeated three times. The results shown in this report are the arithmetic average of those multiple measurements.

3 Hardware Architectures and Vectorization

3.1 Intel Xeon X5570 CPU with Nehalem Architecture

The HPC-FF (High Performance Computer for Fusion) computer cluster [E] was installed in mid of 2009 at the Jülich Supercomputing Center (JSC). HPC-FF is supposed to be a production machine for fusion simulations. Since the HLST gives high level support to users of HPC-FF, the priority target for this vectorization performance study was the hardware setup of HPC-FF.

HPC-FF is equipped with 2160 Intel Xeon X5570 quad core CPUs which are build using the Nehalem architecture in 45 nm component size. It has a basic clock rate of 2.93 GHz and a L3 cache of 8 MB. Furthermore, this processor houses 256 KB L2 cache per core and 64 KB L1 cache per core (32 KB L1 Data and 32 KB L1 Instruction). Additionally, it has a new point-to-point processor interconnect, the Intel QuickPath Interconnect, replacing the legacy front side bus, a second-level branch predictor and a second-level translation lookaside buffer [F]. The Xeon CPU is in addition equipped with vector integer/floating point registers of 128 bit total width (see Fig. 1). Whenever we are using the terms “the Intel Nehalem CPU” or “Nehalem architecture” we refer to the CPU as described above.

3.2 IBM POWER6 CPU and Architecture

In May 2008, the new IBM Power6 supercomputer was installed at the Rechenzentrum Garching [G], named VIP.

VIP is equipped with IBM POWER6 dual core server CPUs with a basic clock rate of 4.7 GHz and a 32 MB L3 cache. The CPU has 64 KB, four-way set-associative instruction cache per core and 64 KB data cache per core of an eight-way set-associative design with a two-stage pipeline supporting two independent 32-bit reads or one 64-bit write per cycle. Each core has semi-private 4 MB unified L2 cache, where the cache is assigned a specific core, but the other has a fast access to it. The POWER6 CPU is also equipped with vector integer/floating point registers of 128 bit total width (see Fig. 1). Whenever we are using the terms “the IBM POWER6 CPU” or “POWER6 architecture” we refer to the CPU as described above.

4 The Intel Fortran Compiler `ifort` and the Intel Performance Libraries

For this study we used the Intel Fortran compiler `ifort` [H] version 11.1.072 which provides an auto-vectorizer unit. This part of the compiler converts simple enough loop source code statements automatically into vectorized code sequences.

The auto-vectorizer unit can be handled by compiler flags. It is turned on by specifying the vector instruction set of the processor, e.g. `-xSSE4.2` [I] or by using one or more of the following similar compiler flags `-ax`, `-m`, `-arch`, or `-minstruction`. The vectorization behavior of single source code parts can be controlled by pragma compiler statements. A compiler report can be generated with the flag `-vec-reportN`, where `N` is a number in the range of `1` to `5` which specifies the details of the output content. Here `3` corresponds to the most verbose mode. Note that, the auto-vectorizer unit is active for an automatic optimization level higher than two, i.e. higher than `-O2`. All above mentioned switches are explained in the compiler documentation [H].

For the work presented here we use the following optimization compiler options: `-O3 -xSSE4.2 -vec-report3` (see Section 6).

Moreover we used the two Intel performance libraries, the Intel Math Kernel (MKL) library version 10.2.5.035 and the Intel Integrated Performance Primitives (IPP) Library of version 7.0.3.048. The vector functions of the two libraries were used to compare their performance to the corresponding Fortran functions forced to be non-vectorized or vectorized by the auto-vectorizer of `ifort`.

4.1 The Math Kernel (MKL) Library

The Intel Math Kernel Library (Intel MKL) is a library of highly optimized, extensively threaded math routines for science, engineering, and financial applications that require maximum performance. Core math functions include BLAS, LAPACK, ScaLAPACK, Sparse Solvers, Fast Fourier Transforms, Vector Math, and more [J].

For this work the MKL library [C] was of great interest because it provides optimized vector versions for a large set of intrinsic math functions, i.e.

```
INTERFACE
   INTEGER(KIND=4) FUNCTION ippsExp_64f(a,r,n)
   ! C Function Reference:
   ! IppStatus ippsExp_64f(const Ipp64f* pSrc, Ipp64f* pDst, int len);
   !DEC$ ATTRIBUTES C, DECORATE, ALIAS : 'ippsExp_64f':: ippsExp_64f
   IMPLICIT NONE
   INTEGER(KIND=4),INTENT(IN) :: n
   REAL    (KIND=8),INTENT(IN) :: a(n)
   REAL    (KIND=8),INTENT(OUT):: r(n)
  END FUNCTION ippsExp_64f
END INTERFACE
```

Figure 4: Example for a Fortran interface for the Intel IPP Library. Here the interface for the double precision (64 bit) exponential function is shown.

the Vector Mathematical Functions (VMF) which are part of the MKL Vector Mathematical Functions Library (VML). The library further contains vector pack/unpack and vector service functions. We do not use the vector service functions, however, a performance test using the vector pack/unpack functions is part of this work. All vector functions are described in detail in chapter 9 of the MKL manual [C].

Moreover, MKL contains include files providing interfaces for FORTRAN 77, Fortran 90 and C. For the work presented here, we used the Fortran 90 interfaces.

4.2 The Integrated Performance Primitives (IPP) Library

In this study vector instructions of the Intel Integrated Performance Primitives (IPP) Library [K] were used. The IPP is an extensive library of highly optimized software functions for multimedia, data processing, and communications applications which include video coding, signal processing, audio coding, image processing, speech coding, JPEG coding, speech recognition, computer vision, data compression, data integrity, image color conversion, cryptography/CAVP validated, string processing/regular expressions, vector/matrix mathematics, ray tracing/rendering [L].

Although, only a small fraction of its capabilities was of interest for the work presented here, namely the included vector/matrix mathematics, the IPP is of interest for additional comparisons. On the one hand IPP provides instructions which are analogous to the vector instructions of the MKL library which allows performance comparisons of the two libraries. On the other hand IPP provides some instructions which are not provided by MKL at all, such as vector versions of type conversions which offered further performance comparison with the autovectorizer unit of the `ifort` compiler.

Since the IPP is provided by Intel only as a pure C/C++ library which means that there are only C interfaces available, it was necessary to write suitable Fortran interfaces by ourselves. An include file providing the suitable Fortran 90 interfaces was developed using Intel compiler pragma statements. An example for such an interface is given in Fig. 4. There are several ways to call a C function in Fortran, e.g. using wrapper routines in C or Fortran, providing a Fortran interface, etc. Moreover, one must follow the C naming convention and pass arguments by value. This can be done by e.g. using the C ISO bindings of the Fortran 2003 standard or using compiler pragma statements.

Since we make use of an Intel library and we used the `ifort` compiler, we made the decision to provide Fortran interfaces using Intel `ifort` pragma

statements which start with !DEC$ (Fig. 4). In this example, the pragma ATTRIBUTES C is specified to pass arguments (except for arrays and characters) by value. A Fortran alias name is assigned to the C function by ATTRIBUTES ALIAS. And ATTRIBUTES DECORATE specifies that the correct prefix and suffix decorations are assigned to the function name specified by ATTRIBUTES ALIAS. This refers to the name mangling done by the Fortran compiler.

5 Vectorization and the Intel Nehalem CPU

Vectorized machine code generated by `ifort` using the auto-vectorizer unit or the vector functions of either the Intel MKL or IPP library will be executed in the vector pipelines of Intel's Nehalem architecture named SSE (SIMD streaming extension [I]). SIMD (Single Instruction, Multiple Data) denotes that instructions are only decoded once, however, if possible they are executed several times in a pipeline as described in Section 1.

5.1 Basic Run Time Behavior

The measurements show a basic structure which shows only in parts the imprint of the vectorization behavior. Large parts of the measured run time curves are dominated by the cache and bus structures of the CPU (Fig. 5). However, the exact shape of measurements for different intrinsic functions depends on the ratio of calculation operations to load/store operations. For higher ratios the influence of the cache and bus structures of the CPU will be smoothed out since the overall execution time increases and therefore the latencies of cache and memory are more and more negligible. This can be seen for e.g. the ADD operation (Fig. 10) compared to the SIN intrinsic function (Fig. 20).

For very small loop sizes (<16) the execution time is dominated by the loop overhead. Then follows a more or less distinct broad minimum in execution time which corresponds to vector sizes fitting completely into the Data Cache (L1).

With growing vector size it fits no longer completely into the respective level cache and the behavior is dominated by the cache hierarchy and the bandwidth of the bus system. This results in a step like structure where each step corresponds to certain cache level. This is emphasized in Fig. 5 by dashed black lines denoting the cache sizes of the Nehalem architecture in terms of 64 bit vector sizes.

5.2 Vectorization using the Intel `ifort` auto-vectorizer unit

In general, making use of the auto-vectorizer of a compiler is the preferred way, since no platform dependent code structures are required [2]. Moreover, in case of the Intel Fortran compiler `ifort` this statement holds as well from the performance point of view, since source code statements vectorized by the `ifort` auto-vectorizer unit show in most tested cases a better performance than the functions included in the Intel performance libraries (Tab. 1, Figs. 10 – 12, 20, and 22).

In some other cases the Intel auto-vectorizer achieves similar results compared to the Intel performance libraries (Tab. 1, Fig. 9 and 13). However, there

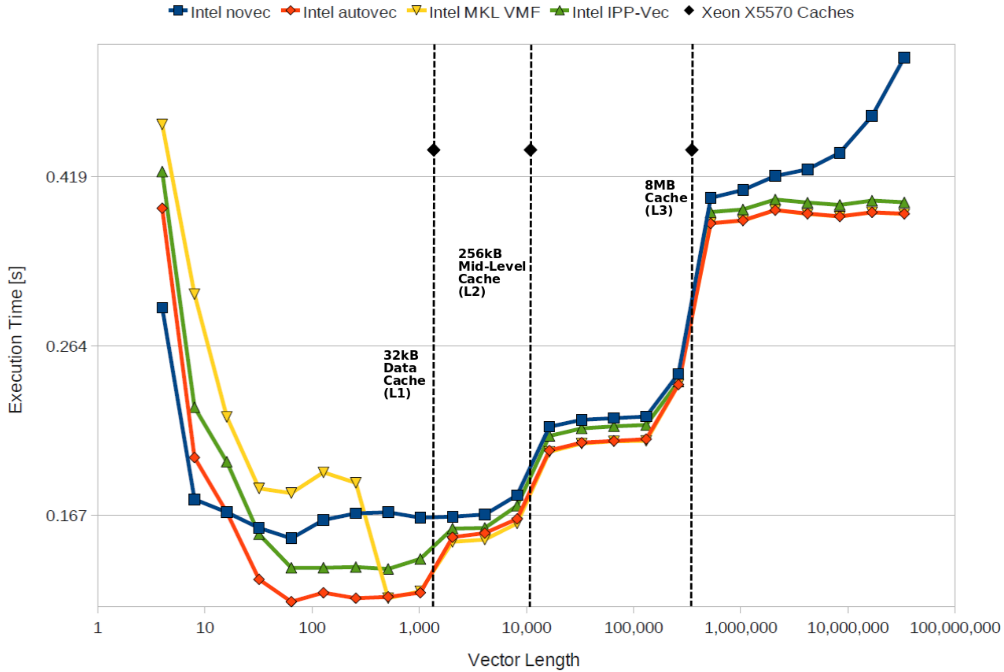

Figure 5: Run time of a MULTIPLICATION versus vector length compiled with `ifort` shown for a non-vectorized loop (blue) and loops vectorized by auto-vectorizer (red) and manually using the Vector Math Functions in the Intel MKL (yellow) and Intel IPP (green) library on a Xeon X5570 CPU of HPC-FF. The basic structure shows inefficient behavior for very small loop sizes, followed by a broad minimum in execution time for vector sizes fitting into the Data Cache (L1). For larger vector sizes the behavior is dominated by the cache hierarchy which leads to a step like structure. To emphasize this, the effective cache sizes of an Intel Xeon X5570 CPU per used vector have been plotted in terms of double precision (64 bit) as black dashed lines. For a MULTIPLICATION two argument and one result vector are used and therefore the effective cache size is one third of the actual cache size. Note that this is a log log plot.

also exist code structures where the auto-vectorizer fails to vectorize or may not achieve the expected performance (Tab. 1, Figs. 21 and 23). In those cases it is preferable to manually implement the calls to e.g. the VMF of the MKL, however, to the price of generating platform dependent code which restricts the universality of the code (see [2]). The corresponding chapter 9 of reference [2] can be found as well on the web [M].

Moreover, e.g. in the case of the MIN/MAX function the gain is almost zero for the arbitrary case (Tab. 1, Figs. 6, and 16), however, in the special case of using the MIN and MAX function to limit a single vector to a given upper and lower limit, respectively, the vectorization is counter productive (Tab. 1, Figs. 6, and 15). In those cases the vectorized statement has a larger execution time than the non-vectorized loop.

5.3 Vectorization using the Intel MKL Library

The Vector Math Functions (VMF) of Intel's Math Kernel (MKL) library don't achieve in most of the tested cases the performance of the corresponding intrinsic Fortran functions vectorized by the `ifort` auto-vectorizer unit. The only exceptions seem to be the already above mentioned SQUARE ROOT and FLOOR function shown in Figs. 21 and 23, respectively. In some cases the vector length seems to have a strong impact on the execution runtime (e.g. Figs. 10 and 22). This is discussed in greater detail in Subsection 9.1.

A vector gather operation has been tested as well, to find out about the performance of the vector packing function of the Intel MKL VML compared to gather using a loop-if statement. A vector gather operation denotes the following procedure. If a work vector contains elements which e.g. do not fulfill a conditional statement, they are removed and the remaining vector elements, which are now scattered across the former work vector are packed together into a smaller temporary work vector. This can be done by a copy process inside the conditional statement or by accessing the original vector using a mask vector which represents the conditional statement.

The results are shown in the Figs. 24 and 25. Note that we scaled the execution times of packing operations involving more than one vector packing to the number of vectors involved, e.g. the execution time of the packing operation using three vectors was divided by three and so on. As expected, the run time per packed vector for both, the loop-if statement and the MKL packing function, gets shorter the more vectors are packed using the same packing pattern. The gain for a non-vectorized and an auto-vectorized single loop statement is the same. However, this gain is larger than the one for the Intel MKL packing functions.

5.4 Vectorization using the Intel IPP Library

The Intel Integrated Performance Primitives (IPP) Library in most cases cannot compete with neither the Intel `ifort` auto-vectorizer unit nor the Intel Math Kernel (MKL) library for the presented double precision test cases. Only in very few cases the Intel IPP shows an equal (SIN, Fig. 20) or better (ADD, Figs. 10) performance than the Intel MKL, however, in those cases the performance achieved by the Intel `ifort` auto-vectorizer unit is still better. At least for the ADD operation (Figs. 10, upper panel) the `ifort` auto-vectorizer and the IPP give comparable results within a five percent range (Tab. 1).

The situation seems to be different for some cases of operations with single precision (32 bit) arguments (MULT, Fig. 26, upper panel). This is not surprising since the Intel IPP library is optimised for multimedia application which are mainly programmed in single precision. However, this seems to be not a general trend (compare the EXP function, Figs. 13 and 26, lower panel). More detailed investigations would have to be done on this subject.

5.5 Vectorization gain on Intel Nehalem

The absolute gain due to vectorization on the Intel Nehalem CPU are given for all examined intrinsic functions in Tab. 3. The relative gain is given in Tab. 2 and Fig. 6. Some of the functions show a performance loss due to vectorization,

Table 1: Table listing the fastest execution time of different function statements measured for different vectorisation methods. The overall fastest execution time are high lighted in red. If the execution time of different methods agree within an error limit of five percent the methods are considered to show an equal performance and both results are high lighted.

Function		non-vec. (sec)	auto-vec. (sec)	MKL-VMF (sec)	IPP-vec (sec)
ABS	(Fig. 9)	0.112	0.092	0.090	0.178
ADD	(Fig. 10)	0.170	0.144	0.151	0.148
ATAN2	(Fig. 11)	1.695	1.152	1.317	2.549
DIV	(Fig. 12)	0.561	0.308	0.356	0.575
EXP	(Fig. 13)	0.738	0.360	0.352	1.339
FLOOR	(Fig. 23)	0.155	0.153	0.072	0.538
INT	(Fig. 14)	0.053	0.053	—	0.178
LIM-MIN	(Fig. 15)	0.053	0.068	—	0.108
MADD	(Fig. 10)	0.367	0.213	0.231	0.226
MIN	(Fig. 16)	0.108	0.108	—	0.232
MODULO	(Fig. 19)	0.507	0.274	0.346	—
MULT	(Fig. 22)	0.157	0.132	0.133	0.224
NINT	(Fig. 14)	0.063	0.063	—	0.190
POW	(Fig. 17)	1.872	1.443	0.947	1.985
POW3O2	(Fig. 18)	0.757	0.395	0.659	4.446
REAL	(Fig. 23)	0.086	0.061	—	0.117
SIN	(Fig. 20)	1.214	0.486	0.599	0.658
SQR	(Fig. 22)	0.095	0.068	0.072	0.129
SQRT	(Fig. 21)	0.733	0.428	0.358	0.811
VPACKM	(Figs. 24, 25)	0.124	0.124	0.169	—

this is denoted by a gain factor smaller than one (Tab. 2) and a negative absolute gain (Tab. 3), respectively. The average gain factor of all examined functions — if all functions are weighted the same way — is 1.6 on the Intel Nehalem CPU (Tab. 2).

Two power functions have been examined as well (see Section 8). These are the general POW function (Figs. 6, 17 and Tabs. 3, 4) for the special case with a constant power factor of 1.5 and the special power function for a constant power factor of 1.5 named POW3O2 (Figs. 6, 18 and Tab. 3). Although, both function yield a comparable vectorization gain of roughly a factor of two (Tab. 2), the special function POW3O2 shows the absolute smaller execution time (Tab. 3) and should be preferred for that case.

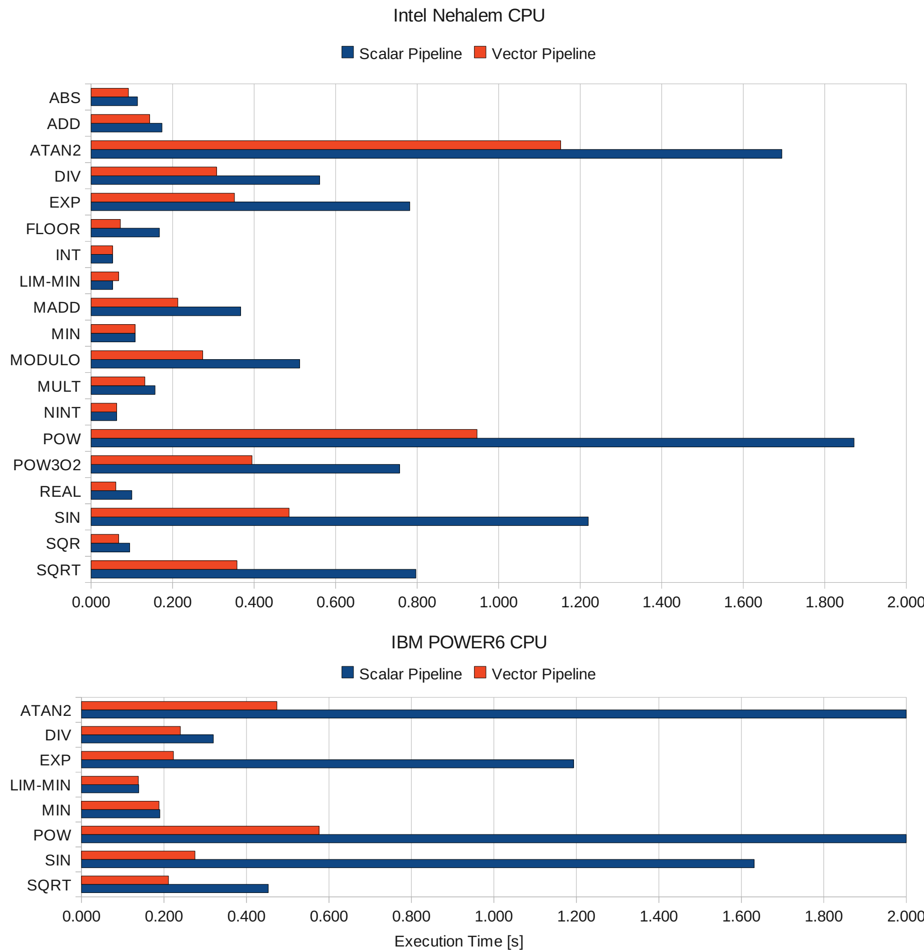

Figure 6: The best execution runtime for specific Fortran intrinsic function given in seconds for both, the scalar (blue) and vector (red) pipeline. The intrinsic function showed here exhibited their best performance typically for vector lengths in the range from 128 to 1024 elements. The Upper Panel: On the Intel CPU. Lower Panel: On the IBM POWER6 CPU. Note that the scalar execution runtimes for the ATAN2 and the POW function are exceptionally large (19.3 s and 3.7 s, respectively) and therefore exceed the shown time scale.

6 The IBM XL Fortran (XLF) compiler and the Mathematical Acceleration Subsystem (MASS) Library

For this study we used the IBM XL Fortran compiler XLF [N], version 13.1.0.3 which provides an auto-vectorizer unit, as well, which converts simple loop source code statements automatically into vectorized code sequences.

The auto-vectorizer unit is handled by compiler flags. It is put in use by switching on the corresponding option of the high-order transformations, i.e.

-qhot=vector, or by using the SIMD compiler flag, i.e. -qsimd. The -qreport switch produces a list file containing information about the optimizations performed by the compiler. It contains information about vectorized statements as well, however, unfortunately the vectorization report is not as detailed as the vectorization report of ifort. All above mentioned switches are explained in the compiler documentation [N].

However, for the work presented here we used the combination -O4 -qnoipa -qreport which includes the -qhot=vector. This combination of compiler options triggers those compiler optimizations which are comparable to the ones used for the Intel ifort compiler (see Section 4).

Mathematical Acceleration Subsystem (MASS) consists of libraries of tuned mathematical functions, available in versions for the AIX and Linux platforms. The libraries offer improved performance over the standard mathematical library routines, are thread-safe and support compilations in C, C++, and Fortran applications and are shipped with the XL C, XL C/C++, and XL Fortran compiler products [O].

The IBM MASS performance library provides optimized math functions including vector intrinsic math functions [P] which used for this work. We employed the platform optimized IBM POWER6 vector MASS library of version 6.1.0.3 which is named MASS VP6. It has to be linked using the -lmassvp6 linker flag. If possible the auto-vectorizer unit of IBM's XL Fortran compiler makes use of the math vector function in the IBM MASS library [N]. This can be clearly seen e.g. in Figs. 13 and 20 where the curves of both measurements are on top or each other.

For comparison we linked the scalar MASS library (-lmass) as well. However, if the higher optimization levels of the XLF compiler are activated [N], e.g.-O4 -qnoipa, the compiler first tries to vectorize and if not possible it tries to use the optimized scalar math functions of the MASS library. In such a case XLF automatically links with the XLOPT library which contains a copy of all IBM's optimized math functions. Hence, the measurements performed with the XLF compiler and the optimization level specified above, show no difference in performance for non-vectorized loop statements whether linked with -lmass or without (e.g. Fig. 13). One exception is the ATAN2 function which is by default linked to the standard math library libm. Only when linking with -lmass the function is replaced by the corresponding optimized scalar ATAN2 provided by the MASS (Fig. 11).

7 Vectorization and the IBM POWER6 CPU

7.1 Vectorization using the IBM XLF auto-vectorizer unit and the vector MASS library

Qualitatively, the basic run time behavior on the IBM POWER6 CPU is the same as on the Intel Nehalem CPU (see Subsection 5.1 and Fig. 5) as both are RISC (Reduced Instruction Set Computing) processors. However, there are differences arising from the hardware as e.g. cache sizes, bus bandwidths, and clock rates. In addition, the compilers themselves are not identical in their optimization strategies. Nevertheless, we tried to use similar compiler options for optimization purpose (see Secs. 4 and 6).

In some of the tested cases the performance of the vector function in IBM's MASS performance library is better than the performance of the corresponding auto-vectorized function. However, if possible the auto-vectorizer unit of XLF is making use of vector function in IBM's MASS library (see Section 6). Generally, our impression is that XLF's capabilities in detecting vectorizable source code statements is not as elaborated as of Intel's `ifort`.

The gain in execution time between a non-vectorized source code statement and one vectorized using IBM's vector MASS library is quite large in some cases, e.g. the vectorized two argument arc tangent (ATAN2) function in the vector MASS library was running nearly 41 times faster than the non-vectorized standard Fortran intrinsic function (see Section 6, last paragraph).

The lack of performance of the standard scalar ATAN2 intrinsic function on POWER6 leads to the recommendation, that whenever possible software developers should use the vector ATAN2 function of IBM's MASS library instead. Please note that there is a scalar ATAN2 intrinsic function available in the IBM MASS library which shows a better performance than the scalar ATAN2 intrinsic function included in the library XLOPT [Q].

7.2 Vectorization gain on IBM POWER6

The absolute gain due to vectorization on the IBM POWER6 architecture are given in Tab. 4 for all examined Fortran intrinsic functions. Tab. 2 and Fig. 6 show the relative gain. The average gain factor of all examined functions — if all functions are weighted the same way — is 3.3 on the IBM POWER6 CPU (Tab. 2).

Please note that, due to is exceptional large scalar execution time, the ATAN2 intrinsic function was not taken into account, when calculating the average gain.

8 Intel Nehalem vs. IBM POWER6

On the first glance, the gain due to vectorization on the IBM POWER6 CPU seems to be more pronounced compared to vectorization on the Intel Nehalem CPU. The gain of executing source code statements using the vector pipeline compared to a scalar execution is on average 50% larger than on the Intel CPU (Tab. 2 and Fig. 7). Although, the actual gain is different for the various functions, the overall picture seems to be similar for all tested functions. The gain which can be achieved on the IBM POWER6 architecture for our test examples is larger than the one on the Intel Nehalem architecture or at least on the same level (Tab. 2 and Fig. 12). However, the lowest absolute execution times and the highest vectorization gain which is achievable for the various intrinsic function on both architectures do not coincide (Fig. 6).

Two scalar intrinsic functions show an observable weaker performance on the IBM POWER6 CPU than the other functions. This are the ATAN2 (Figs. 6, 7, 11 and Tabs. 3, 4) function which is executed more than 11 times faster on the Nehalem architecture and the POW function with a constant argument of 1.5 (Figs. 6, 7, 17 and Tabs. 3, 4) which is executed nearly twice as fast on Nehalem. However, the corresponding vectorized intrinsic functions show an opposed behavior. The vector ATAN2 of IBM's MASS library shows a tremendous vectorization gain of 41 resulting in the fact that the vector ATAN2 of IBM's

Table 2: Table listing the gain factors between the execution time of different function statements in the scalar and the vector pipeline of an Intel Xeon 5570 and an IBM POWER6 CPU, respectively. For comparison reasons the ratio of the scalar execution times of the same statement on the Intel Nehalem CPU and an IBM POWER6 CPU, respectively, is shown in the last column.
We defined the vectorization gain shown in this table to be the ratio of scalar to vector execution time corresponding to the vector length which yields the overall smallest execution time (Tabs. 3 and 4)
Please note that, the average gain factors given in the last line are calculated excluding the ATAN2 function (see Subsection 7.1).

Function		HPC-FF (Nehalem)	VIP (POWER6)	HPC-FF/VIP scalar exec.
ABS	(Fig. 9)	1.26	—	—
ADD	(Fig. 10)	1.21	—	—
ATAN2	(Fig. 11)	1.49	41.0	0.09
DIV	(Fig. 12)	1.82	1.33	1.75
EXP	(Fig. 13)	2.22	5.35	0.66
FLOOR	(Fig. 23)	2.33	—	—
INT	(Fig. 14)	1.00	—	—
LIM-MIN	(Fig. 15)	0.75	1.01	0.38
MADD	(Fig. 10)	1.72	—	—
MIN	(Fig. 16)	1.00	1.01	0.57
MODULO	(Fig. 19)	1.87	—	—
MULT	(Fig. 22)	1.19	—	—
NINT	(Fig. 14)	1.00	—	—
POW	(Fig. 17)	1.92	6.37	0.51
POW3O2	(Fig. 18)	2.03	—	—
REAL	(Fig. 23)	1.64	—	—
SIN	(Fig. 20)	2.51	5.94	0.75
SQR	(Fig. 22)	1.40	—	—
SQRT	(Fig. 21)	2.23	2.15	1.76
VPACKM	(Figs. 24, 25)	1.00	—	—
AVERAGE	(excl. ATAN2)	1.60	3.31	

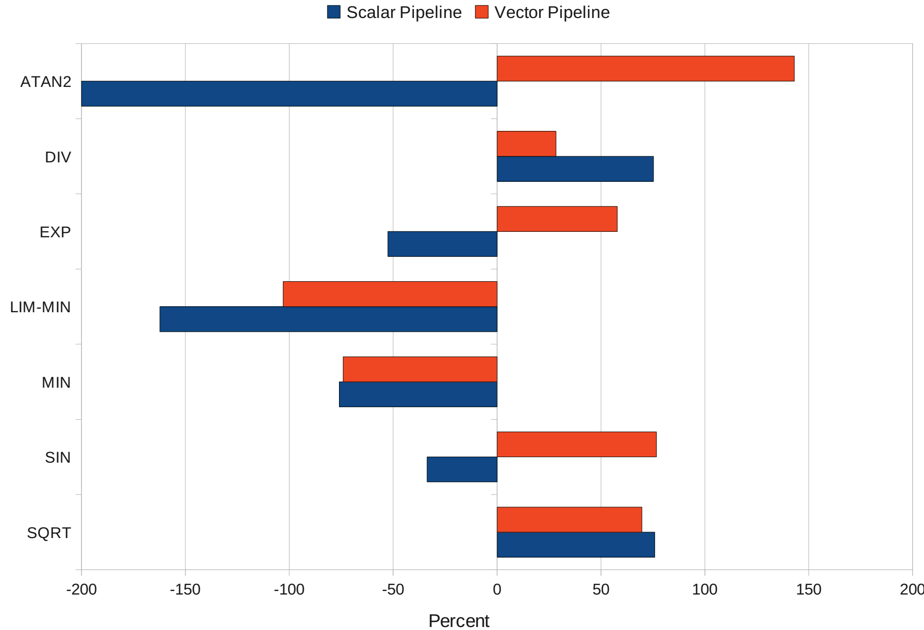

Figure 7: Special bar diagram indicating the difference of execution runtime of specific intrinsic functions measured on the IBM POWER 6 CPU compared to the same execution runtime measured on the Nehalem CPU of HPC-FF. The difference in this special bar diagram is given in percent for both, the scalar (blue) and vector (red) pipelines. We define that -100 percent denotes that a specific statement was executed with the double execution runtime on IBM POWER 6 compared to Intel Nehalem $[100 * (t_{\text{Intel}} - t_{\text{IBM}})/t_{\text{IBM}}]$, whereas +100 percent denotes that a statement in half the execution runtime $[100 * (t_{\text{IBM}} - t_{\text{Intel}})/t_{\text{Intel}}]$. Note that the scalar performance of the ATAN2 function on IBM's POWER6 compared to Intel's Nehalem is exceptionally weak (-1045%) and therefore exceed the shown scale.

MASS is around twice as fast as the vectorized ATAN2 function on Nehalem. The special case of the POW function shows a vectorization gain of 6.4, so that the vectorized version is executed 1.6 times faster on IBM POWER6. However, one should mention that the special case having a constant power factor of 1.5 is implemented in a separate intrinsic function named POW3O2 (Figs. 6, 18 and Tab. 3) in the Intel MKL which shows a comparable or slightly better performance (see Subsection 5.5) than the POW function of IBM's MASS library.

All in all, we find that there is no clear trend in the performance of the investigated function on IBM POWER6 compared to Intel Nehalem (Fig. 7). For two out of seven compared functions the performance of both, the vector as well as the scalar function is better on IBM POWER6 (DIV Figs. 12 and SQRT Figs. 21). For another two functions it is the other way round (LIM-MIN Figs. 15 and MIN Figs. 16). Three functions show a weaker scalar performance on

IBM POWER6, however, the vectorized corresponding functions show a better performance than on Intel Nehalem (ATAN2 Figs. 11, EXP Figs. 13, and SIN Figs. 20).

Therefore, no statement can be made whether POWER6 or Nehalem exhibits a higher absolute vector performance. Instead this report is just meant to be a practical guideline for software developers where to put the focus when trying to exploit the vectorization potentials of scientific numeric codes on one of the two architectures covered by this report. The summarized behavior of the investigated intrinsic functions can be found in Tabs. 1, 2 and Figs. 6, 7. The details of the measurement are expressed in Appendix B and C.

9 Issues of Vectorization

In this chapter we list some special vectorization issues of the compilers and architectures used for this study. All listed issues are part of the experience we gained with this study. Therefore, we make no claim for this chapter to be complete.

9.1 Impact of the vector length on the execution runtime

In some cases the vector length seems to have a strong impact on the execution runtime on Intel Nehalem (e.g. Figs. 10 and 22).

To test this in detail we sampled the vector length in small and equidistant steps. The result of this measurement for the MULTIPLICATION operation is shown in Fig. 8.

For most of the investigated vector length the MKL vector MULTIPLICATION (VDMUL) shows the least performance of all compared vectorized statements. Only near vector lengths which correspond to multiples of 512 (2^9) is the performance comparable to the auto-vectorized loop statement. This behavior can be found for all operations directly performed in registers, i.e. the ADD, the MULT, and the SQR operation. Hence in the case of those operations the user has to take special care about the actual vector length.

9.2 Unsupported Formats of the `ifort` Auto-Vectorizer

A common known example of an unsupported format of the `ifort` auto-vectorizer are double precision complex (`COMPLEX*16`) variables [R]. They are treated by the CPU in the same way as quad precision real (`REAL*16`) variables. They cannot be processed by the hardware vector register of Intel's Nehalem architecture since it has just one type of floating point register serving as scalar or vector register, respectively [F]. This register has a total width of 128 bits in Nehalem. For this reason it can hold two double precision real variables, but only one quad precision real variable and one double complex variable, respectively. Therefore, the auto-vectorizer does not vectorize statements including double precision complex variables [R].

9.3 IBM XL Fortran SIMD Functions on POWER6

Besides the vector math functions included in the IBM MASS library, IBM's XL Fortran provides a number of different additional vector functions for all

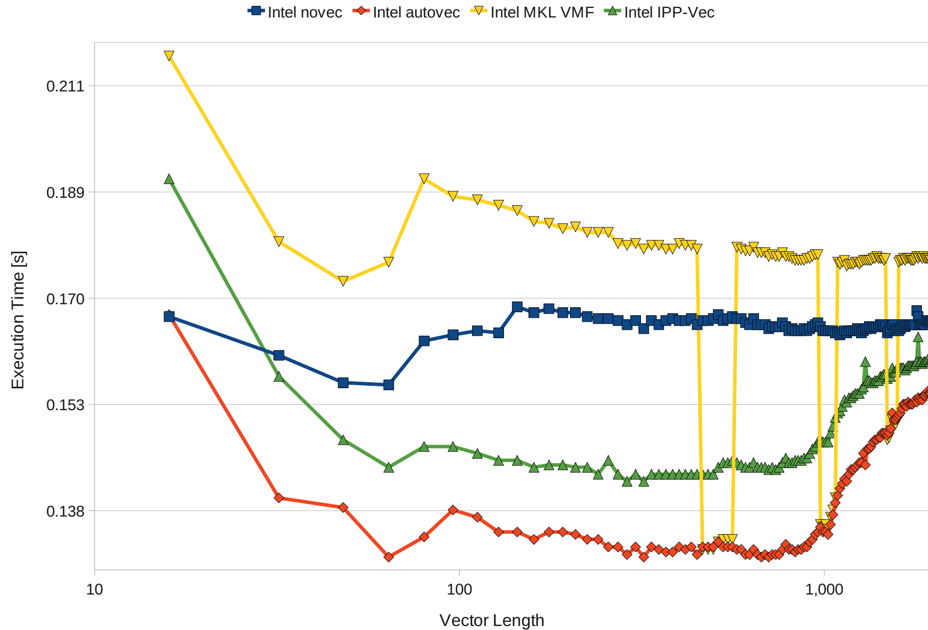

Figure 8: Run time of a MULTIPLICATION versus vector length compiled with `ifort` on a Xeon X5570 CPU of HPC-FF shown for a non-vectorized loop (blue) and loops vectorized by the auto-vectorizer (red) and manually using the Vector Math Functions in the Intel MKL (yellow) and IPP (green) library. The curve is measured for vector length of 16 (2^4) to 2048 (2^{11}) in steps of 16 (see Fig. 22, upper panel).

kinds of operations (math, conversion, etc.), the so called Vector Multimedia eXtension (VMX) and the Vector Scalar eXtension (VSX) intrinsic functions [S]. These functions use XL Fortran intrinsic derived types, e.g. the `VECTOR FLOAT` type to work with Fortran `REAL*4` type vectors.

However, on the IBM POWER6 architecture the VSX instruction set has only been partly implemented under the name of AltiVec [S]. This leads to the constraint that the 128 bit vector unit of AltiVec can be subdivided into 16x8, 8x16, and 4x32 bit elements [T], but not into 2x64 bit (see Fig. 1, left panel). Thus the AltiVec instruction set of XLF does not support, in contrast to the vector MASS library, double precision real types. Intrinsic functions that use or result in the INTEGER(8), UNSIGNED(8), or REAL(8) vector types require an architecture that supports the VSX instruction set extensions, such as POWER7 [S].

10 Summary and Conclusions

The most important question a reader of this report may have in mind is if her/his scientific numerical code can benefit from vectorisation. We tried to give an answer on that using a conservative approach. In the test case used

(Section 2) we assumed that all data are originally located in the main memory and that they have to be transferred through the cache hierarchy to perform one operation on this data.

In a real case this situation is most likely a worst case scenario. The amount of overhead introduced due to the data transfer from the main memory is probably smaller than in our test case. So the numbers for specific operations given in this report are supposed to be a conservative estimate of how much one can gain by vectorizing specific operation in a scientific numerical code. However, one should keep in mind that it is not always possible to vectorize single operations without vectorizing a whole or large parts of a subroutine, as well. This can make restructuring of the code unavoidable.

For our investigation on vectorization performance we used the Intel Xeon X5570 CPU of HPC-FF at JSC based on Intel's Nehalem architecture (Subsection 3.1) together with the Intel Fortran compiler `ifort` (v11.1.072) and the Intel MKL (v10.2.5.035) and IPP (v7.0.3.048) performance libraries. For the Intel `ifort` compiler we used the `-O3 -xSSE4.2` optimization flags (Section 4). Additionally, the vectorization performance of specific intrinsic functions was tested on the IBM POWER6 CPUs (Subsection 3.2) of VIP at RZG using the IBM XL Fortran compiler (v13.1.0.3) and IBM's MASS library (v6.1.0.3) described in Section 6. For the IBM XLF compiler we used the optimization flags `-O4 -qnoipa` (Section 6).

The work presented in this report results in following statements:

- In total the auto-vectorizer unit of the Intel `ifort` compiler version 11.1.072 used for this study does a better job vectorizing the intrinsic functions being under consideration than both, the Intel MKL (v10.2.5.035) and IPP (v7.0.3.048) performance libraries. Therefore, using the auto-vectorizer should be preferred. In addition this approach does not produce platform specific source code and it requires less programming efforts by the software developer.

  However, in a few cases the vector functions of the Intel MKL show a better performance. In those cases the MKL provides an alternative if pure performance is required.

- The Intel IPP library of version 7.0.3.048 cannot be recommended for our double precision (64 bit) test cases, since it shows a performance which is in all cases less or merely equal to the performance achieved by the Intel `ifort` auto-vectorizer unit and Intel MKL library, respectively.

- In almost all tested cases the performance of the scalar functions linked with and without the scalar MASS library, respectively, is the same. As well as the performance of the functions vectorized by the auto-vectorizer unit of XLF and the functions of the vector MASS library respectively, is the same. This is the case because depending on the optimization level, the IBM XLF compiler uses the math functions from the XLOPT library which includes amongst others identical math functions as the MASS library. The exception in this study is the is the ATAN2 function (Fig. 11).

- The IBM POWER6 architecture yield for most of the investigated functions the larger vectorization gain factors (3.2 in average) compared to the Intel Nehalem architecture (1.5 in average). Altogether, both architectures show a comparable absolute performance, however, each has its strength and weaknesses.

- IBM's XL Fortran compiler seems to be not as effective in analyzing the vectorization potential of complexer source code structures as Intel's Fortran compiler `ifort`.

  If pure performance is required in those cases the vector MASS intrinsic functions have to be implemented manually. It assures that XLF is making use of the full vectorization potentials of a specific numerical code on POWER6. However, doing so generates platform dependent source code and it requires a larger amount of programming efforts.

- The length of the work vector when using the so called "strip mining" should fulfill the following requirements. The length must be large enough to make the loop overhead negligible (i.e. around 32 in our test case) and it must be small enough that all data work vectors fit into the L1 cache of the CPU (i.e. around 1024 in our test case). Furthermore, the length should be a power of two (i.e. $2^n, n \in \mathbb{N}$).

Generally, to determine whether a scientific numerical code will benefit from vectorization a number of steps are required. First the code needs to be profiled to identify the "hot spots" of the code. Then the software developer has to review the fraction of computationally intensive intrinsic functions in the "hot spots" of the code. For this especially Fig. 6 should give a guideline.

However, it has to be noted that using optimized and vectorized math function leads to a slightly reduced accuracy of the function results (e.g. see [U] and [V]).

Acknowledgments

The author likes to thank Roman Hatzky from the HLST Core Team, Max-Planck-Institut für Plasmaphysik (IPP) for a lot of fruitful discussions. Furthermore, the author likes to thank Ingeborg Weidl from the Operation of High Performance Computers team, Rechenzentrum Garching (RZG) for providing exclusive access to computing resources on the VIP machine and Markus Rampp from the Application support for High Performance Computers team, RZG for useful discussions.

References

[1] M. Flynn, *IEEE Trans. Comput.* **C-21**, 948+ (1972).

[2] A. J. C. Bik, *Software Vectorization Handbook, The: Applying Intel Multimedia Extensions for Maximum Performance*, Intel Press, 2004.

Online References

[A]. Wikipedia Arcticle: Flynn's Taxomony
`http://en.wikipedia.org/wiki/Flynn's˙taxonomy`

[B]. Wikipedia Arcticle: Streaming SIMD Extensions
`http://en.wikipedia.org/wiki/Streaming˙SIMD˙Extensions`

[C]. Intel® Math Kernel Library for Linux* OS User's Guide, Chapter: Vector Mathematical Functions
`http://software.intel.com/sites/products/documentation/hpc/compilerpro/en-us/fortran/lin/mkl/refman/index.htm`

[D]. Rechenzentrum Garching: Simple Performance Toolkit
`http://www.rzg.mpg.de/computing/hardware/power4/perf˙c.html`

[E]. The High Performance Computer for Fusion (HPC-FF) at Jülich Supercomputing Centre (JSC)
`http://www.fz-juelich.de/jsc/juropa`

[F]. Wikipedia: Nehalem (microarchitecture)
http://en.wikipedia.org/wiki/Nehalem˙(microarchitecture)

[G]. Max-Planck-Gesellschaft & IPP – Rechenzentrum Garching: The IBM Power6 System (VIP)
http://www.rzg.mpg.de/computing/hardware/Power6/the-ibm-power6-system

[H]. Intel® Fortran Compiler User and Reference Guides, Chapter: Automatic Vectorization Overview
http://software.intel.com/sites/products/documentation/hpc/compilerpro/en-us/fortran/mac/compiler˙f/index.htm

[I]. Intel® Software Network - Articles: Intel® compiler options for SSE generation (SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX) and processor-specific optimizations
http://software.intel.com/en-us/articles/performance-tools-for-software-developers-intel-compiler-options-for-sse-generation-and-processor-specific-optimizations/

[J]. Intel®Math Kernel Library: The Flagship High Performance Computing Math Library for Windows, Linux, and Mac OS X
http://software.intel.com/en-us/articles/intel-mkl

[K]. Intel® Integrated Performance Primitives for Intel® Architecture, Reference Manual, Volume 1: Signal Processing, Chapters: Essential Functions, Fixed-Accuracy Arithmetic Functions
http://software.intel.com/sites/products/documentation/hpc/ipp/ipps/ipps˙ch5/ch5˙Intro.html

[L]. Intel®Integrated Performance Primitives Performance Library: Multicore Power for Multimedia and Data Processing
http://software.intel.com/en-us/articles/intel-ipp

[M]. Aart J. C. Bik, The Software Vectorization Handbook, Chapter 9, Intel® Press 2004
http://software.intel.com/sites/products/documentation/hpc/compilerpro/en-us/fortran/lin/mkl/refman/index.htm

[N]. Language Reference of XL Fortran for AIX, V13.1
http://publib.boulder.ibm.com/infocenter/comphelp/v111v131/index.jsp?noscript=1

[O]. Mathematical Acceleration Subsystem: Libraries of high-performance mathematical functions
http://www-01.ibm.com/software/awdtools/mass

[P]. MASS vector libraries for AIX – Overview
http://www-01.ibm.com/support/docview.wss?uid=swg27018490

[Q]. Performance information for the MASS libraries for AIX
http://www-01.ibm.com/support/docview.wss?uid=swg27005374

[R]. Intel® Software Network: Vectorization – Autovectorizing complex*16 data type
http://software.intel.com/en-us/forums/showthread.php?t=70027&o=d&s=lr

[S]. XL Fortran for AIX, V13.1: Language Reference – Vector intrinsic procedures (IBM extension)
http://publib.boulder.ibm.com/infocenter/comphelp/v111v131/topic/com.ibm.xlf131.aix.doc/language˙ref/vmxintrinsics.html

[T]. Wikipedia: AltiVec
http://en.wikipedia.org/wiki/AltiVec

[U]. Measured Accuracy of VML Functions
http://laplace.phas.ubc.ca/C2/intel/mkl/functions/˙accuracyall.html

[V]. Accuracy information for the MASS libraries for AIX
http://www-01.ibm.com/support/docview.wss?uid=swg27007004

A Summary of Vector Statement Execution Times

Table 3: Table listing the execution times of different function statements in the scalar and the vector pipeline of an Intel Xeon 5570 CPU corresponding to the vector length which yields the overall smallest execution time in our measurements.

Function		scalar pipeline (sec)	vector pipeline (sec)	difference (sec)
ABS	(Fig. 9)	0.114	0.092	0.022
ADD	(Fig. 10)	0.174	0.144	0.030
ATAN2	(Fig. 11)	1.695	1.152	0.543
DIV	(Fig. 12)	0.561	0.308	0.253
EXP	(Fig. 13)	0.782	0.352	0.430
FLOOR	(Fig. 23)	0.168	0.072	0.096
INT	(Fig. 14)	0.053	0.053	0.000
LIM-MIN	(Fig. 15)	0.053	0.068	-0.015
MADD	(Fig. 10)	0.367	0.213	0.154
MIN	(Fig. 16)	0.108	0.108	0.000
MODULO	(Fig. 19)	0.512	0.274	0.238
MULT	(Fig. 22)	0.157	0.132	0.025
NINT	(Fig. 14)	0.063	0.063	0.000
POW	(Fig. 17)	1.872	0.947	0.925
POW3O2	(Fig. 18)	0.757	0.395	0.362
REAL	(Fig. 23)	0.100	0.061	0.039
SIN	(Fig. 20)	1.220	0.486	0.734
SQR	(Fig. 22)	0.095	0.068	0.027
SQRT	(Fig. 21)	0.797	0.358	0.439
VPACKM	(Figs. 24, 25)	0.124	0.124	0.000

Table 4: Table listing the absolute minimal execution time of different function statements in the scalar and the vector pipeline of an IBM POWER6 CPU corresponding to the vector length which yields the overall smallest execution time in our measurements.

Function		scalar pipeline (sec)	vector pipeline (sec)	difference (sec)
ATAN2	(Fig. 11)	19.41	0.474	18.93
DIV	(Fig. 12)	0.320	0.240	0.080
EXP	(Fig. 13)	1.193	0.223	0.970
LIM-MIN	(Fig. 15)	0.139	0.138	0.001
MIN	(Fig. 16)	0.190	0.188	0.002
POW	(Fig. 17)	3.668	0.576	3.092
SIN	(Fig. 20)	1.631	0.275	1.356
SQRT	(Fig. 21)	0.453	0.211	0.242

B Double Precision Performance Measurement

In the following only performance curves for intrinsic functions using double precision, i.e. 64 bit arguments are collected (see Tab. 2). Note that, all plots show the average of three independent measurements and all are log log plots.

The plots are organized as follows. For those functions which are available for the Intel Nehalem architecture as well as for the IBM POWER6 architecture, the plot panel shows the results on Intel in the upper panel and those on IBM in the lower panel. The plot panels which show the results of functions only available on Intel are combined arbitrarily, however, as far as possible showing two plots per panel.

Please note that in most of the plots showing measurements on the IBM POWER6 CPU, the brown line which denotes the functions of the scalar MASS library is lying underneath the blue line showing the scalar functions of the standard math library `libm`.

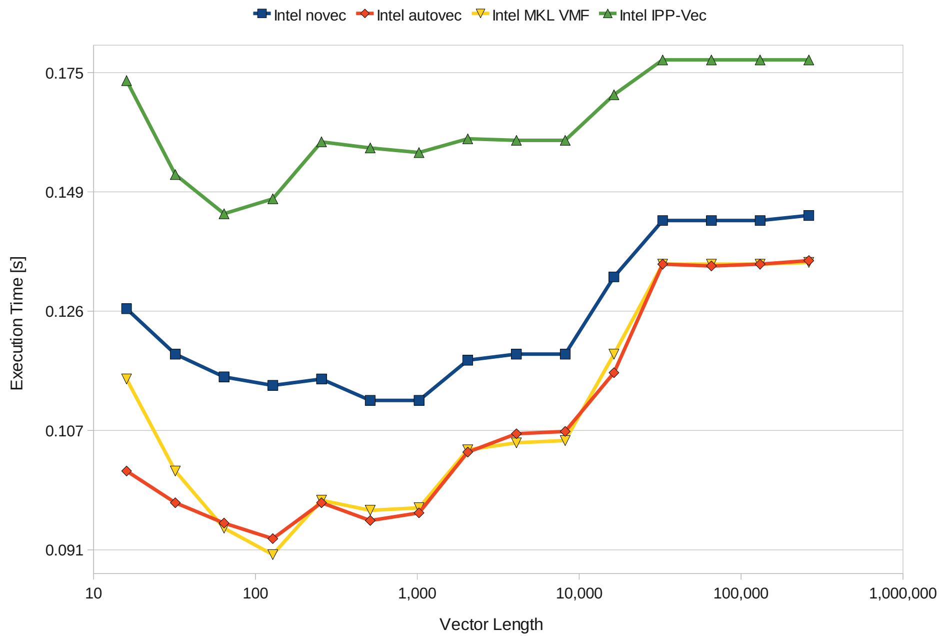

Figure 9: Run time of the Fortran intrinsic ABSOLUTE (ABS) function versus vector length compiled with `ifort` on a Xeon X5570 CPU of HPC-FF shown for a non-vectorized loop (blue) and loops vectorized by the auto-vectorizer (red) and manually using the Vector Functions of the Intel MKL (yellow) and IPP (green) library.

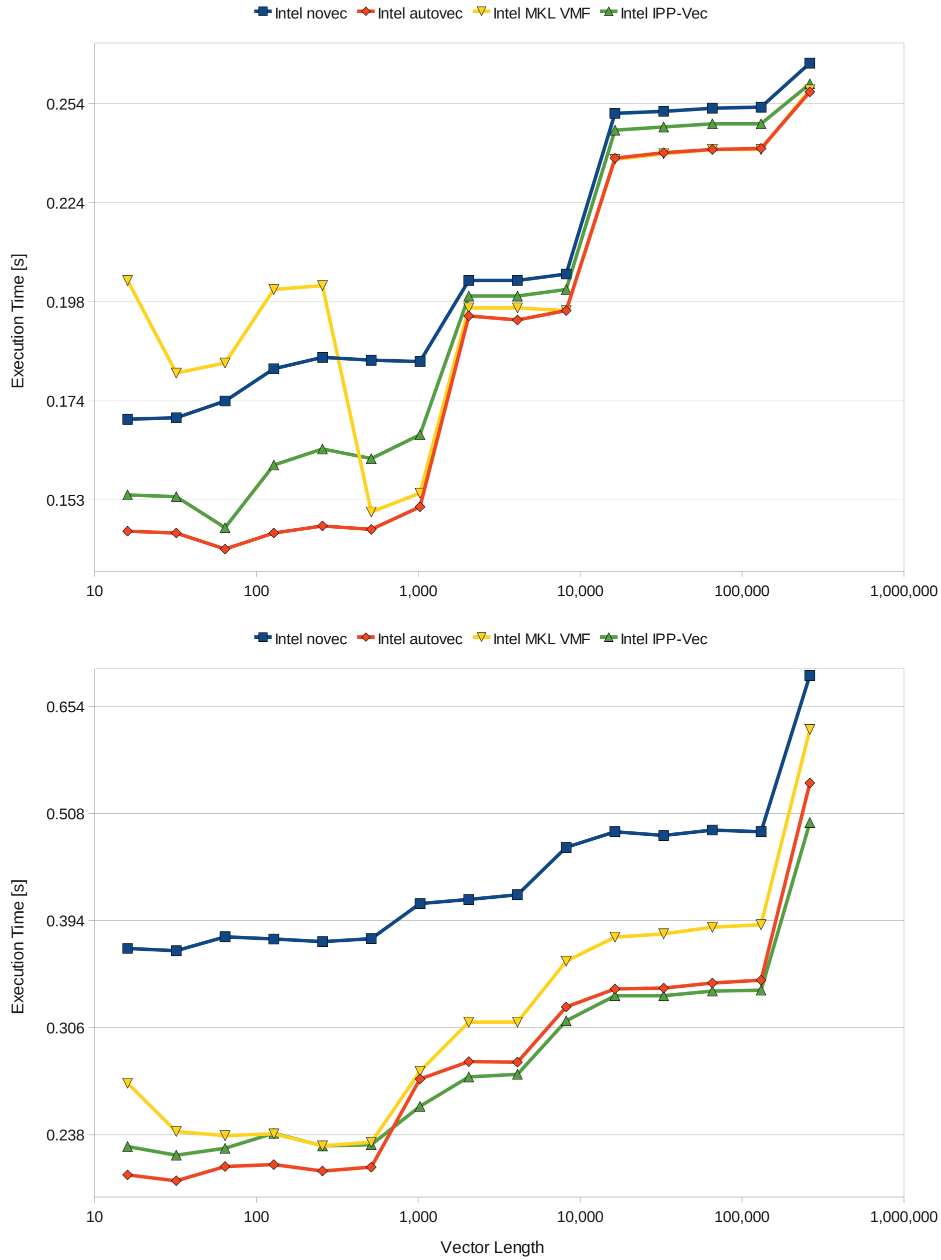

Figure 10: Run time of the Fortran intrinsic function versus vector length on a Xeon X5570 CPU of HPC-FF using `ifort` shown for a non-vectorized loop (blue) and loops vectorized by the auto-vectorizer (red) and manually using the Vector Functions of the Intel MKL (yellow) and IPP (green) library. Upper Panel: The ADD function. Lower Panel: The fused MULTIPLY ADD operation.

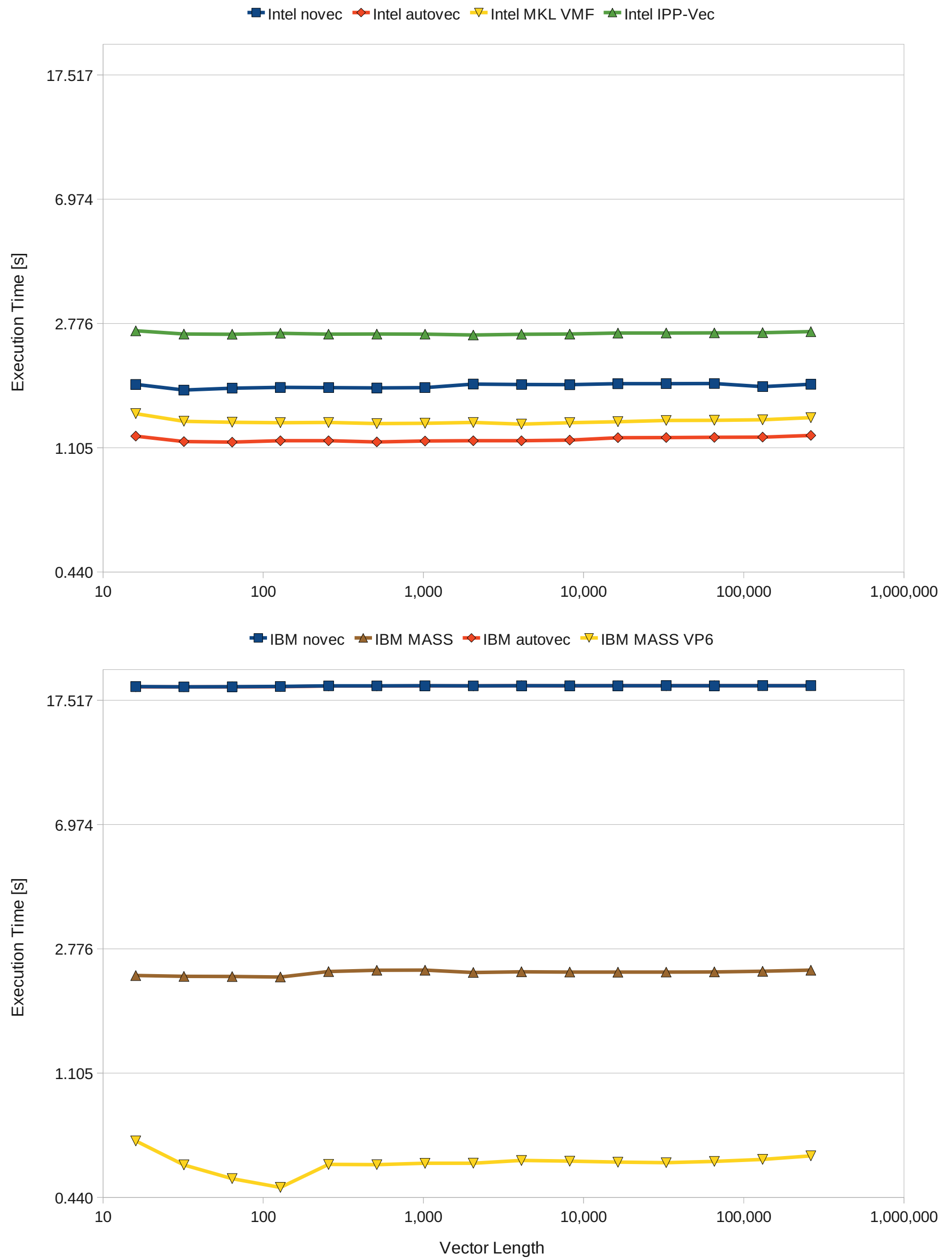

Figure 11: Run time of the Fortran intrinsic 2 ARGUMENT ARC TANGENT (ATAN2) function versus vector length shown for a non-vectorized loop (blue, brown) and loops vectorized by the auto-vectorizer (red) and manually (yellow). Upper Panel: Using `ifort` and the Vector Math Functions in the Intel MKL library on a Xeon X5570 CPU of HPC-FF. Lower Panel: Using XLF and the Vector Functions in the IBM MASS VP6 library on an IBM POWER6 CPU of VIP. Note that the blue line is lying on top of the red line.

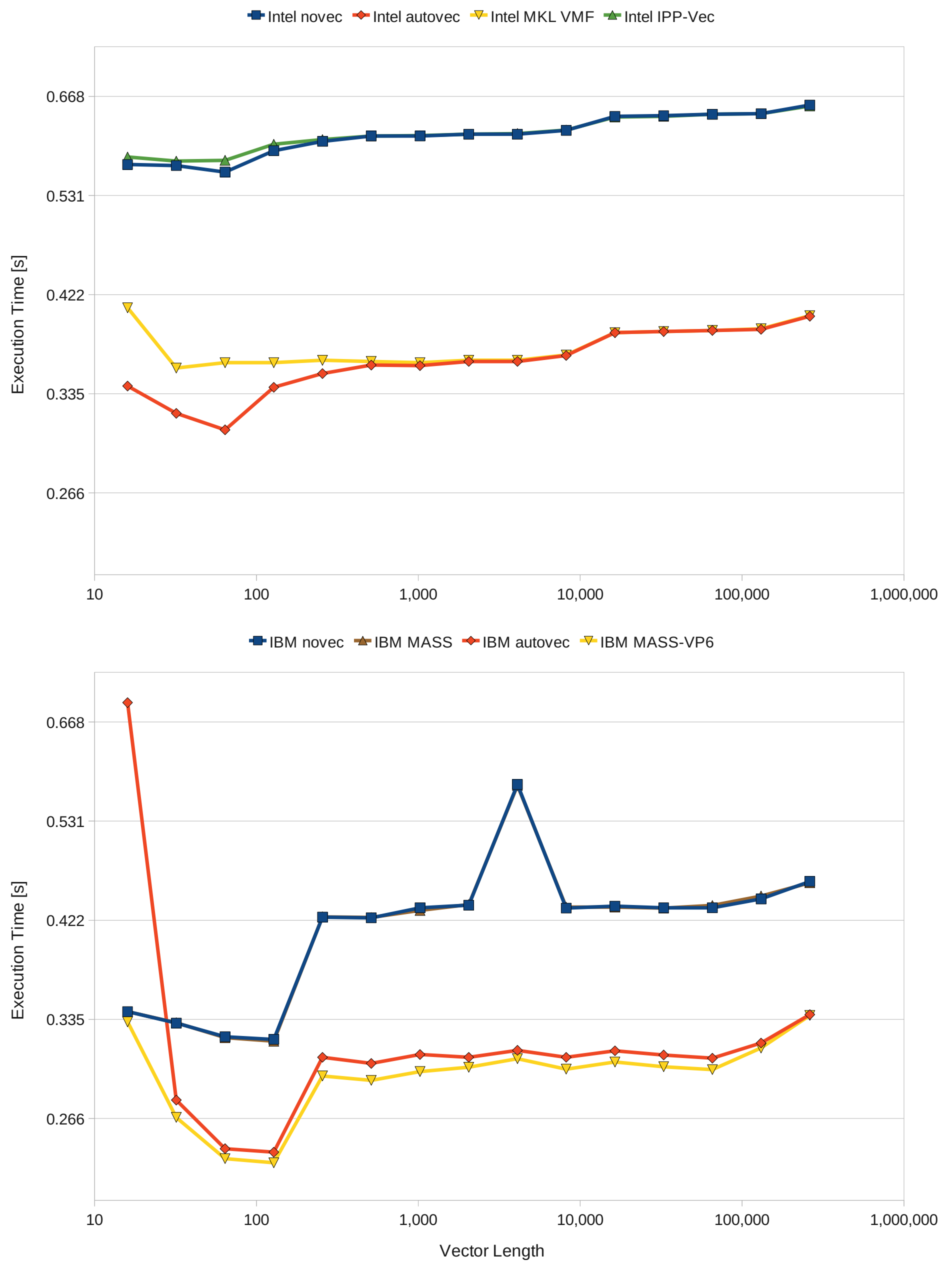

Figure 12: Run time of a DIVISION operation versus vector length shown for a non-vectorized loop (blue, brown) and loops vectorized by the auto-vectorizer (red) and manually (yellow, green). Upper Panel: Using `ifort` and the Vector Functions in the Intel MKL (yellow) and IPP (green) library on a Xeon X5570 CPU of HPC-FF. Lower Panel: Using XLF and the Vector Functions in the IBM MASS VP6 library on an IBM POWER6 CPU of VIP.

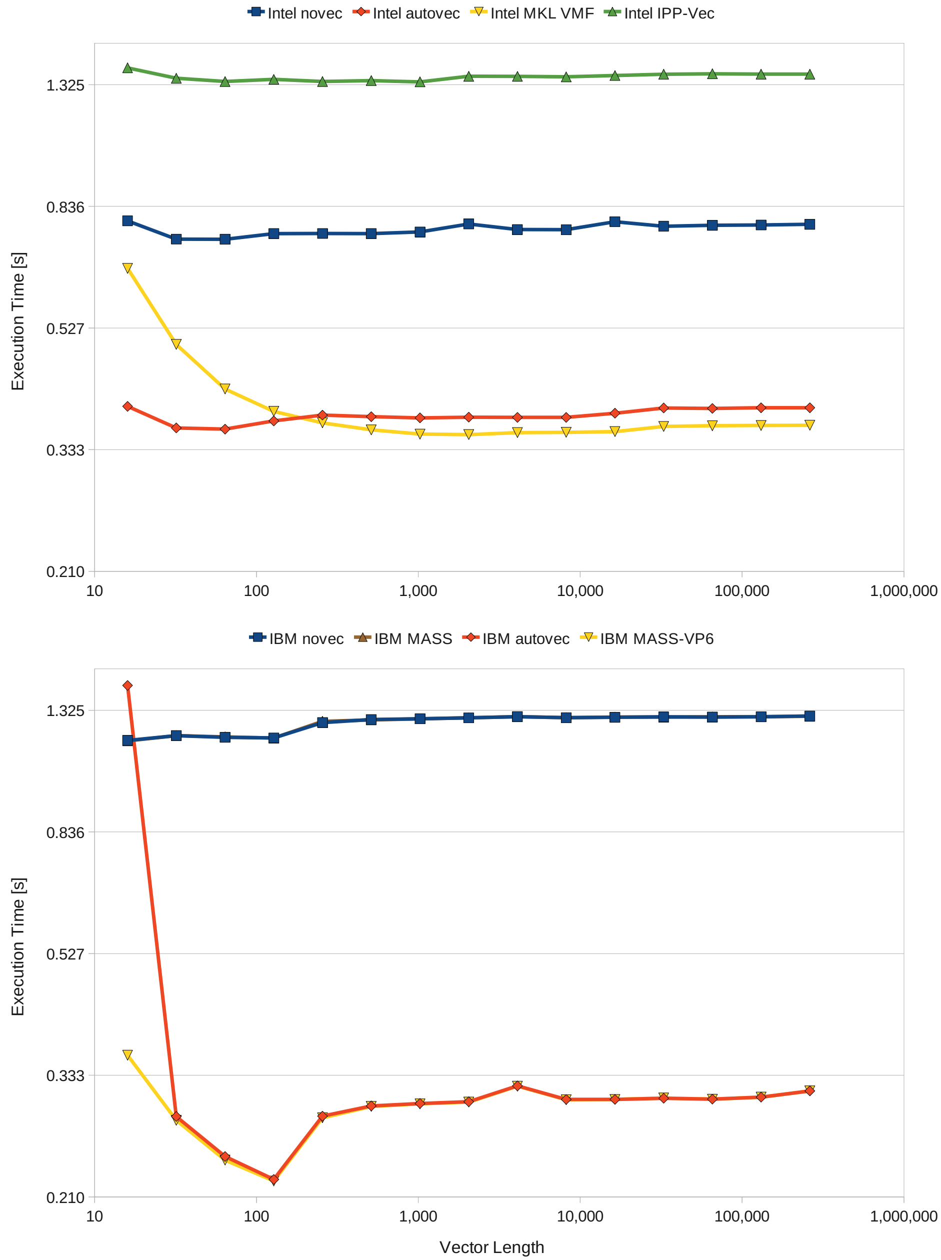

Figure 13: Run time of the Fortran intrinsic EXPONENTIAL (EXP) function versus vector length shown for a non-vectorized loop (blue, brown) and loops vectorized by the auto-vectorizer (red) and manually (yellow, green). Upper Panel: Using `ifort` and the Vector Math Functions in the Intel MKL and Intel IPP library on a Xeon X5570 CPU of HPC-FF. Lower Panel: Using XLF and the Vector Functions in the IBM MASS VP6 library on an IBM POWER6 CPU of VIP.

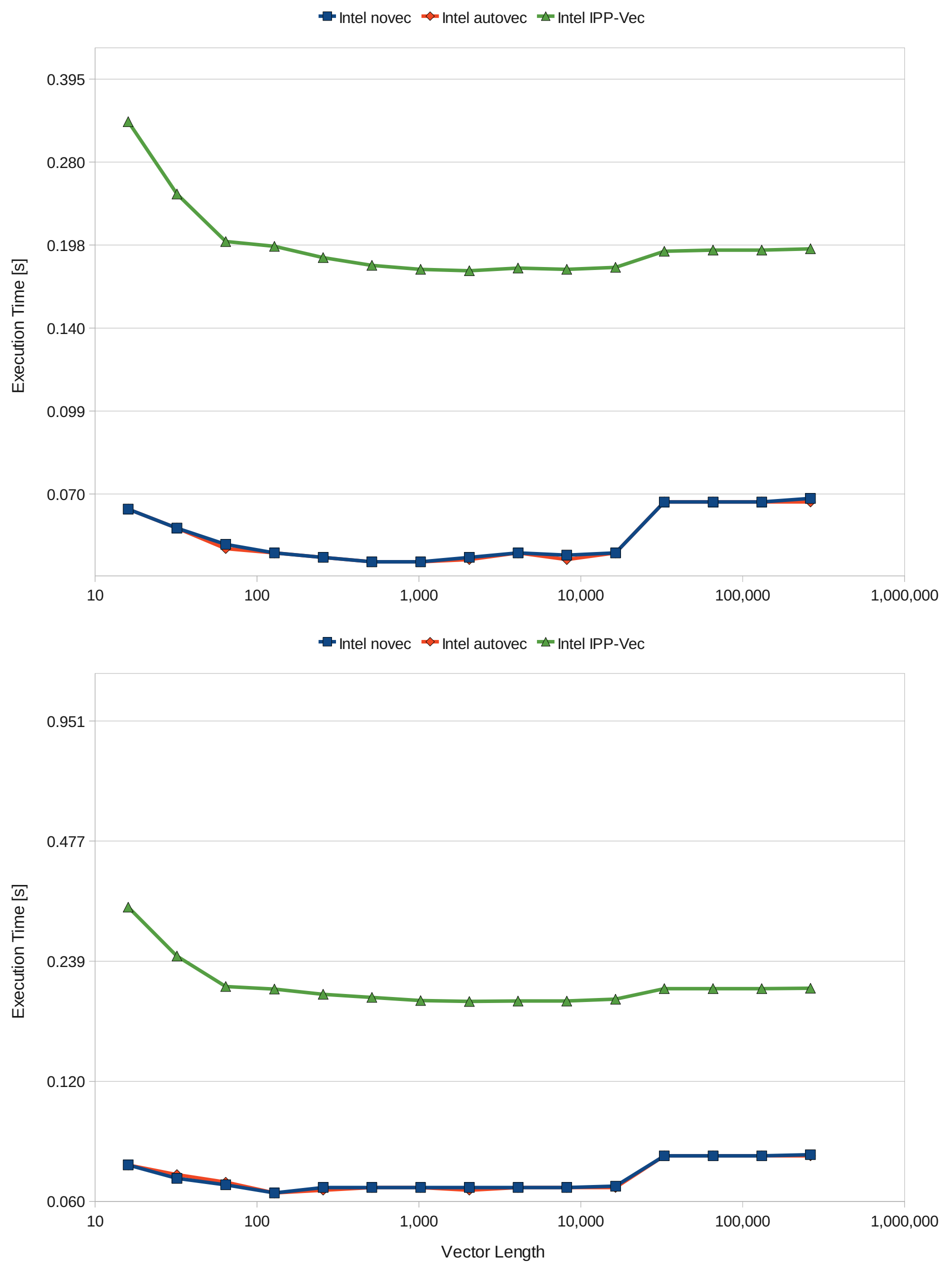

Figure 14: Run time of FORTRAN intrinsic functions versus vector length on a Xeon X5570 CPU of HPC-FF compiled with `ifort` shown for a non-vectorized loop (blue) and loops vectorized by the auto-vectorizer (red) and manually using the Vector Functions in the Intel IPP library(green). Upper Panel: CONVERT TO INTEGER (INT) function. Lower Panel: CONVERT TO NEAREST INTEGER (NINT) function.

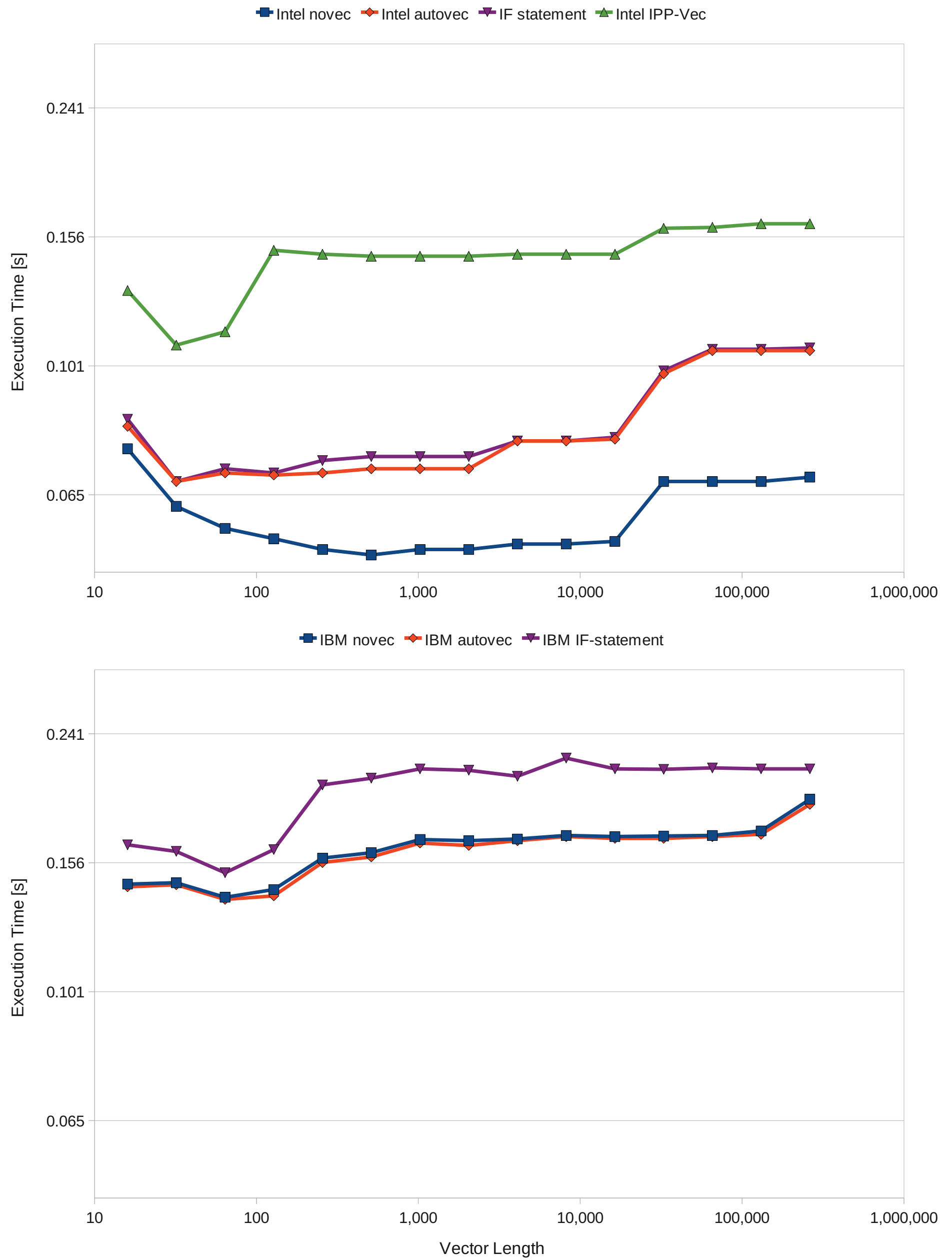

Figure 15: Run time of the Fortran MIN intrinsic function versus vector length using the min function to limit one vector to a given upper limit shown for a non-vectorized loop (blue) and loops vectorized by auto-vectorizer (red), manually (green), and an if-statement performing the same operation vectorized by the auto-vectorizer (magenta). Upper Panel: Using `ifort` and the Vector Math Functions in the Intel IPP library on a Xeon X5570 CPU of HPC-FF. Lower Panel: Using XLF and the Vector Functions in the IBM MASS VP6 library on an IBM POWER6 CPU of VIP.

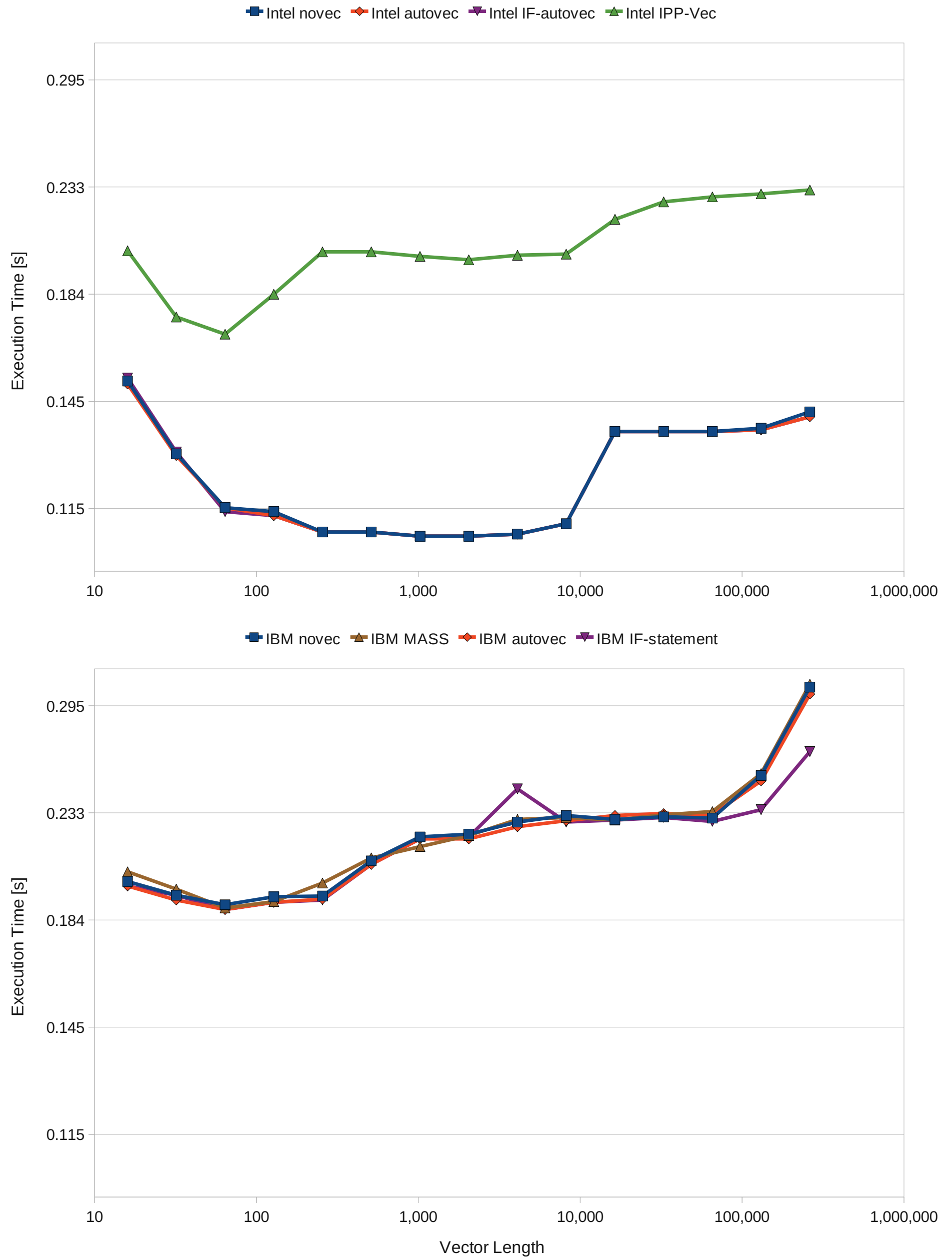

Figure 16: Run time of the Fortran intrinsic MIN function versus vector length evaluating the minimum for each element of two vectors shown for a non-vectorized loop (blue, brown) and loops vectorized by the auto-vectorizer (red) and manually (green), and an if-statement performing the same operation vectorized by the auto-vectorizer (magenta). Note that the blue, the red and the magenta curve lie on top of each other. Upper Panel: Using `ifort` and the Vector Math Functions in the Intel IPP library on a Xeon X5570 CPU of HPC-FF. Lower Panel: Using XLF and the Vector Functions in the IBM MASS VP6 library on an IBM POWER6 CPU of VIP.

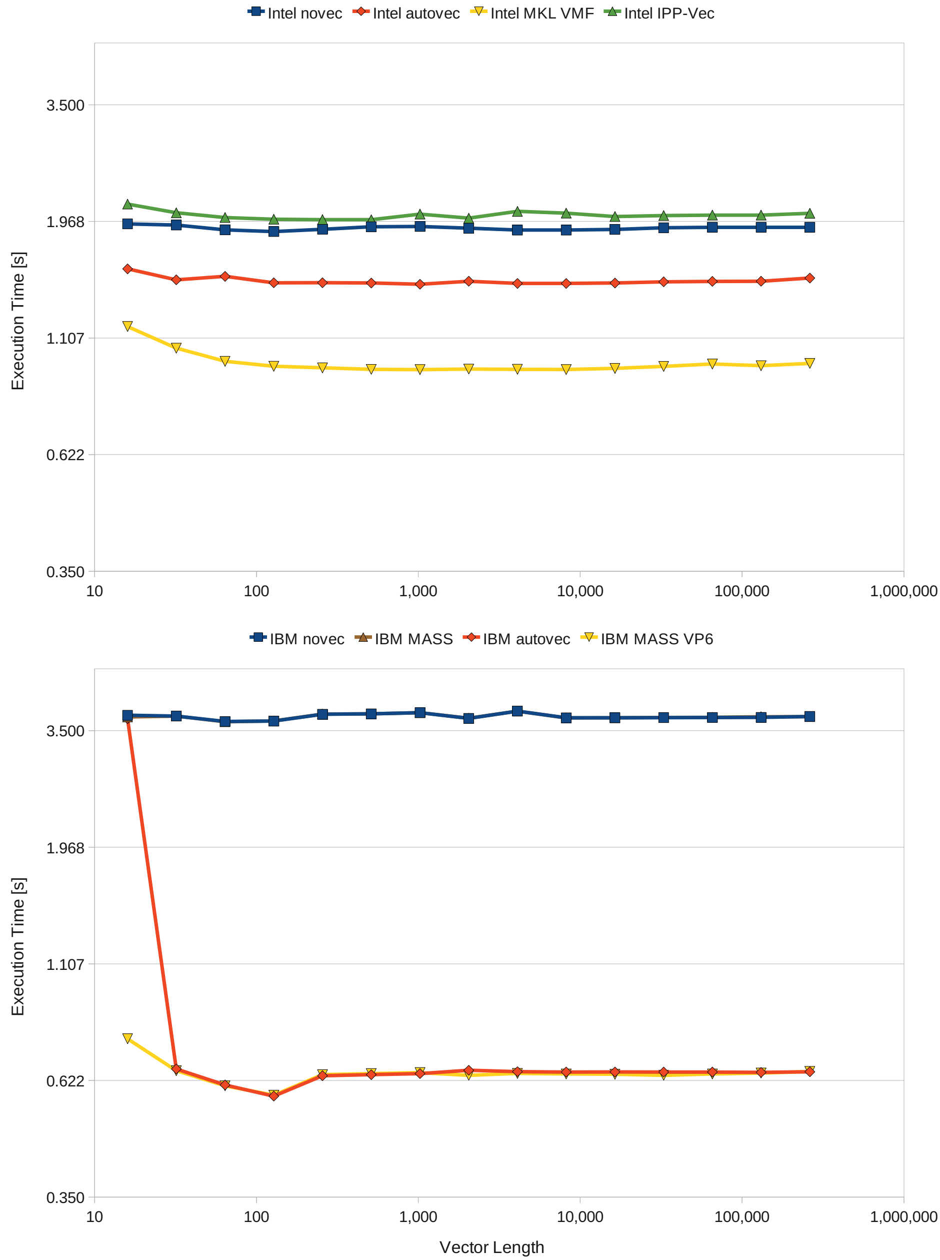

Figure 17: Run time of the intrinsic POW operation versus vector length shown for a non-vectorized loop (blue, brown) and loops vectorized by the auto-vectorizer (red) and manually (yellow, green). The POW operation sets a specific vector element of the result vector to the result of the corresponding vector element of the first input vector raised to the power of the second input vector. However, for this measurement all elements of the second input vector are set to 1.5. Upper Panel: Using `ifort` and the Vector Math Functions in the Intel IPP library on a Xeon X5570 CPU of HPC-FF. Lower Panel: Using XLF and the Vector Functions in the IBM MASS VP6 library on an IBM POWER6 CPU of VIP.

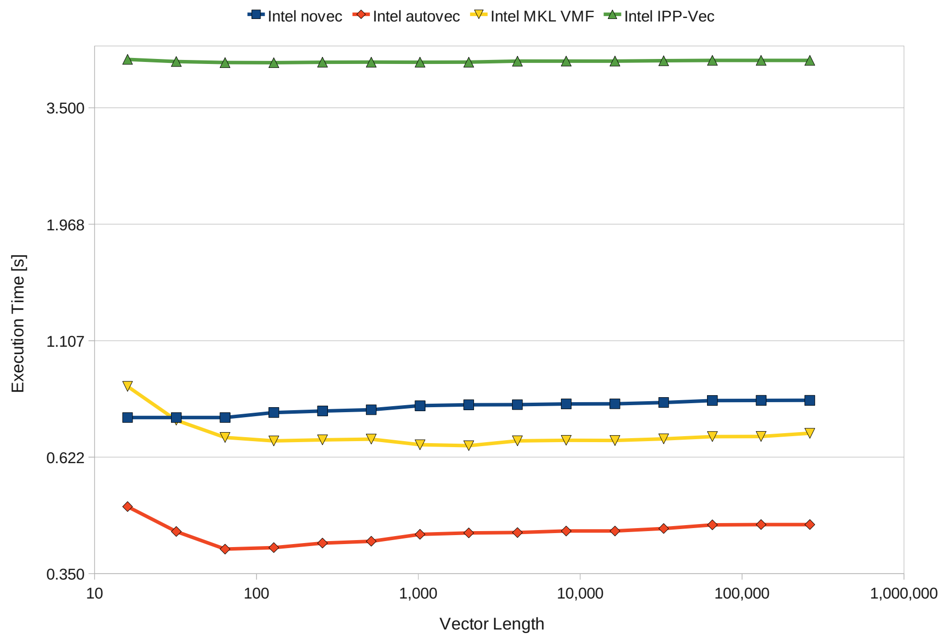

Figure 18: Run time of the intrinsic POW3O2 operation versus vector length shown for a non-vectorized loop (blue) and loops vectorized by the auto-vectorizer (red) and manually (yellow, green). The POW3O2 operation sets the result vector's element to the result of the arguments vector's element to the power of 1.5.

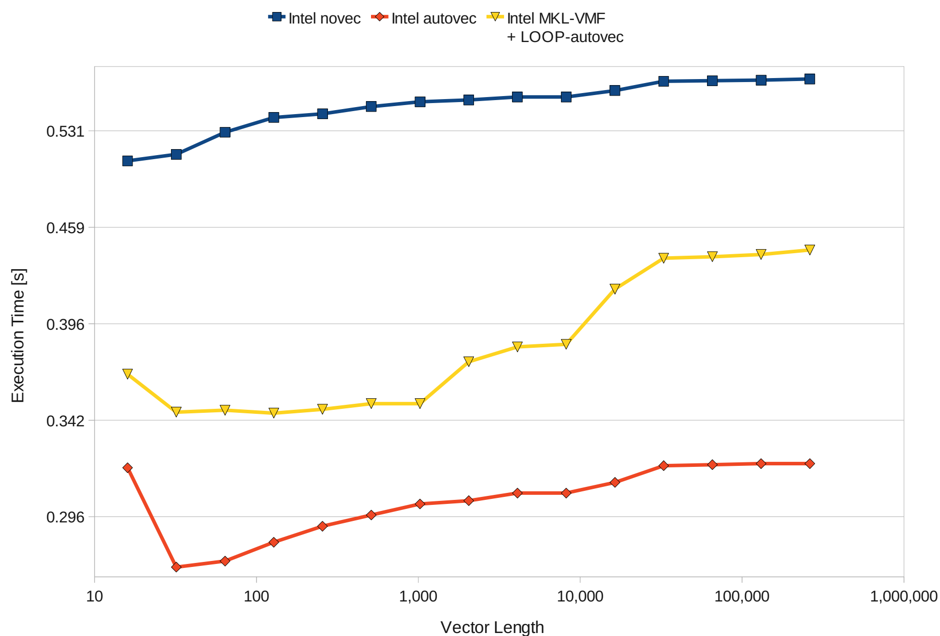

Figure 19: Run time of the Fortran intrinsic MODULO function versus vector length compiled with `ifort` on a Xeon X5570 CPU of HPC-FF shown for a non-vectorized loop (blue) and loops vectorized by the auto-vectorizer (red) and manually using the Vector Functions in the Intel MKL (yellow) and IPP (green) library.

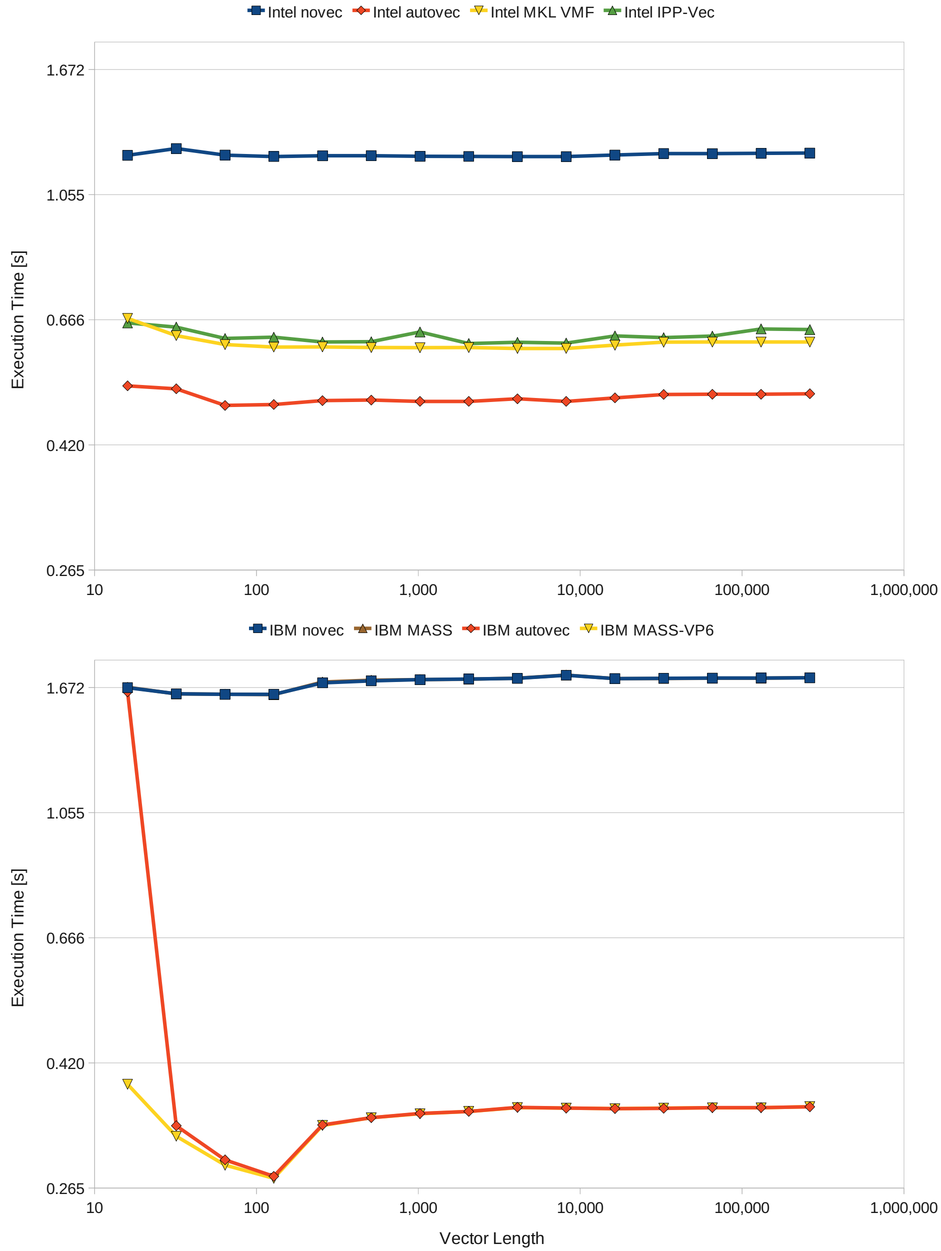

Figure 20: Run time of the Fortran intrinsic SINE (SIN) function versus vector length shown for a non-vectorized loop (blue, brown) and loops vectorized by the auto-vectorizer (red) and manually (yellow, green). Upper Panel: Using `ifort` and the Vector Math Functions in the Intel MKL and Intel IPP library on a Xeon X5570 CPU of HPC-FF. Lower Panel: Using XLF and the Vector Functions in the IBM MASS VP6 library on an IBM POWER6 CPU of VIP.

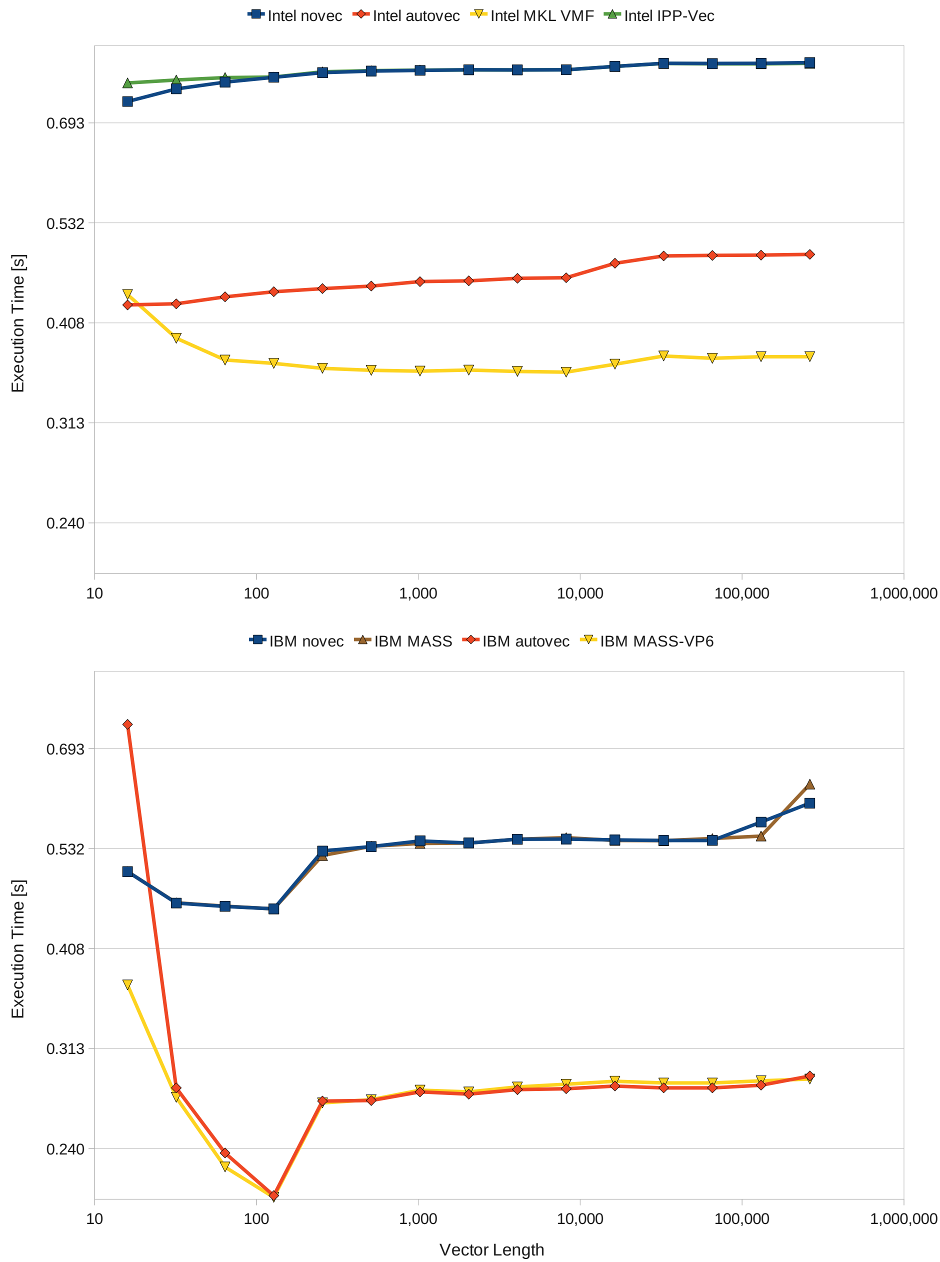

Figure 21: Run time of the Fortran intrinsic SQUARE ROOT (SQRT) function versus vector length shown for a non-vectorized loop (blue, brown) and loops vectorized by the auto-vectorizer (red) and manually (yellow). Upper Panel: Using `ifort` and the Vector Math Functions in the Intel MKL library on a Xeon X5570 CPU of HPC-FF. Lower Panel: Using XLF and the Vector Functions in the IBM MASS VP6 library on an IBM POWER6 CPU of VIP.

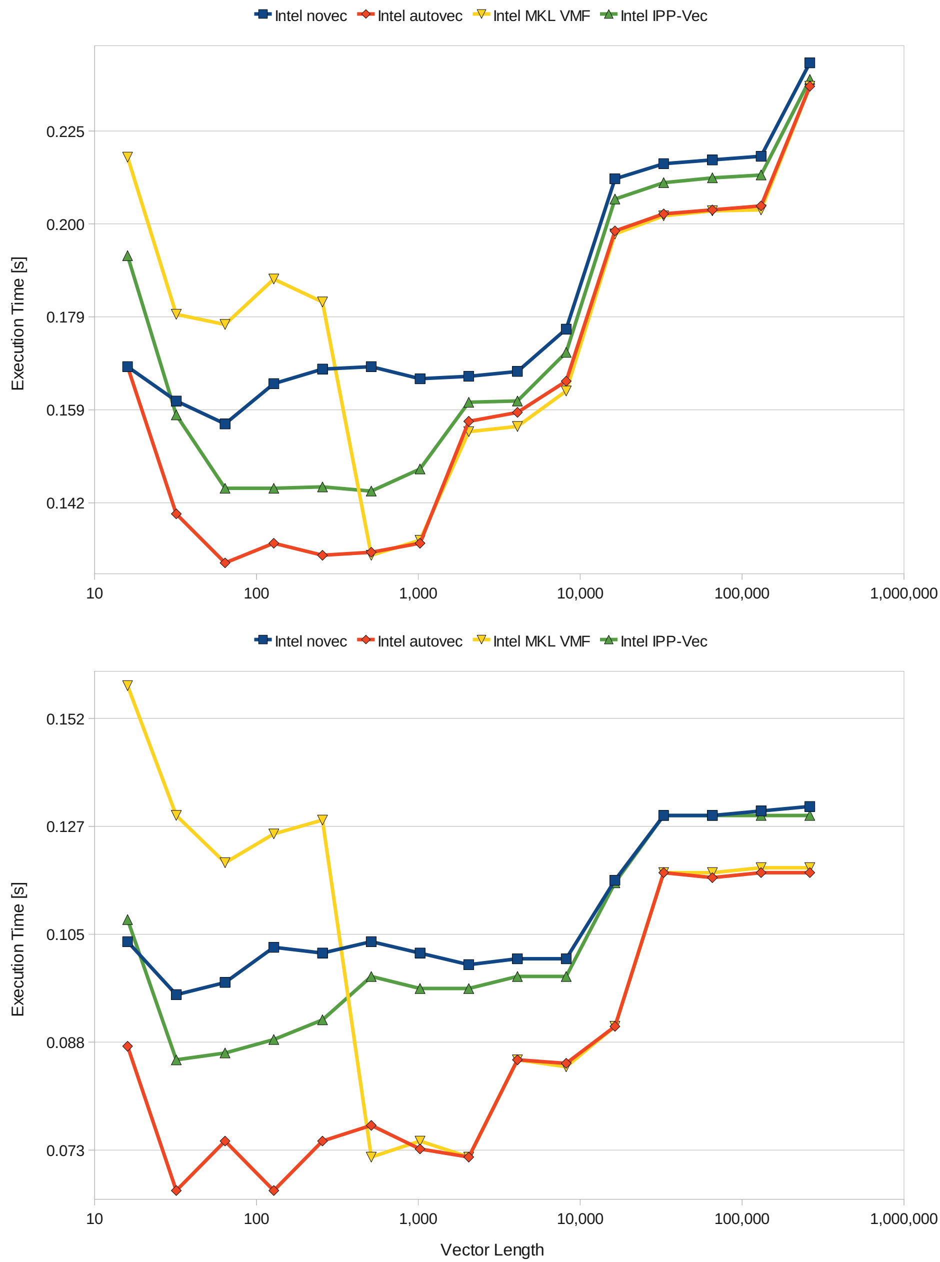

Figure 22: Run time of FORTRAN intrinsic functions versus vector length on a Xeon X5570 CPU of HPC-FF compiled with `ifort` shown for a non-vectorized loop (blue) and loops vectorized by the auto-vectorizer (red) and manually using the Vector Functions in the Intel IPP library(green). Upper Panel: MULTIPLY (MULT) function. Lower Panel: SQUARE (SQR) function.

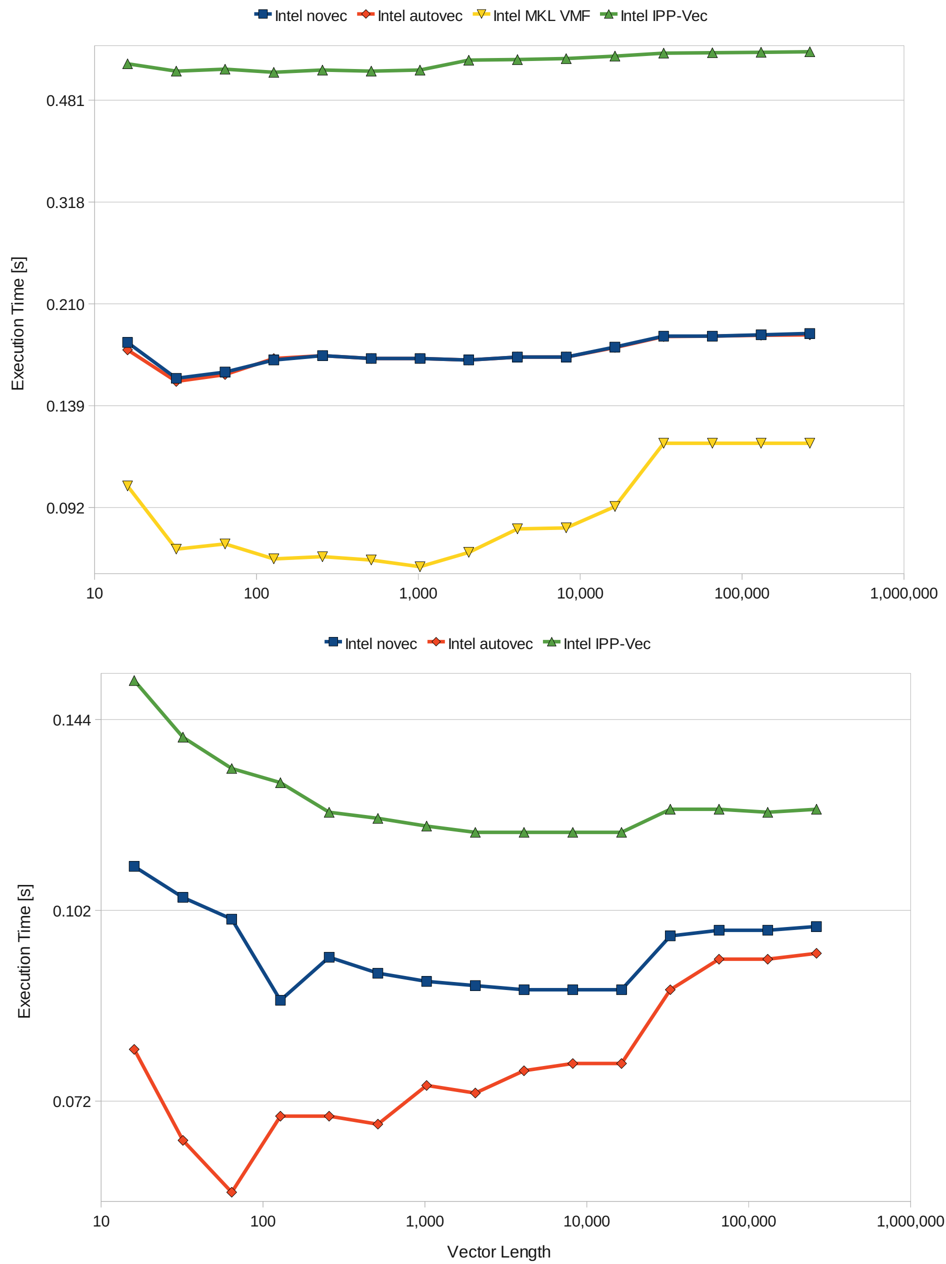

Figure 23: Run time of FORTRAN intrinsic functions versus vector length compiled with `ifort` on a Xeon X5570 CPU of HPC-FF shown for a non-vectorized loop (blue) and loops vectorized by the auto-vectorizer (red) and manually using the Vector Functions in the Intel MKL (yellow) and IPP (green) library. Upper Panel: FLOOR function. Lower Panel: CONVERT TO REAL (REAL) function.

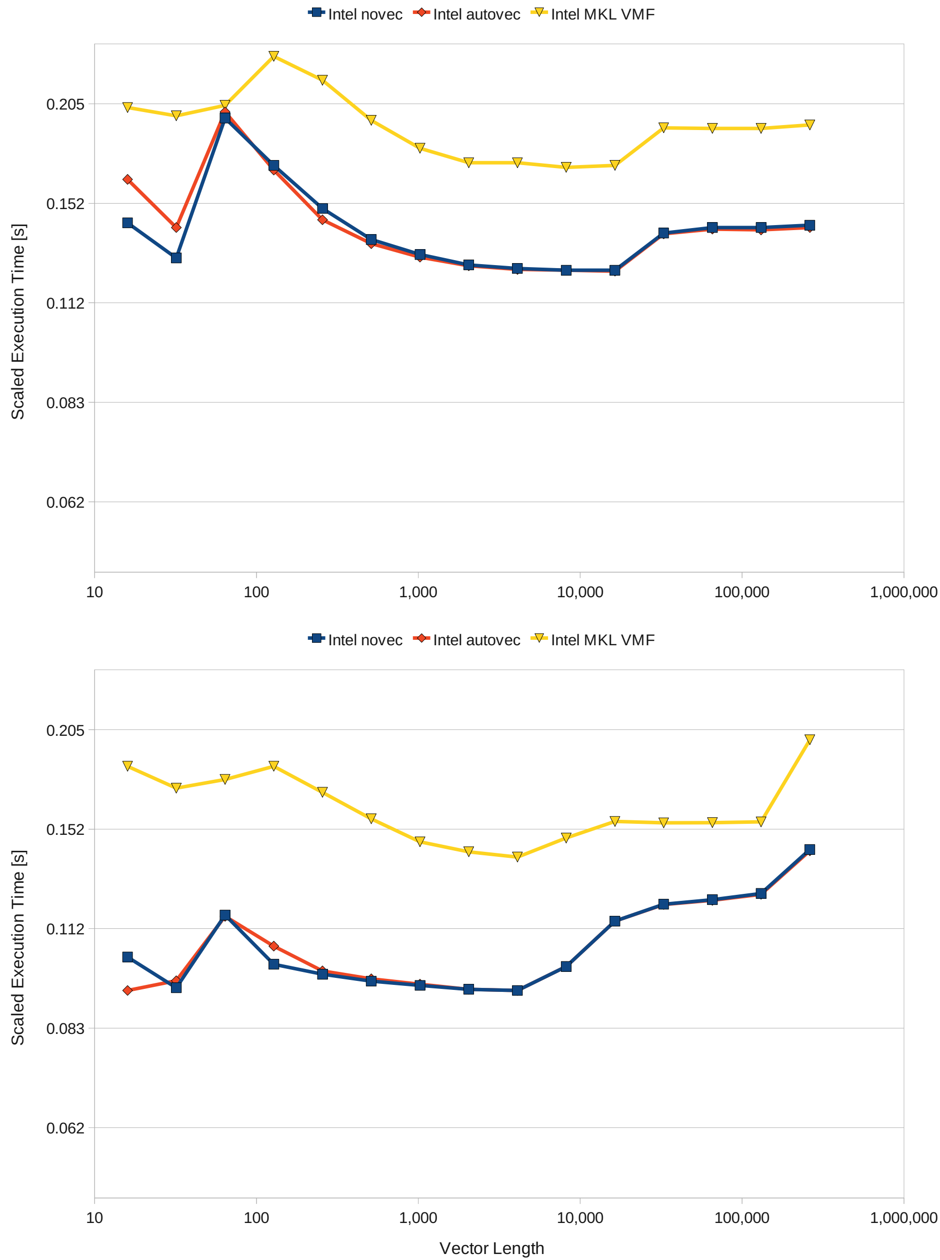

Figure 24: Run time of the Intel IPP VECTOR PACKING operation versus vector length a Xeon X5570 CPU of HPC-FF compiled with `ifort` shown for a non-vectorized loop (blue) and loops vectorized by the auto-vectorizer (red) and manually using the Vector Math Functions of the Intel MKL library (yellow). Run times scaled to run time corresponding to a packing operation with one vector (see Subsection 5.3). Upper Panel: Packing 1 vector using a defined packing pattern. Lower Panel: Packing 3 vectors using the same packing pattern as above.

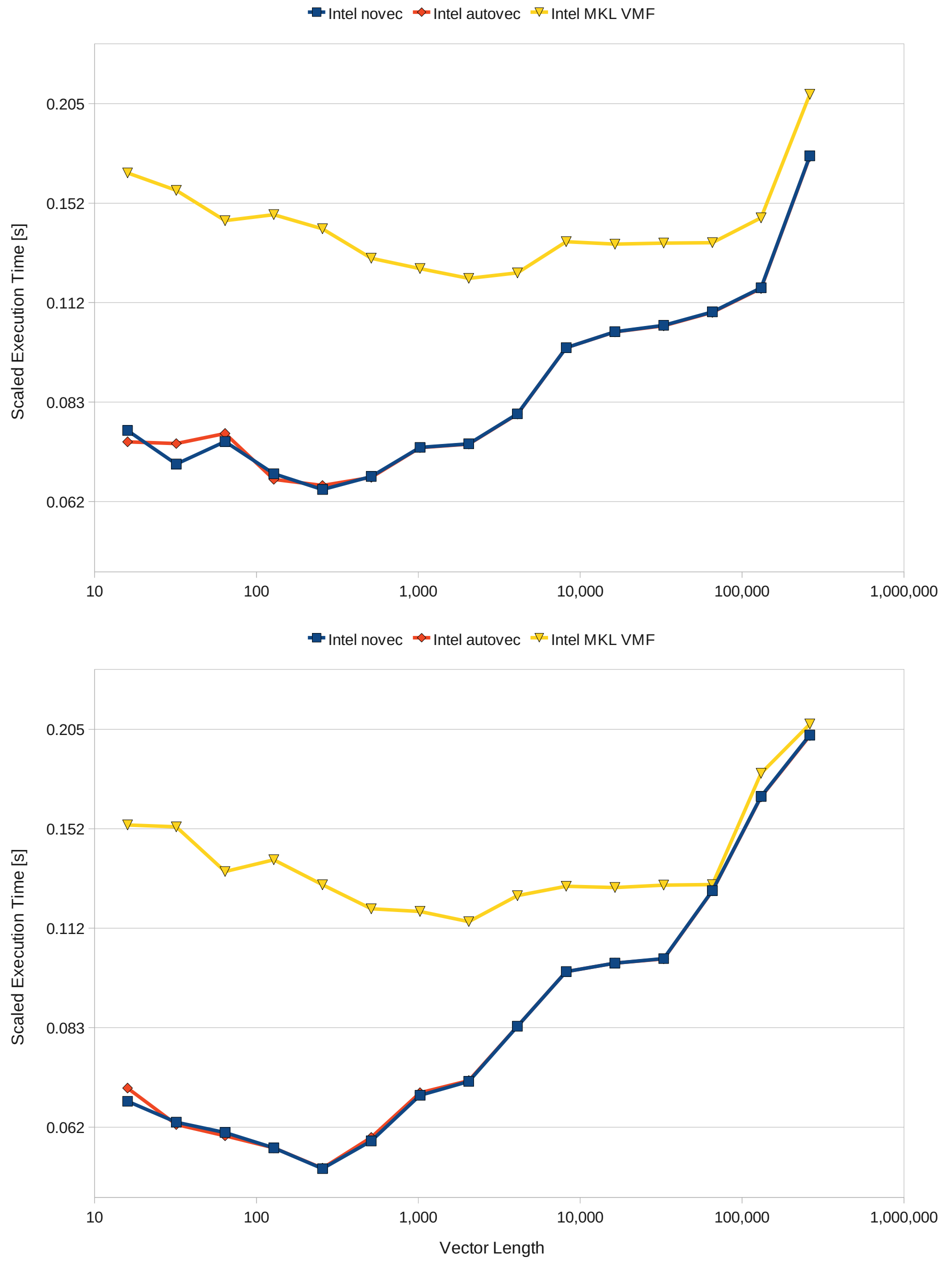

Figure 25: Same as Fig. 24. Run times normalized corresponding to run time of one vector. Upper Panel: Packing 6 vector using a defined packing pattern. Lower Panel: Packing 9 vectors using the same packing pattern as above.

C Single Precision Performance Measurement

In the following only performance curves for intrinsic functions using single precision, i.e. 32 bit arguments are collected (see Subsection 4.2). Note that, all are log log plots and show the average of three independent measurements.

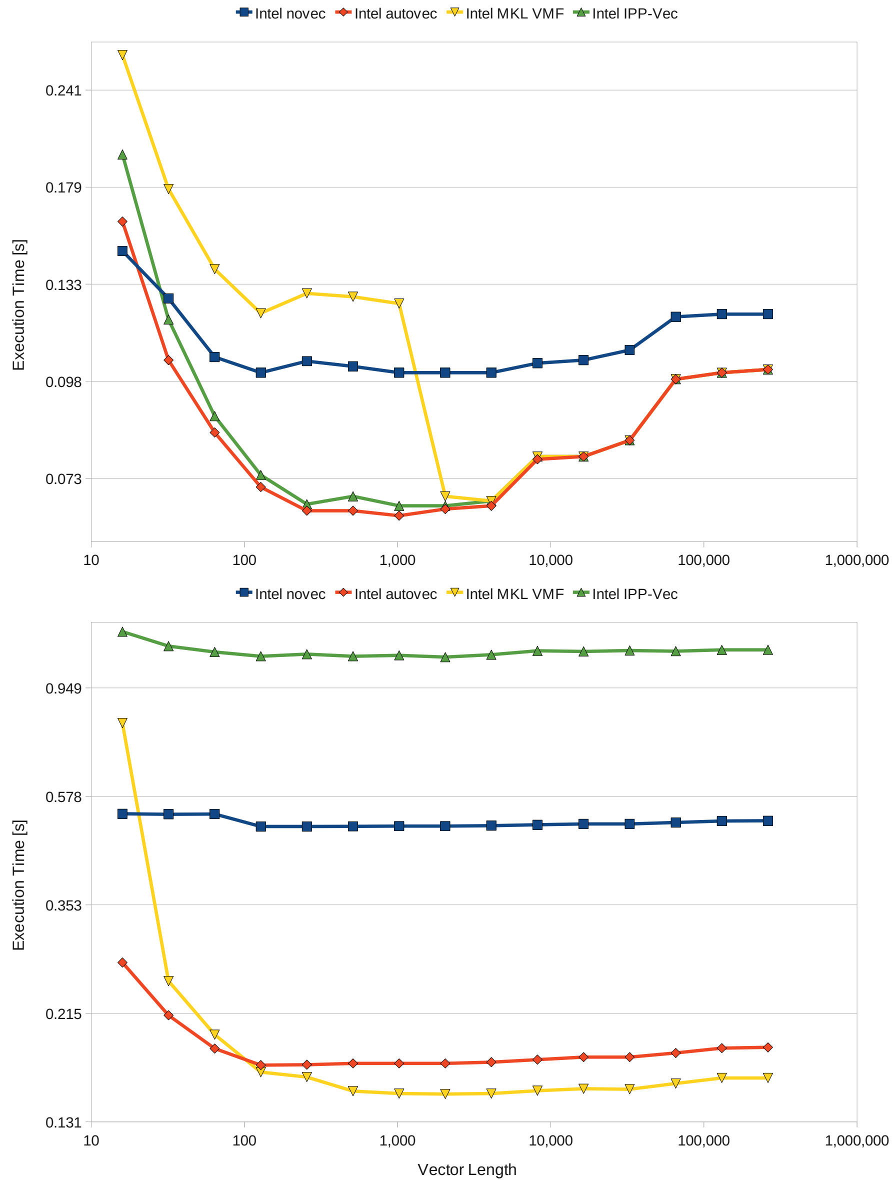

Figure 26: Run time of FORTRAN intrinsic functions for single precision, i.e. 32 bit arguments versus vector length compiled with `ifort` on a Xeon X5570 CPU of HPC-FF shown for a non-vectorized loop (blue) and loops vectorized by the auto-vectorizer (red) and manually using the Vector Functions in the Intel MKL (yellow) and IPP (green) library. Upper Panel: MULTIPLY (MULT) function. Lower Panel: EXPONENTIAL (EXP) function.