Konrad-Zuse-Zentrum
für Informationstechnik Berlin

Takustraße 7
D-14195 Berlin-Dahlem
Germany

FLORIAN WENDE

SIMD Enabled Functions on Intel Xeon CPU and Intel Xeon Phi Coprocessor

Conditional Function Calls, Branching, Early Return

ZIB-Report 15-17 (February 2015)

Herausgegeben vom
Konrad-Zuse-Zentrum für Informationstechnik Berlin
Takustraße 7
D-14195 Berlin-Dahlem

Telefon: 030-84185-0
Telefax: 030-84185-125

e-mail: bibliothek@zib.de
URL: http://www.zib.de

ZIB-Report (Print) ISSN 1438-0064
ZIB-Report (Internet) ISSN 2192-7782

SIMD Enabled Functions on Intel Xeon CPU and Intel Xeon Phi Coprocessor

Conditional Function Calls, Branching, Early Return

Florian Wende (Zuse Institute Berlin)

Mail: wende.at.zib.de

Introduction: To achieve high floating point compute performance, modern processors draw on short vector SIMD units, as found e.g. in Intel CPUs (SSE, AVX1, AVX2 as well as AVX-512 on the roadmap) and the Intel Xeon Phi coprocessor, to operate an increasingly larger number of operands simultaneously. Making use of SIMD vector operations therefore is essential to get close to the processor's floating point peak performance.

Two approaches are typically used by programmers to utilize the vector units: compiler driven vectorization via directives and code annotations, and manual vectorization by means of SIMD intrinsic operations or assembly.

In this paper, we investigate the capabilities of the current Intel compiler (version 15 and later) to generate vector code for non-trivial coding patterns within loops. Beside the more or less uniform data-parallel standard loops or loop nests, which are typical candidates for SIMDfication, the occurrence of e.g.

- (conditional) function calls including branching, and
- early returns from functions

may pose difficulties regarding the effective use of vector operations. Recent improvements of the compiler's capabilities involve the generation of SIMD-enabled functions ("vector functions" hereafter). We will study the effectiveness of the vector code generated by the compiler by comparing it against hand-coded intrinsics versions of different kinds of functions that are invoked within inner-most loops.

1 Branching and Conditional Function Calls

Consider the following code snippet:

```
/***** LISTING 1 ************************************/
double *x=(double *)_mm_malloc(N*sizeof(double),64);
for(int i=0; i<N; i++)
  f1(&x[i]);
```

```
void f1(double *x){
  *x-=100.0;
  if(*x>10000.0)
    f1(x);
  else
    f2(x);
}

void f2(double *x){
  ...
}
```

Depending on the value pointed to by `x`, `f1` calls either itself or `f2`—the definition of `f2` is not relevant here.

We assume that neither `f1` nor `f2` is inlined by the compiler. Vectorization of the `for` loop then requires to call vector versions of `f1` and `f2`, say `vf1` and `vf2`. A possible implementation using Xeon Phi SIMD intrinsics may look as follows:

```
/***** LISTING 2 ************************************/
double *x=(double *)_mm_malloc(N*sizeof(double),64);
for(int i=0; i<N; i+=8)
  vf1((__m512d *)&x[i],0xFF);

void vf1(__m512d *x,__mmask8 mask_0){
  __mmask8 mask_1,mask_2;
  __m512d temp_1;
  temp_1=_mm512_sub_pd(*x,_mm512_set1_pd(100.0));
  *x=_mm512_mask_mov_pd(*x,mask_0,temp_1);
  mask_1=_mm512_cmp_pd_mask(temp_1,_mm512_set1_pd(10000.0),_MM_CMPINT_GT);
  if((mask_2=_mm512_kand(mask_0,mask_1))!=0x0)
    vf1(x,mask_2);
  else if((mask_2=_mm512_kand(mask_0,_mm512_knot(mask_1)))!=0x0)
    vf2(x,mask_2);
}

void vf2(__m512d *x,__mmask8 mask_0){
  ...
}
```

We use the fact that the data type `__mmask8` is an integer bitmask, with bits set to 1 for SIMD lanes for which the predicate evaluates to `true` (active SIMD lanes), and 0 (inactive SIMD lanes) otherwise. Testing for at least one SIMD lane remaining active can be done by comparing the mask against non-zero value. Our vector versions of the functions `f1` and `f2` carry an additional masking argument `mask_0`. For function arguments that are modified within any of `vf1` and `vf2` the new values on active

SIMD lanes are blended in using the mask while those on inactive lanes are retained. Furthermore, the mask is used for branching within any of `vf1` and `vf2`, where SIMD lanes that are inactive have no vote. This can be implemented by combining the results of comparison operations with `mask_0` using a logical AND operation.

Vector versions of functions `f1` and `f2` can be also generated by the compiler using appropriate annotations (instead of writing intrinsics code):

```
/***** LISTING 3 *************************************/
double *x=(double *)_mm_malloc(N*sizeof(double),64);
#pragma simd
for(int i=0; i<N; i++)
  f1(&x[i]);

__attribute__((vector(linear(x:1),vectorlength(8))))
void f1(double *x){
  __assume_aligned(x,64);
  *x-=100.0;
  if(*x>10000.0)
    f1(x);
  else
    f2(x);
}

__attribute__((vector(linear(x:1),vectorlength(8))))
void f2(double *x){
  ...
}
```

2 Early Return from Functions

For functions containing a pattern of the following form

```
/***** LISTING 4 *************************************/
void f3(double *x){
  double a;
  ...
  if(*x>a)
    return;
  ...
}
```

an early return happens in case of `(*x>a)` evaluates to `true`. The vector version of `f3` can only return if `(*x>a)` evaluates to `true` on all SIMD lanes. Otherwise, the execution needs to continue. Write operations on function arguments after the "early return" have to use the mask which is the result of the predicate evaluation.

With Xeon Phi SIMD intrinsics, the early return can be implemented as follows (again an additional function argument `mask_0` is present):

```
/***** LISTING 5 ************************************/
void vf3(__m512d *x,__mmask8 mask_0){
  __mmask8 mask_1;
  __m512d a;
  ...
  mask_1=_mm512_kand(mask_0,_mm512_cmp_pd_mask(*x,a,_MM_CMPINT_LE));
  if(mask_1==0x0)
    return;
  ...
}
```

Instead of `(*x>a)` we evaluate `(*x<=a)`, as we ask for which SIMD lanes the execution will continue. Only if `mask_1` is identical to zero, we can return from `vf3`.

Again the compiler can generate the vector version of `f3` using the above given annotations.

3 Performance Comparison

We compare the performance (our metric is the total execution time of 1024 loop iterations) of scalar and vector code for cases where

(1) all SIMD lanes call the same function(s) recursively n times.

(2) a subset of the SIMD lanes each call the same function(s) recursively n times.

(3) all SIMD lanes call the same function(s) recursively at most n times, that is, some SIMD lanes become inactive before the n-th recursion.

(4) SIMD lanes call different functions in a recursive schema at most n times, that is, some SIMD lanes become inactive before the n-th recursion.

We define four different functions (pseudo-code):

```
/***** LISTING 6 ************************************/
void func_[1,2,3,4](double *x,const double *p){
  "func_body_[1,2,3,4](*x)"
  if(*p==1.0)
    func_1(x,p+8)
  else if(*p==2.0)
    func_2(x,p+8)
  else if(*p==3.0)
    func_3(x,p+8)
  else if(*p==4.0)
    func_4(x,p+8)
}
```

```
func_body_1(x): LOOPN(x+=100.0)
                if(x>10000.0) x=-1.0
func_body_2(x): LOOPN(x-=250.0)
                if(x<-10000.0) x=1.0
func_body_3(x): LOOPN(x+=exp(-1.0/x))
                if(x>100.0) x=-log(x)
func_body_4(x): if(x<0.0||x>10000.0) return
                LOOPN(x+=sqrt(x))
```

The function argument `p` encodes the calling tree: it basically corresponds to a two-dimensional array of size `[n][8]` (for the Xeon Phi the SIMD width is 8 for 64-bit words). `p` contains values 1.0, 2.0, 3.0, 4.0 and 0.0 (exit). The former values are used for the branching, whereas the latter signals the end of the recursion.

A possible calling tree may look as follows (for $n = 10$):

lane_1	lane_2	lane_3	lane_4	[lane_5	lane_6	lane_7	lane_8]
func_1	func_4	func_4	func_3	func_4	func_3	func_4	func_4
func_4	func_1	func_1	func_2	func_3	func_2	func_3	func_1
func_2	func_4	func_2	func_1	func_4	func_4	func_2	func_1
func_3	func_4	func_3	func_1	func_1	func_1	func_3	func_3
func_3	func_1	func_3	func_4	func_2	func_1	func_3	func_4
func_4	func_2	func_4	func_3	func_4	func_4	func_2	func_2
func_4	func_3	func_1	func_1	func_1	func_3	func_2	func_4
func_2	func_3	func_3	func_2	func_1	func_2	func_3	func_2
func_4	func_1	func_3	func_1	func_3	func_4	func_2	func_3
func_4	func_2	func_1	func_2	func_1	func_2	func_4	func_1
exit	exit	exit	exit	exit	exit	exit	exit

Here, all SIMD lanes survive throughout the recursion with no early returns (exit). Within the functions `func_[1,2,3,4]` we use a macro definition `LOOPN(x)` that inserts `x` multiple times in succession. By this means we can control the ratio of arithmetic operations to control logic. `x` refers to an abstract arithmetic operation which might be a combination of several elementary arithmetic operations.

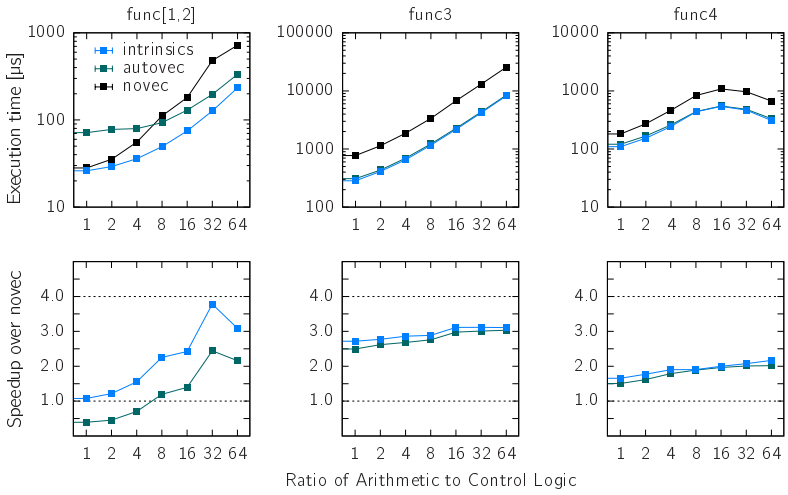
Build: We use the Intel C/C++ compiler (ver. 15.0.1, 20141023) to generate scalar and vector code. On AVX1 hosts, we use the flags `-O3 -xAVX -fp-model=precise -qopt-assume-safe-padding`. On AVX2 hosts, we use the flags `-O3 -xcore-avx2 -fp-model=precise -qopt-assume-safe-padding`. For Xeon Phi coprocessor executions, we use the offload model, where compile flags are inherited from the host. We use the Intel MPSS 3.4.1.

Platforms: Our platforms comprise (a) Xeon E5-2680 CPU (Sandy Bridge, AVX1), (b) Xeon E5-2680v3 CPU (Haswell, AVX2), and (c) Xeon Phi 7120P coprocessor.

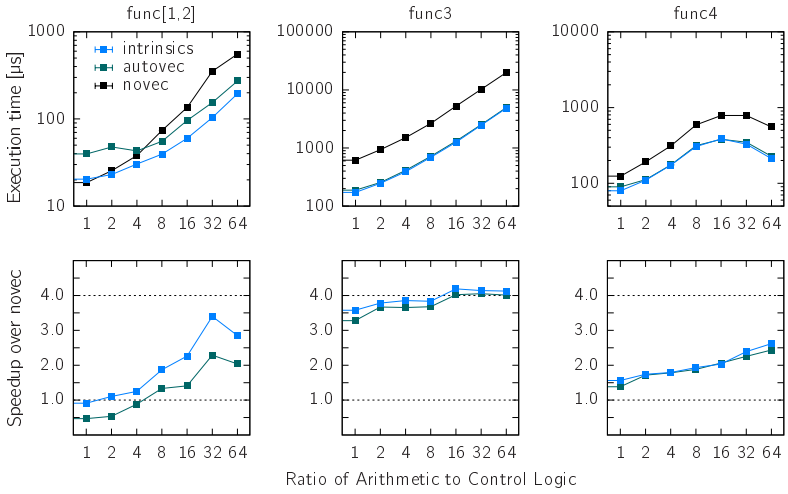
3.1 All SIMD Lanes Call the Same Function

On all platforms the execution of `func_[1,2]` is dominated by the control logic in case of only a few (abstract) arithmetic operations are performed. The SIMD intrinsics versions, however, introduce less overhead than the compiler generated vector versions on all three platforms.

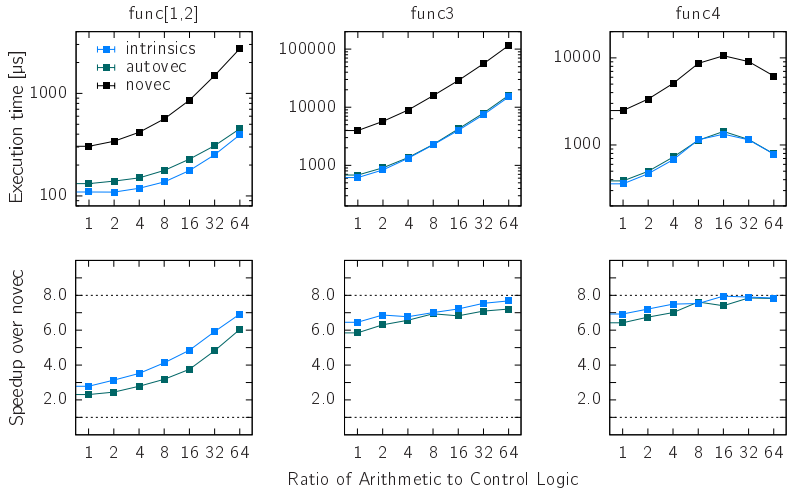
Platform (a), AVX1:



Platform (b), AVX2:



Platform (c), “Xeon Phi”:

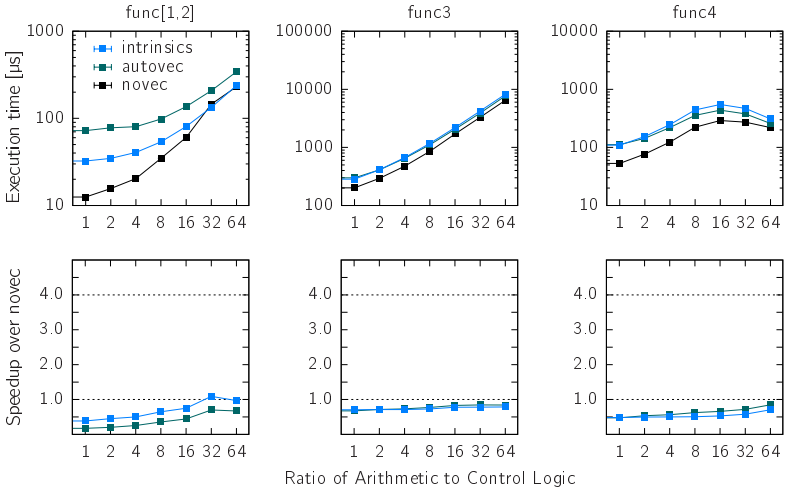


Increasing the ratio of arithmetic operations to control logic seems to move the speedup over the scalar execution (“novec”) towards the (theoretical) limit of 4 for AVX1 and AVX2, and 8 on Xeon Phi. For `func_3` platforms (b) and (c) give speedups over “novec” close to 4 respectively 8, whereas platform (a) is behind at about a factor 3. For `func_4` only the Xeon Phi gets close to the (theoretical) speedup limit.

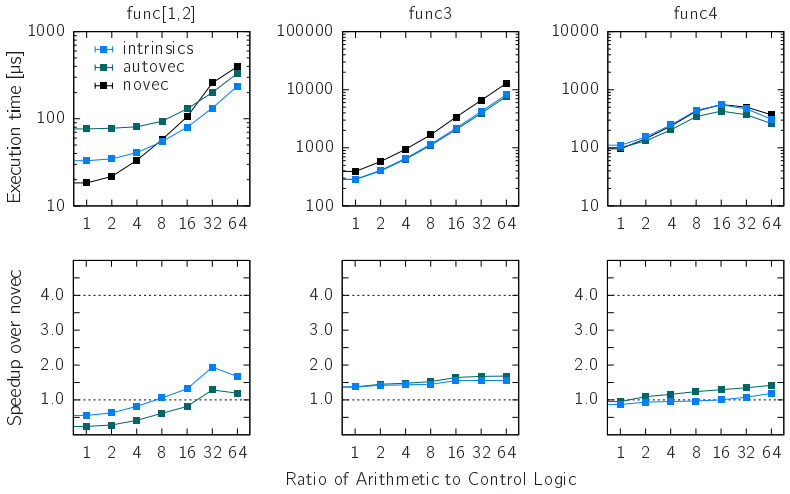
3.2 A Subset of the SIMD Lanes Calls the Same Function

The number of active SIMD lanes is reduced from 4 to 3, 2, 1 on platform (a) and (b), and from 8 to 6, 4, 2, 1 on platform (c). Using the above annotations, the compiler automatically generates (un)masked versions of the functions in these cases.

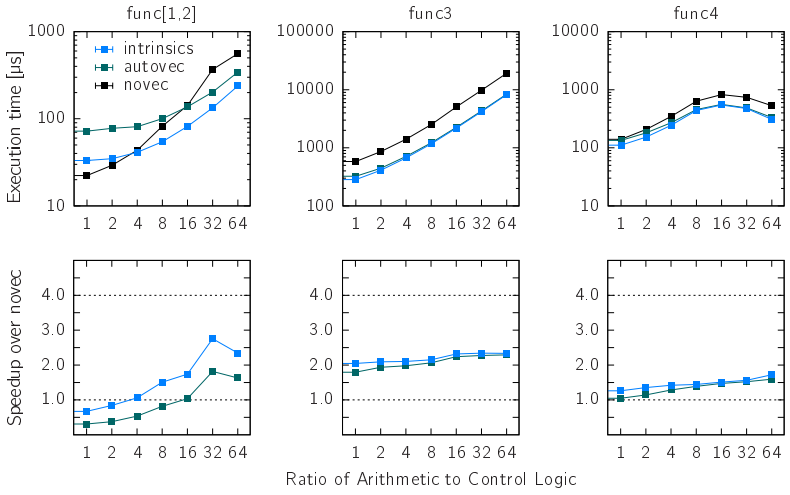
Platform (a), AVX1: 1 SIMD lane active



Platform (a), AVX1: 2 SIMD lanes active



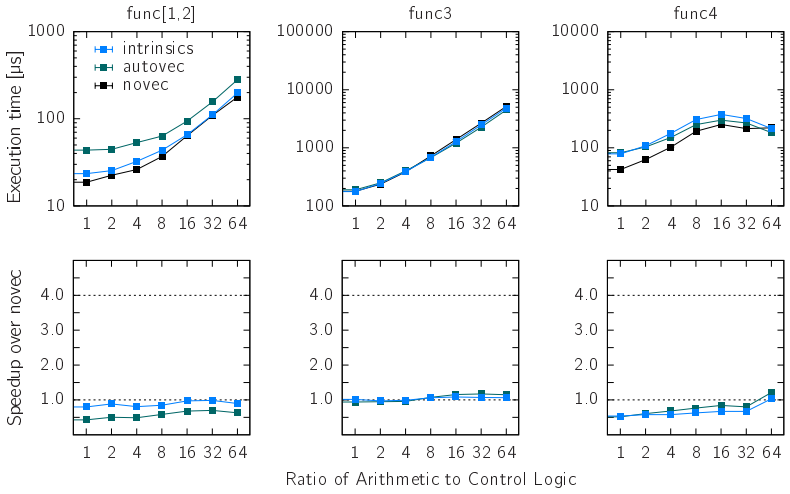
Platform (a), AVX1: 3 SIMD lanes active



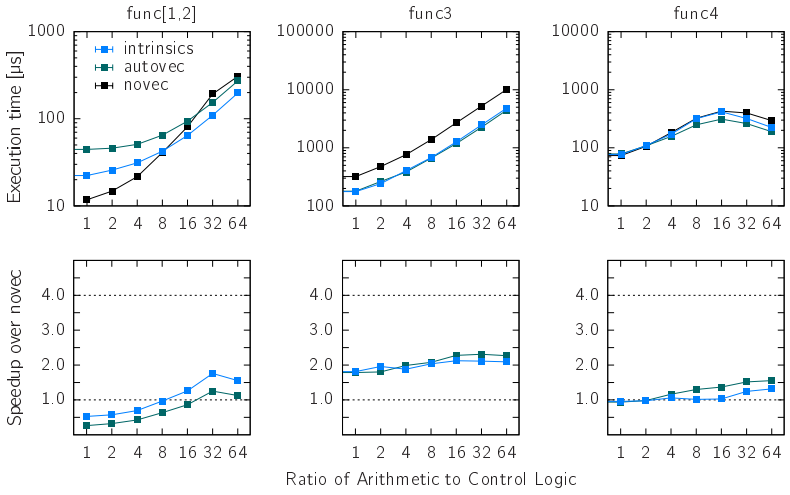
The most interesting case is the one where only a single SIMD lane is active. Why? It directly shows the performance slowdown over the scalar execution. In case of nested branching, it is likely that exactly this situation will occur. Our results show that with AVX1 only in case of a large ratio of arithmetic operations to control logic the scalar performance can be reached.

For the other cases with m active SIMD lanes, the expected speedup over the scalar execution (“novec”) is m. For none of the functions m is reached.

Platform (b), AVX2: 1 SIMD lane active



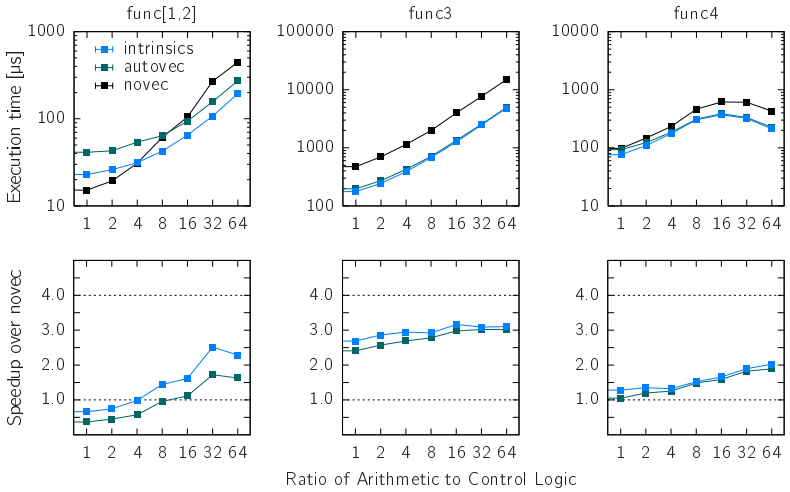
Platform (b), AVX2: 2 SIMD lanes active



With AVX2 the performance gain over scalar execution is close to the expected one just for `func_3`. For the other three functions the speedup values over “novec” are comparable to the AVX1 case, with only little increase.

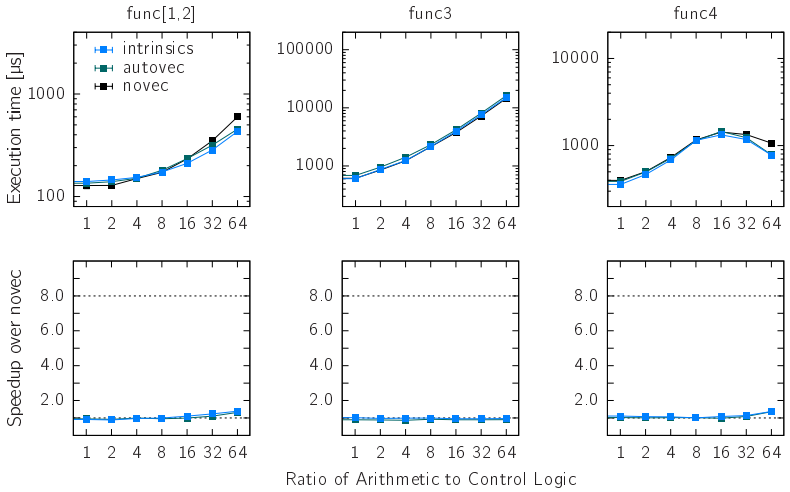
The execution of vector functions with nested branching and hence reduced number of active SIMD lanes thus is expected to perform below the scalar execution (we will consider such cases below).

Platform (b), AVX2: 3 SIMD lanes active

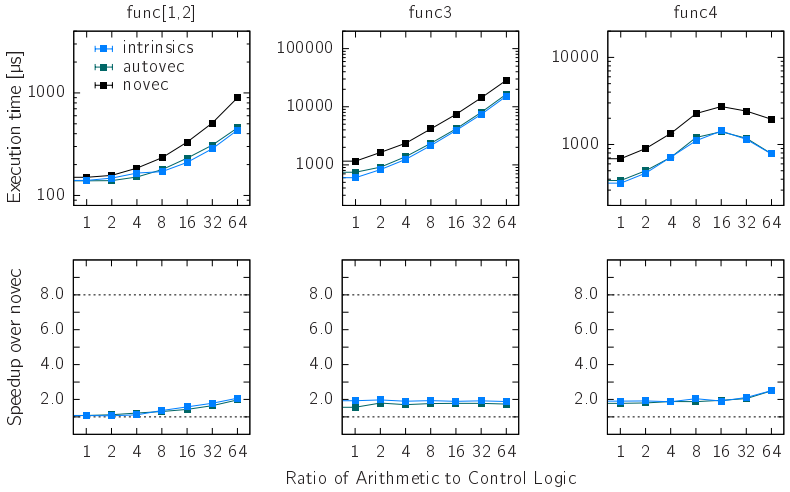


Why are the speedups over “novec” below the expected ones for $m < 4$ active SIMD lanes? One reason might be the point that both AVX1 and AVX2 do not support masked SIMD operations. Our way to introduce masking anyhow is using logical operations on 256-bit vectors together with blending for masked data movement. We use the AVX representation of `true` (`0xFFFFFFFF`) and `false` (`0x0`) returned e.g. by `_mm256_cmp_pd()`. Since AVX SIMD registers then hold both masks and operands for computations, the number of SIMD registers effectively available for arithmetic operations reduces. The latter may affect the performance of the vector execution.

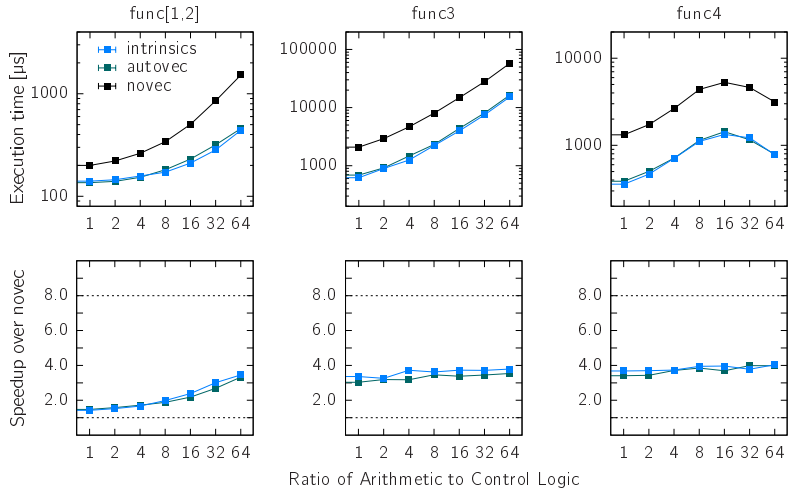
Platform (c), “Xeon Phi”: 1 SIMD lane active



Platform (c), “Xeon Phi”: 2 SIMD lanes active



Platform (c), “Xeon Phi”: 4 SIMD lanes active

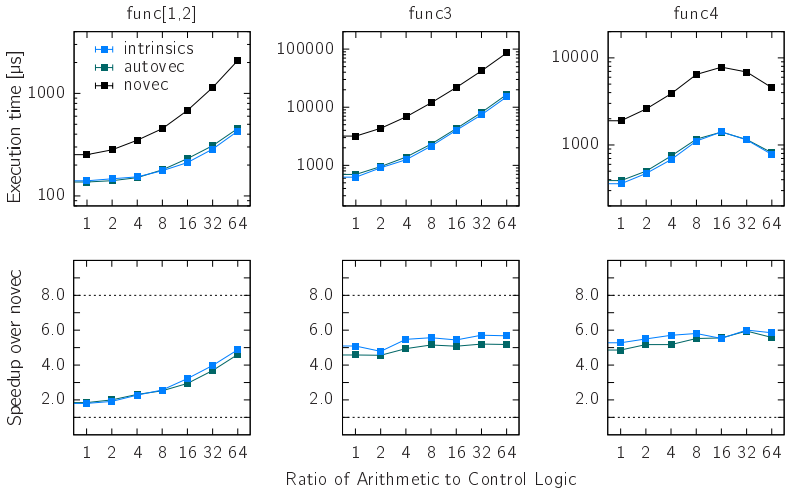


On the Xeon Phi the vector execution with one active SIMD lane is almost exactly comparable to the scalar execution. This means, that even in the worst case where all execution on the SIMD lanes is serialized—e.g. due to nested branching or early returns—the performance does not fall below the scalar performance.

Compared to AVX1 and AVX2 the speedup of the m-active-SIMD-lanes executions over “novec” is very close to the expectations.

As already seen for AVX2, the performance difference between the compiler vectorized versions and those using SIMD intrinsics almost vanishes. That means the compiler generated vector functions perform equal well as their intrinsics counterparts.

Platform (c), “Xeon Phi”: 6 SIMD lanes active



3.3 All SIMD Lanes Call the Same Function + Early Return

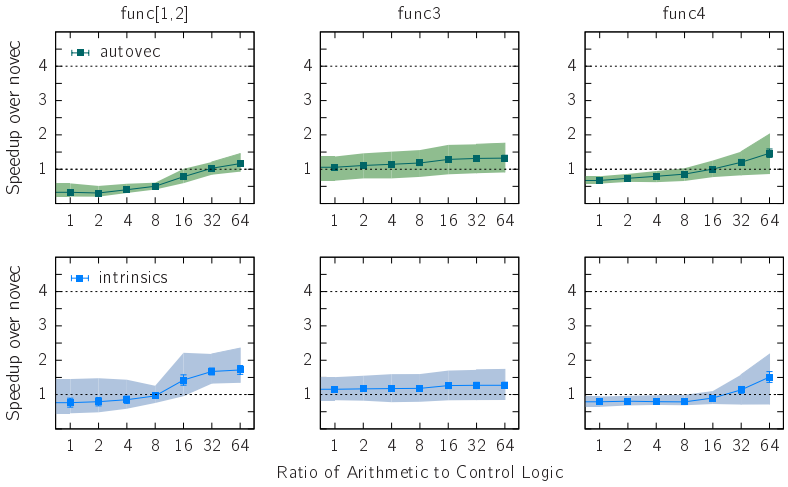
We consider the case where for the SIMD lanes the depth of the per-lane calling trees may vary. Particularly, we use a per-lane probability $p_i \in [0, 1)$ to decide about performing a further call or not. That is, the number of active SIMD lanes gradually decreases. A possible calling tree on the Xeon Phi may look as follows:

```
lane_1  lane_2  lane_3  lane_4 [lane_5  lane_6  lane_7  lane_8]
------  ------  ------  ------  ------  ------  ------  ------
func_2  func_2  func_2  func_2  func_2  func_2  func_2  func_2
func_2  func_2  exit    func_2  func_2  func_2  func_2  func_2
func_2  func_2  exit    func_2  func_2  func_2  func_2  func_2
func_2  func_2  exit    func_2  func_2  func_2  func_2  func_2
exit    func_2  exit    func_2  exit    func_2  exit    func_2
exit    func_2  exit    func_2  exit    func_2  exit    func_2
exit    func_2  exit    func_2  exit    func_2  exit    func_2
exit    func_2  exit    func_2  exit    func_2  exit    func_2
exit    func_2  exit    exit    exit    func_2  exit    func_2
exit    func_2  exit    exit    exit    exit    exit    func_2
exit    exit    exit    exit    exit    exit    exit    exit
```

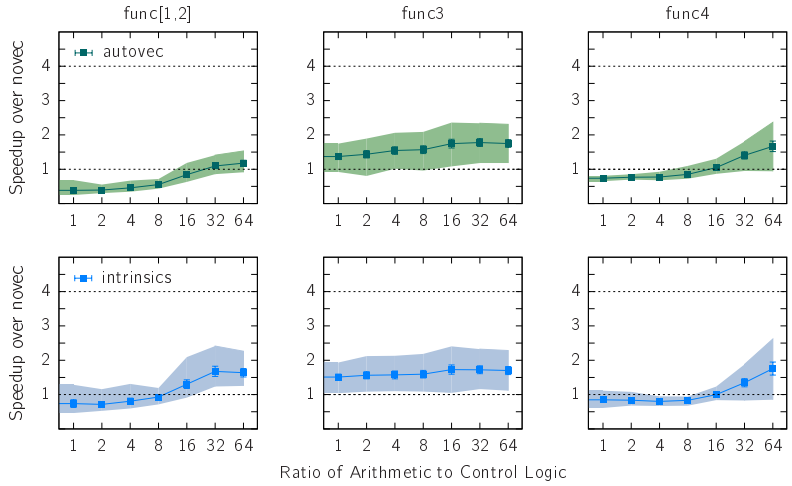
The maximum calling depth is fixed to 10 for our experiments.

For the different functions and ratios of arithmetic operations to control logic, we consider 10 randomly generated setups—the random number sequences used are the same on all platforms. For each setup we determine the gain of the vector execution over the scalar execution, and give the minimum, average and maximum values below.

Platform (a), AVX1:



Platform (b), AVX2:

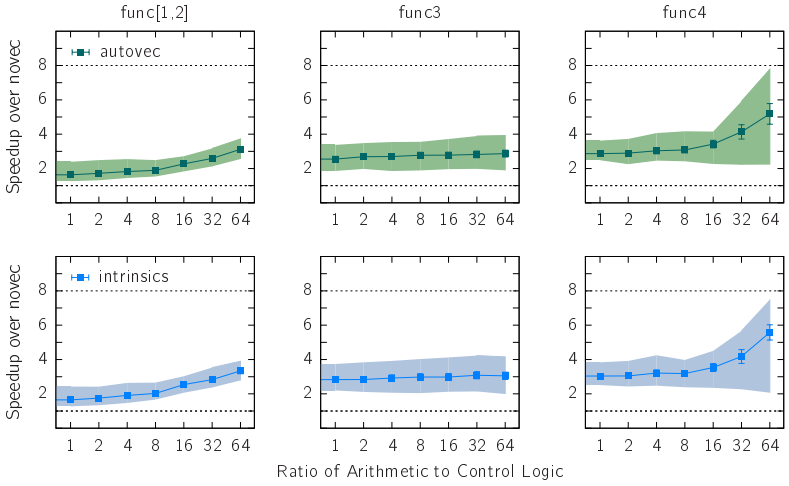


On both platform (a) and (b) the performance of the compiler generated vector functions falls below the scalar performance in case of `func_[1,2]` and for small ratios of arithmetic operations to control logic. The intrinsics based version, however gives about twice the performance and moves the average speedup above 1 for the ratio larger than 8.

For `func_3` the average speedup over “novec” is larger than 1 in all cases. With AVX2 even the minimum speedups are above 1.

In case of `func_4` the average speedup is only slightly below 1 for the ratio of arithmetic to control logic smaller than 16, and increases up to about a factor 1.5 otherwise. Maximum speedups up to a factor 2.5 can be noted for platform (b).

Platform (c), “Xeon Phi”:



On the Xeon Phi platform minimum speedups are always larger than 1.0, that is, in no case the performance is behind the scalar execution. Average speedups up to a factor 6 can be noted for `func_4`, and maximum speedups reach up to a factor 8.

3.4 SIMD Lanes Call Different Functions + Early Return

Following up the last section, we now allow different functions to be called along the per-lane calling trees. For a given number of different functions (selected at random) the number of active SIMD lanes may reduce for two reasons: SIMD lanes have already finished their execution, or they do not participate in other lanes’ function calls.

A possible calling tree with three different functions may look as follows (note: lanes 5 – 8 are mirrored from lane 1 – 4 to get the same calling trees and hence comparable results across all platforms):

```
lane_1  lane_2  lane_3  lane_4 [lane_5  lane_6  lane_7  lane_8]
------  ------  ------  ------  ------  ------  ------  ------
func_2  func_4  func_2  func_2  func_2  func_4  func_2  func_2
func_4  func_2  func_4  func_4  func_4  func_2  func_4  func_4
func_1  func_1  func_1  func_2  func_1  func_1  func_1  func_2
func_2  func_2  func_2  func_4  func_2  func_2  func_2  func_4
func_1  func_1  func_4  func_2  func_1  func_1  func_4  func_2
func_1  func_2  func_1  func_4  func_1  func_2  func_1  func_4
func_4  func_4  func_1  func_2  func_4  func_4  func_1  func_2
func_4  func_1  exit    func_1  func_4  func_1  exit    func_1
func_1  func_1  exit    func_4  func_1  func_1  exit    func_4
func_1  func_4  exit    exit    func_1  func_4  exit    exit
exit    exit    exit    exit    exit    exit    exit    exit
```

The depth-first execution of these functions (as used within our function definitions; see Listing 6) is as follows:

```
lane_1  lane_2  lane_3  lane_4 [lane_5  lane_6  lane_7  lane_8]
------  ------  ------  ------  ------  ------  ------  ------
func_2  *       func_2  func_2  func_2  *       func_2  func_2
func_4  *       func_4  func_4  func_4  *       func_4  func_4
func_1  *       func_1  *       func_1  *       func_1  *
func_2  *       func_2  *       func_2  *       func_2  *
func_1  *       *       *       func_1  *       *       *
func_1  *       *       *       func_1  *       *       *
func_4  *       *       *       func_4  *       *       *
func_4  *       *       *       func_4  *       *       *
func_1  *       *       *       func_1  *       *       *
func_1  *       *       *       func_1  *       *       *
*       *       func_4  *       *       *       func_4  *
*       *       func_1  *       *       *       func_1  *
*       *       func_1  *       *       *       func_1  *
*       *       *       func_2  *       *       *       func_2
*       *       *       func_4  *       *       *       func_4
*       *       *       func_2  *       *       *       func_2
*       *       *       func_4  *       *       *       func_4
*       *       *       func_2  *       *       *       func_2
*       *       *       func_1  *       *       *       func_1
*       *       *       func_4  *       *       *       func_4
*       func_4  *       *       *       func_4  *       *
*       func_2  *       *       *       func_2  *       *
*       func_1  *       *       *       func_1  *       *
*       func_2  *       *       *       func_2  *       *
*       func_1  *       *       *       func_1  *       *
*       func_2  *       *       *       func_2  *       *
*       func_4  *       *       *       func_4  *       *
*       func_1  *       *       *       func_1  *       *
*       func_1  *       *       *       func_1  *       *
*       func_4  *       *       *       func_4  *       *
```

The execution happens along the vertical direction from top to bottom. The asterisks (“⋆”) mark out SIMD lanes that either finished their calling tree, or do not participate in the current (vector) function execution. For the functions `func_[1,2,4]` the number of vector calls with 1, 2, 3, 4 and 6 active lanes is noted in Table 1.

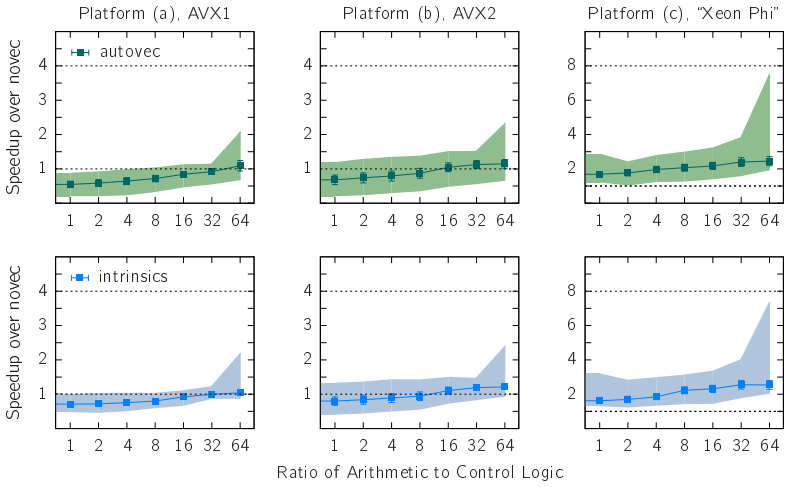
	`func_1`	`func_2`	`func_4`
1 (resp. 2) lanes active	11	6	9
2 (resp. 4) lanes active	1	1	0
3 (resp. 6) lanes active	0	1	1

Table 1: Vector function count depending on the number of active SIMD lanes for `func_[1,2,4]`.

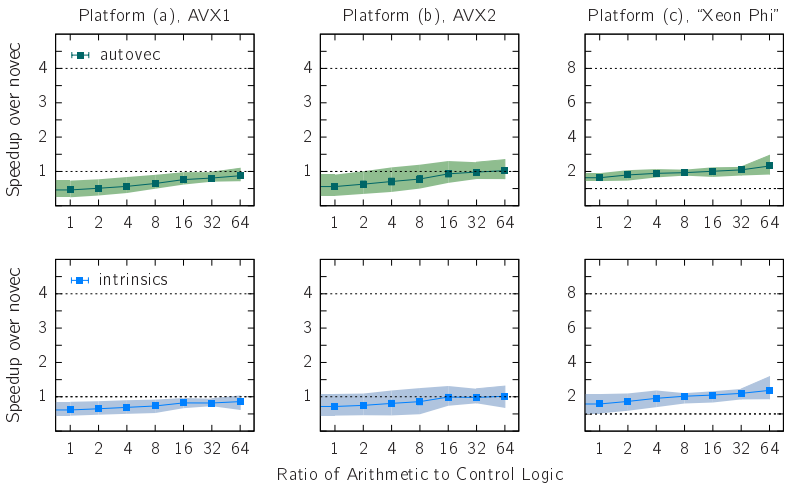
In case of scalar execution `func_1` counts 13 (resp. 26) times, `func_2` counts 11 (resp. 22) times, and `func_3` counts 12 (resp. 24) times—we need to distinguish between vector execution with AVX1 and AVX2, and vector execution on Xeon Phi, where twice as many SIMD lanes are available.

Assuming that vector executions with just one active SIMD lane do not fall behind the respective scalar executions, we can expect at least a factor $\min(\frac{13}{12}, \frac{11}{8}, \frac{11}{10}) \approx 1.1$ performance gain over the scalar execution on platform (a) and (b), and at least a factor $\min(\frac{26}{12}, \frac{22}{8}, \frac{24}{10}) \approx 2.2$ gain on platform (c). Thus, only on Xeon Phi the vector execution may give performance improvements over scalar execution—because of the mirroring of lanes $1-4$, at least a factor 2 speedup should be achievable. On Platform (b) we measure $291 \pm 1\,\mu s$ for scalar execution, and $297 \pm 3\,\mu s$ for vector execution. This result meets our expectation: no performance gain with AVX1 and AVX2. On platform (c) we measure $2424 \pm 153\,\mu s$ total execution time, whereas for the vector execution we note $1012 \pm 1\,\mu s$. The gain matches the expected value of 2.2.

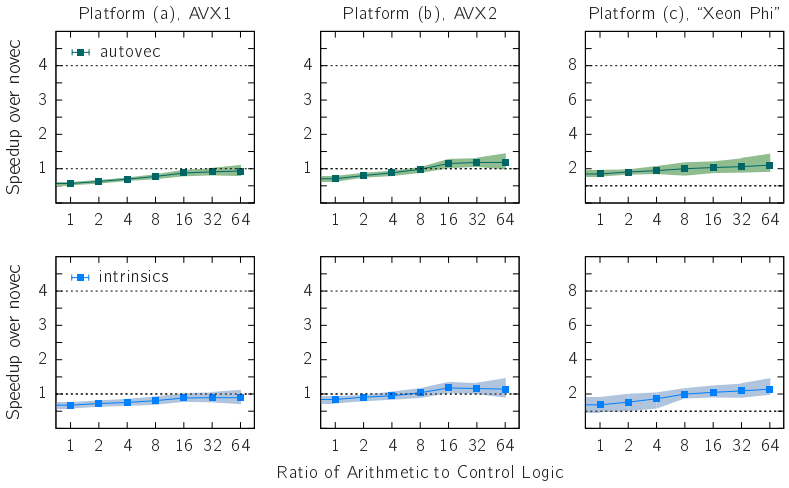
Calling Tree with 2 Different Functions + Early Return:



Calling Tree with 3 Different Functions + Early Return:



Calling Tree with 4 Different Functions + Early Return:



On platform (a) already the case with two different functions along the per-lane calling trees results in the vector performance drops below its scalar counterpart. Platform (b) can only just handle the two-function case, but cannot make it in the other two cases.

Throughout all experiments, the compiler generated vector functions perform almost equal well as our hand-coded intrinsics versions.

4 Summary

We investigated the effectiveness of compiler generated (SIMD-enabled) vector functions in the context of conditional function calls, branching, and early return from function calls. For different kinds of functions and different execution setups, we found the compiler generated vector functions perform almost equal well as manually vectorized functions using SIMD intrinsics. Only in cases where the ratio of arithmetic operations to control logic is low, SIMD intrinsics give measurably larger performance.

We found that "highly" irregular calling trees (together with early returns) can only be handled by the Xeon Phi platform (at the current time), whereas with AVX1 and AVX2 the vector execution performs below the scalar execution.

Acknowledgment

This work has been supported by Intel Corp. within the "Research Center for Many-core High-Performance Computing" at Zuse Institute Berlin.