2017 IEEE International Conference on Cluster Computing

Preliminary Performance Evaluation of Application Kernels using ARM SVE with Multiple Vector Lengths

Yuetsu Kodama*, Tetsuya Odajima*, Motohiko Matsuda*, Miwako Tsuji*, Jinpil Lee* and Mitsuhisa Sato*
*RIKEN Advanced Institute for Computational Science
Email: yuetsu.kodama,tetsuya.odajima,m-matsuda,miwako.tsuji,jinpil.lee,msato@riken.jp

***Abstract*—Modern high performance processors are equipped with very wide SIMD instruction set. SVE (Scalable Vector Extension) is an ARM® SIMD technology that supports vector lengths from 128 bits to 2048 bits. One of its promising features is to offer "vector-length agnostic" programming to allow the same SVE code to run on hardware of any vector length without any modification of the code. This feature would be useful to explore the best vector length with appropriate hardware resources in the space of various combinations of hardware parameters in order to make more efficient use of hardware resources, since we can use the same vectorized SIMD code. In this paper, we report the performance of application kernels using ARM SVE with multiple vector lengths while keeping the hardware resource the same. We have confirmed that when the performance of the program is limited by a bottleneck of a long chain of arithmetic operations or instruction issues, the performance can be improved by increasing the vector length. However, it was necessary to prepare a sufficient number of physical registers for performance improvement, and when the number of physical registers was too small, it was found that with such a program, the performance might be reduced. When the performance is limited by memory access bandwidth to cache and memory, the vector length does not affect the performance significantly.**

1. Introduction

Recent processors improve their performance by using a large number of cores and very wide SIMD instructions. Intel's latest Xeon Phi processor (Knights Landing) [1] supports over 60 cores and AVX512 with a vector length of 512 bits. ARM also recently announced a vector instruction set named SVE (Scalable Vector Extension) [2], [3], [4]. It uniformly supports vector lengths from 128 bits to 2048 bits in units of 128 bits, enabling assembler programs to be independent of vector length. It was announced that the Japanese flagship supercomputer, post-K computer, will adopt ARM SVE as the SIMD instruction set for its many-core processor.

In this paper, we report the performance of application kernels using ARM SVE with multiple vector lengths and explore the best vector length with appropriate hardware resources in the space of combinations of various hardware parameters in order to make use of hardware resources more efficiently. The ARM SVE instruction set offers a *vector-length agnostic* program to allow the same SVE code to run on hardware of any vector length without any modification of the code.

Generally speaking, as the SIMD width is larger, larger numbers of elements in a vector can be computed at the same time, resulting in improvement of the peak performance. However, in order to compute more elements in the SIMD arithmetic unit, a larger amount of hardware resources is required. Since the hardware resources used for arithmetic units occupy a large part of the total hardware resources, the number of elements to be computed in the SIMD unit at the same time is critical in processor design. On the other hand, it is possible to support a vector size exceeding the hardware SIMD size of the arithmetic unit by connecting multiple arithmetic units or by executing one arithmetic unit for multiple cycles. In addition to the number of the arithmetic units for the SIMD, the out-of-order resources such as the size of ROB (reorder buffer) and the number of hardware registers are important factors that affect the performance of modern microprocessors.

It is rather trivial to claim that the performance can be easily improved by increasing hardware resources. In this paper, we are interested in how to improve the performance by changing the SIMD vector lengths, keeping the amount of hardware resources the same, or increasing some part of the hardware resources. We are also interested in which hardware resources are significant for improving the performance. The SVE *vector-length agnostic* feature enables us to evaluate the performance using the same code.

Whether such support is effective or not depends on the program to be executed, so we examine what kind of program can benefit from this approach, and how much such programs can gain from it. We also evaluate the effect of several resource amounts.

In the following sections, we describe the ARM SVE instruction set in Section 2 and the simulator and compiler which comprises the evaluation environment in Section 3. Section 4 shows the evaluation results, the results are discussed in Section 5, and we conclude the paper in Section 6.

2168-9253/17 $31.00 © 2017 IEEE
DOI 10.1109/CLUSTER.2017.93

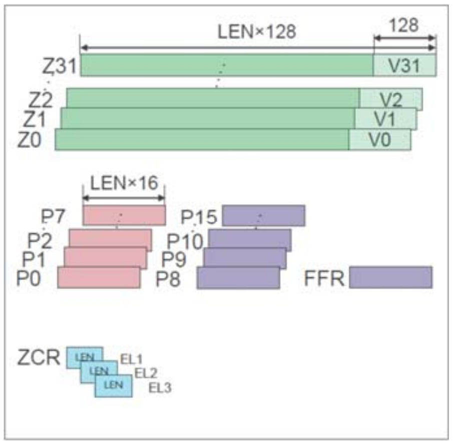

Figure 1. SVE registers

2. Overview of ARM Scalable Vector Extension

SVE is an HPC SIMD extension to the A64 instruction set of the ARMv8-A architecture. The instruction reference manual has been made public for further details in [4]. Its primary feature is supporting vector lengths from 128 bits to 2048 bits in 128-bit increments. Thanks to its *vector-length agnostic* design, the iteration of a SIMD loop can be controlled regardless of the vector length of the target processor. Also, in order to efficiently use a wide range of SIMD instructions, it supports predicate registers that specify whether to execute an instruction for each element in many instructions.

2.1. SVE registers

Figure 1 shows the registers of SVE. There are 32 vector registers from Z0 to Z31, and the size of each vector register is represented by LEN $\times$ 128 bits. LEN is the vector length to be supported, which is different for each machine. LEN is kept in the system register and implicitly used by the instruction set. Each vector register contains the corresponding NEON$^{\text{TM}}$ SIMD register in its lower 128 bits. For example, when LEN=4, a vector register can hold 8 elements of double-precision floating point numbers, 16 elements of single-precision and 32 elements of half-precision. In the case of an integer, it can hold 8, 16, 32 and 64 elements of 64 bits, 32 bits, 16 bits and 8 bits, respectively.

There are 16 predicate registers from P0 to P15 and they specify the elements to be operated on within a vector register. It has 16 bits per LEN so that it can specify the predicate for every 8-bit data item.

2.2. Arithmetic instructions with predicates

Figure 2 shows an example of an instruction with predicates. This is an example of *'fmla'*, one of the fma (floating point fused multiply-add) instructions. For each double-precision vector element it multiplies the first and second operands, adds the product to the third operand, and writes back the sum to the third operand. However, because a predicate is specified, only when the corresponding predicate is TRUE (1), the operation is performed. When it is FALSE (0), the value of the original third operand element is not changed (called merging predication). The operation, when the predicate is FALSE, depends on the instruction. There are instructions which set the corresponding destination element to 0 when the predicate element is FALSE (called zeroing predication).

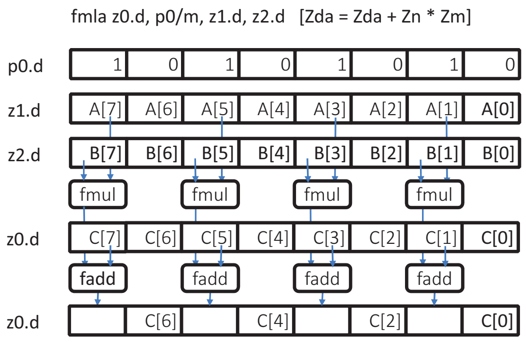
fmla z0.d, p0/m, z1.d, z2.d [Zda = Zda + Zn * Zm]
p0.d
1 0 1 0 1 0 1 0
z1.d
A[7] A[6] A[5] A[4] A[3] A[2] A[1] A[0]
z2.d
B[7] B[6] B[5] B[4] B[3] B[2] B[1] B[0]
fmul fmul fmul fmul
z0.d
C[7] C[6] C[5] C[4] C[3] C[2] C[1] C[0]
fadd fadd fadd fadd
z0.d
C[6] C[4] C[2] C[0]

Figure 2. Operation of fma instruction with predicates

In this example, since only three operands are specified, one of the source operands is overwritten. If you want to save the original value, you need to copy that value to another register. For that purpose, the SVE provides a *'movprfx'* instruction as a special hint instruction and its execution depends on the implementation. It can be executed as an ordinary *'mov'* instruction or can be executed as a 4 operand instruction by fusing with the next instruction at the decode stage.

2.3. Load instructions

There are two types of load instructions: a continuous load instruction and a gather load instruction.

A continuous load instruction loads a contiguous area from a single memory address from a base address plus an offset into a vector register. There are two kinds of addressing, *scalar plus immediate* and *scalar plus scalar*. In *scalar plus immediate* addressing, the immediate value is multiplied by the size of the vector register and added to the scalar base address, which enables addressing independent of the vector length. In *scalar plus scalar* addressing, a scalar index register is multiplied by the element size to be loaded and added to the base address.

The gather load instruction can specify different addresses for each vector element. There are two kinds of addressing, *vector plus immediate* and *scalar plus vector*. In *vector plus immediate* addressing, it refers to the memory that is separated by a common offset specified by an immediate value from the base address in each element of the vector register. In *scalar plus vector* addressing, it refers

```
// ----------------------------------------
// subroutine saxpy(x,y,a,n)
// real*4 x(n),y(n),a
// do i = 1,n
// y(i) = a*x(i) + y(i)
// enddo
// ----------------------------------------
// x0 = &x[0], x1 = &y[0], x2 = &a, x3 = &n
saxpy_:
         ldrsw x3, [x3]                 // x3=*n
         mov x4, #0                     // x4=i=0
         whilelt p0.s, x4, x3           // p0=while(i++<n)
         ld1rw z0.s, p0/z, [x2]         // p0:z0=bcast(*a)
.loop:
         ld1w z1.s, p0/z, [x0,x4,lsl 2] // p0:z1=x[i]
         ld1w z2.s, p0/z, [x1,x4,lsl 2] // p0:z2=y[i]
         fmla z2.s, p0/m, z1.s, z0.s    // p0?z2+=x[i]*a
         st1w z2.s, p0, [x1,x4,lsl 2]   // p0?y[i]=z2
         incw x4                        // i+=(VL/32)
.latch:
         whilelt p0.s, x4, x3           // p0=while(i++<n)
         b.first .loop                  // more to do?
         ret
```

Figure 3. SAXPY program with vector-length agnostic design

to the memory that is separated from common base register by the offset in each element of the vector register, with an option to shift the offset according the element size to be loaded.

2.4. Vector-length agnostic loop

Figure 3 shows a segment of assembler code of SAXPY independent of vector length. This is the code that was introduced in [3]. The *'whilelt'* instruction generates predicates, incrementing the value of the first operand by the number of elements, comparing it with the second operand and setting the corresponding predicate element to TRUE if it is smaller. The *'b.first'* instruction branches conditionally if the first predicate element is TRUE, that is, if at least one element to be processed still remains. The *'incw'* instruction increments by the number of 32-bit words in a vector. Combining these instructions makes it possible to perform *vector-length agnostic* processing.

3. Simulator and compiler used for evaluation

Since there is currently no computer on which SVE has been implemented, we evaluate SVE architecture using a simulator. We use an open source general-purpose processor simulator named gem5 [5], [6]. Gem5 has various execution modes, such as an atomic mode in which an instruction level simulation is performed, and an O3 mode in which an accurate execution cycle number can be estimated by simulating an out-of-order pipeline. In addition, gem5 supports many existing processor architectures such as Alpha, ARM, SPARC and x86.

3.1. Gem5 simulator

Gem5 has a *full system (fs)* mode and a *system emulation (se)* mode. In the *fs* mode, the Operating System (OS) kernel code is executed by gem5 itself, but the execution speed is slower than *se* mode and the OS image needs to be prepared beforehand. On the other hand, in *se* mode, kernel services such as memory management and system calls are not handled by the simulated processor itself but are executed as services in gem5. As a result, the simulation of the user program can be accurately simulated at the pipeline level faster than in the *fs* mode. In *se* mode, the user program to be executed needs to be statically linked. In this evaluation, we use only *se* mode.

Although SVE is not yet supported in gem5 in the currently published version, we were offered the source code of atomic mode for SVE from ARM and created our own O3 mode for SVE. Specifically, preparation and control of vector registers and predicate registers have been added. In addition, 'opclasses' (opcode classes) for SVE instructions are added so that their latency can be controlled.

With these modifications, it was then possible to execute the arithmetic instructions of SVE even in the O3 mode, but since the memory access instructions call different routines in atomic mode and O3 mode, a further modification was necessary. In the atomic mode, as with the arithmetic operation, the memory access instruction calls the instruction processing routine once at the time of execution. On the other hand, in the O3 mode, the memory access instruction is divided into the processing of generating the cache request and the processing after receiving the data from the cache. We expanded gem5 for SVE to deal with these memory operations in the O3 mode.

In the original atomic mode, memory accesses with SVE instructions are divided into memory accesses for each element in a vector register. We implemented continuous load/store instructions in the O3 mode using single memory requests instead of element-wise requests. We also expanded the cache mechanism so that only elements whose predicates are TRUE are written for predicated store instructions. On the other hand, the gather load instruction and scatter store instruction are divided into micro-operations for element-wise memory access.

In the original gem5, the arithmetic unit can select either complete pipeline operation or no pipeline operation at all. We have enabled throughput control so that a pipeline starts once every n cycles (hereinafter referred to as throughput 1/n). For example, if throughput 1/2 is specified for a 1024-bit arithmetic unit, the throughput becomes equivalent to that of a 512-bit arithmetic unit, which is considered to have been realized using the same amount of hardware resources.

3.2. Out-of-order resource parameters on gem5

The basic architecture of out-of-order execution used in O3 mode is based on Alpha 21264, and it consists of each stage of Fetch / Decode / Rename / Issue / Execute / Write-Back / Commit. The latency and concurrent execution width of each stage, the resource requirements of the computing unit and the latency of each instruction class can be freely specified in the parameter file. The main resource definitions are shown in Figure 4.

```
# ALU Instructions have a latency of 1
class O3_ARM_Int(FUDesc):
  opList = [ OpDesc(opClass='IntAlu', opLat=1),
                      ...
      OpDesc(opClass='VectorExtVExu', opLat=1)]
  count = 2

# Floating point and SIMD instructions
class O3_ARM_FP(FUDesc):
  opList = [ OpDesc(opClass='SimdAdd', opLat=4),
                    ...
      OpDesc(opClass='VectorExtVFp', opLat=5)]
  count = 2
...

# Functional Units for this CPU
class O3_ARM_FUP(FUPool):
FUList = [O3_ARM_Int(),
          O3_ARM_FP(),
          O3_ARM_Load(),
          O3_ARM_Store()]

class O3_ARM_3(DerivO3CPU):
    LQEntries = 16
    SQEntries = 16
    ...
    fetchWidth = 3
    decodeWidth = 3
    renameWidth = 3
    dispatchWidth = 6
    issueWidth = 8
    wbWidth = 8
    commitWidth = 8
    numPhysVectorRegs = 96
    numIQEntries = 64
    numROBEntries = 64
...
# Data Cache
class O3_ARM_DCache(Cache):
    size = '32kB'
    assoc = 4
...

# L2 Cache
class O3_ARM_L2(Cache):
    size = ‘2MB'
    assoc = 16
...
```

Figure 4. Gem5 resource parameters

For example, the integer unit is declared in `class O3_ARM_Int`, and the latency of instructions belonging to the *'IntAlu'* opclass is set to 1. The number of pipelines for integer is set to 2 with *'count=2'*. The same applies to the floating point unit, and the floating point instruction of SVE belonging to *'VectorExtVFp'* sets the latency to 5, and full pipeline execution is specified by default. Also, there are two floating point pipelines. In addition, the maximum number of fetch instructions and the number of issued instructions are set to 3 and 8 as *'fetchWidth'* and *'issueWidth'*, respectively. The number of physical vector registers that is the sum of logical vector registers and rename vector registers is set to 96 as *'numPhyVectorRegs'*. The number of buffers of the reservation station and the reorder buffers are both set to 64 as *'numIQEntries'* and *'numROBEntries'*.

The size of the L1 cache is set to 32 Kbytes with 4 way and the size of the L2 cache is set to 2 Mbytes with 16 way. One load and one store operation are available in a cycle. Continuous load and store instructions access to the L1 cache by the vector length in a cycle, while gather load and scatter store instructions access to it by element-wise in a cycle. The bandwidth of the L2 cache is fixed to 32 bytes per cycle independent of the vector length.

Basically, these were defined based on the file `O3_ARM_v7a.py` originally included in gem5. This parameter seems to correspond to the architecture of the Cortex-A15 [7] processor that is a high-end embedded processor of the ARMv7-A architecture. For HPC and servers, it seems to be for small out-of-order resources, but we decided to use these values as they are, as much as possible. This is to focus the discussion in this paper on the difference depending on the vector length of SVE. However, by changing these resources, the balance of out-of-order performance varies greatly, so how to set these parameters correctly is work for future study.

The main changes that we applied were as follows.

- Instruction latency for SVE was added according to that available for NEON.
- The number of physical registers was increased to 96, assuming it will be halved for LEN=8.
- The number of buffers of the reservation stations and the reorder buffers were both changed from 32 and 40, respectively, to 64 since the original values were somewhat small.

3.3. Compiler

In order to execute the program on gem5, a statically linked binary code is required. We were offered a prototype of an SVE compiler from ARM, which we used. The compiler version number used is as follows. `ARM clang version 1.1 (build number 15) (based on LLVM 3.9.0svn)`

4. Evaluation

4.1. Evaluation settings

In this evaluation, LEN=4 (512 bits) is compared with LEN=8 (1024 bits). When we specify the vector length in gem5, the vector register size is defined accordingly, and the number of arithmetic units which operate simultaneously in accordance with the vector register size is set. Since we want to evaluate the difference in vector length when using almost the same amount of hardware, we set the number of physical registers for LEN=8 to half of that for LEN=4. In addition, we set the throughput of vector operator and L1 cache access for LEN=8 to 1/2, that is, the same throughput for LEN=4. Actually, even if such modifications are made, the amount of hardware required for LEN=8 may increase from that for LEN=4, but we think the increase is small compared with that required for these two.

```
(a) triad
for (i = 0; i < N; i++) {
    a[i] = b[i] + 3.0 * c[i];
}

(b) Nbody
 for (j = 0; j < N; j++) {
        dx = px[j] - x_i;
        dy = py[j] - y_i;
        dz = pz[j] - z_i;
        r2 = (dx * dx) + (dy * dy) + (dz * dz) + EPSILON;
        r = sqrt(r2);
        a = G * m[j] / r2;
        ar = a / r;
        acc_x += dx * ar;
        acc_y += dy * ar;
        acc_z += dz * ar;
 }

(c) Dgemm
 for (i = 0; i < N; i++) {
    for (k = 0; k < N; k++) {
      for (j = 0; j < N; j++) {
        C[i][j] += A[i][k] * B[k][j];
      }
    }
 }
```

Figure 5. Code segments for evaluated application kernels

The following three application kernels were used for the evaluation. Although they are very simple programs, we chose these three as targets of preliminary evaluation because it is easy to understand the characteristics of the programs and it takes less time to evaluate in a cycle-based simulator. In the future, we plan to evaluate more and larger programs. The kernel part of each program is shown in Figure 5.

1) *triad*: is one of the stream benchmarks that measure cache/memory load/store performance. As shown in Figure 5 (a), two memory data locations are read out, one is multiplied by a constant, and the result of the addition is written in another memory data location. Here, all data is in the L2 cache. The data is a double precision floating point number. The load/store ratio is high with respect to the floating point arithmetic, and it is characterized as being L2 cache-bandwidth-limited.
2) *nbody*: is a program for calculating all-to-all interactions of multiple objects. The number of objects was 512, and the data was in the L1 cache. The data is double precision. Figure 5 (b) shows the innermost loop calculating the force from N objects for one object. This kernel has a large amount of computation with respect to load/store and the data dependency between instructions is long.
3) *dgemm*: is a matrix-matrix multiplication program. The matrix size is 256 and it fits in the L2 cache. As shown in Figure 5 (c), we switched loops so that access to the array was continuous and SIMD conversion was made easily. The data is double precision. It is a kernel widely used in scientific computation and it is known that it is computation limited because the computational amount is $O(N^3)$ for data access $O(N^2)$. However, in this program, it is L1 cache-bandwidth-limited because two continuous loads and one continuous store of vector length exist in the inner-most loop while only one SIMD fma instruction exists in the loop.

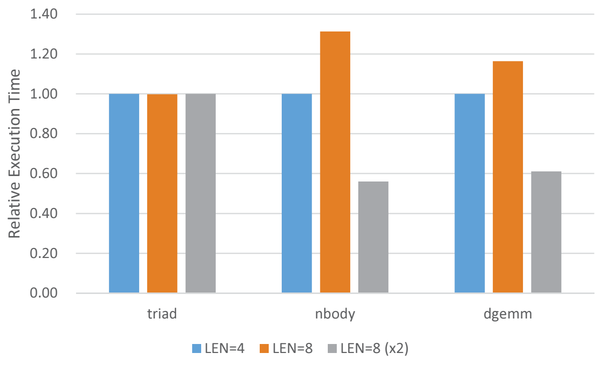

Figure 6. Comparison execution time with different vector length

4.2. Evaluation results

Figure 6 shows the relative execution time with LEN=4 and LEN=8 when the execution time with LEN=4 is 1.0. We use the same binaries of each application kernel for evaluation in LEN=4 and LEN=8. For comparison, we show the result of LEN=8 (x2) that uses the number of physical registers equal to those of LEN=4 (the throughput remains 1/2).

In *'triad'*, the cache bandwidth of L2 is a bottleneck, so even if LEN=8, there is almost no performance change. As described in section 3.2, L2 bandwidth is fixed to 32 bytes/cycle independent of the vector length. In *'nbody'*, when LEN=8, the relative execution time has increased by about 30%. However, when the number of physical registers is the same as that of LEN=4 while the computation throughput remains the same as LEN=4, the relative execution time has been reduced to about 56%. In *'dgemm'*, as in *'nbody'*, when LEN=8, the relative execution time increases by about 16%, but if the number of physical registers is the same as that with LEN=4, it is reduced to about 61%.

Since the influence on the execution time by the number of physical registers is large, we evaluated it by changing the number of physical registers more precisely. For *'triad'*, there was no significant change in execution time in either case. Since the results of *'nbody'* are similar to those of *'dgemm'*, only the results of *'nbody'* are shown in Figure 7. According to Figure 7, for LEN=4, even if the number of physical registers is changed, there is almost no influence on the relative execution time. On the other hand, for LEN=8, it is faster than LEN=4 when the physical register ratio is equal to or larger than 1.25. The relative execution times of 1.75 and 2.00 for LEN=8 are almost the same.

From this evaluation, we have confirmed that when the performance of the program is limited by a bottleneck of a long chain of arithmetic operations or instruction issues,

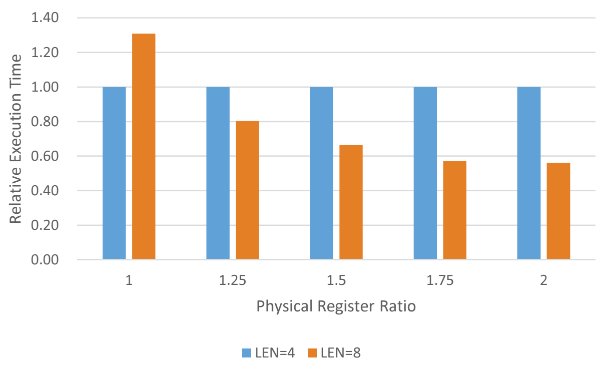

Figure 7. Relative execution time of nbody with different numbers of physical registers

the performance can be improved by increasing the vector length even if the computation unit throughput is almost the same. However, it was necessary to prepare a sufficient number of physical registers for performance improvement, and when the number of physical registers was too small, it was found that with such a program, the performance might be reduced. When the performance is limited by memory access bandwidth to cache and memory, the vector length does not affect the performance significantly.

5. Discussion

5.1. The reason why LEN=8(x2) is faster than LEN=4

We discuss the reasons why performance is improved with LEN=8(x2) even though the computational unit throughput is equivalent. Consider *'dgemm'* as an example. This kernel loop is as shown in Figure 5 (c), and the assembler code of this inner-most loop is shown in Figure 8. As shown in the assembler code, the inner-most loop consists of seven instructions: two load instructions, one fma instruction, one store instruction, one loop counter increment instruction, one predicate generation instruction, and one conditional branch instruction. We considered the arithmetic unit efficiency as follows.

The peak performance in a cycle in LEN=4 is $2 pipelines \times 2FMA \times 8SIMD = 32FLOP/cycle$. The peak performance in LEN=8 is the same as that of LEN=4 because the number of elements is double but the throughput of arithmetic unit is one half. As the results of Figure 6, the performance in LEN=4 is 3.1 FLOP/cycle and the efficiency is only 10%. Since the execution efficiency of the arithmetic unit is less than 50%, even if the throughput of the arithmetic unit remains the same, the performance can be improved by setting LEN=8. This is because even if two cycles are occupied by an fma instruction since the arithmetic unit is not used for the second cycle in LEN=4, the efficiency of the arithmetic unit is increased. In fact, it can be confirmed that the efficiency of the arithmetic unit in LEN=8(x2) is 16.9%, about 1.8 times more efficient than LEN=4.

```
.LBB0_11:                    // %vector.body187
        ld1d    {z1.d}, p0/z, [x13, x14, lsl #3]
        ld1d    {z2.d}, p0/z, [x9, x14, lsl #3]
        fmad    z1.d, p1/m, z0.d, z2.d
        st1d    {z1.d}, p0, [x9, x14, lsl #3]
        incd    x14
        whilelo p0.d, x14, x11
        b.mi    .LBB0_11
```

Figure 8. Assembler code of inner-most loop in dgemm

This *'dgemm'* implementation is not an optimized one. We will develop a more efficient program using blocking and/or unrolling in the future. If the program is well tuned and becomes computation bound, the difference between LEN=4 and LEN=8(x2) may become small.

On the other hand, *'nbody'* is computation limited and the chain of computation is long. Currently *'nbody'* is also not well optimized, and retains not-pipelined SIMD instructions such as sqrt and fdivr because of a prototype compiler issue. The inner-most loop includes 24 instructions which include 11 arithmetic instructions; those operations are estimated as 24 where fma is counted as 2 operations and sqrt and fdivr are counted as 5 operations. The average number of cycles of the inner-most loop is 23 cycles when LEN=4, and 13 cycles when LEN=8(x2) from the results of Figure 6, so the arithmetic unit efficiency is 52% when LEN=4 and 93% when LEN=8(x2), respectively. Since the efficiency is about 50% when LEN=4, there is room to improve performance when two cycles are used for an instruction.

In the current implementation of gem5, the throughput of non-pipelined instructions is not controlled, so the performance on LEN=8 is twice that of LEN=4 in sqrt and fdivr. This is not a fair comparison. So we will develop a more efficient program for *'nbody'* using reciprocal instructions instead of sqrt and fdivr instructions and will evaluate this issue again. But since this program has a long chain of instruction dependency and a limited number of loops are overlapped, the arithmetic unit efficiency will not be high when LEN=4. So we think the difference between LEN=4 and LEN=8(x2) may remain.

5.2. The reason why LEN=8 is slower than LEN=4

Next, we consider the reason why the performance is not improved even if the number of physical registers is increased with LEN=4 and the reason why the performance of LEN=8 with a small number of physical registers is lower than that of LEN=4. To examine such situations, we counted the cycles in which the out-of-order resource became full during kernel loop execution. We checked five resources: a reservation station (IQ), a reorder buffer (ROB), a load queue (LQ), a store queue (SQ), and a physical register file (Reg). Table 1 shows the results when physical registers are increased for *'nbody'* .

As shown in Table 1, when LEN=4, basically, there are many cycles in which ROB is full and other resources are

TABLE 1. THE CYCLE COUNTS WHEN OUT-OF-ORDER RESOURCES ARE FULL ON NBODY

LEN	Reg Ratio	IQFull	ROBFull	LQFUll	SQFull	RegFull
LEN=4	1.00	0	118242	30	0	0
	1.25	0	118242	30	0	0
	1.50	0	118242	30	0	0
	1.75	0	118242	30	0	0
	2.00	0	118242	30	0	0
LEN=8	1.00	0	0	0	0	116868
	1.25	0	0	0	0	49671
	1.50	0	3	56	0	57290
	1.75	0	3148	60	0	64852
	2.00	0	61151	86	0	0

not fully utilized. Therefore, even if the number of physical registers is increased, there is almost no change.

In LEN=8, when the ratio of the number of registers is 1.0, that is, when it is half of the number of registers for LEN=4 (the number of bytes of a register file is the same as in LEN=4), the number of cycles in which the physical register is full is very large, and other resources are not fully utilized.

When the number of physical registers is increased to reduce the load on the physical registers, the performance is improved, and the number of cycles in which the ROB becomes full gradually increases. When the number of registers is doubled, that is, when the number of registers is the same as that of LEN=4, the physical registers never become full and the number of cycles at which the ROB becomes full is very large.

As shown in Table 1, when LEN=8, the number of physical registers initially becomes a bottleneck and the performance is degraded. However, by increasing the number of physical registers, they are no longer a bottleneck and performance is improved. On the other hand, when LEN=4, since ROB is the bottleneck, increasing physical registers has no effect.

In this evaluation, only the number of physical registers is changed, but as discussed above, it is necessary to change and evaluate other out-of-order resources as well. However, if a bottleneck is removed, another resource may become the bottleneck. In addition, it is necessary to estimate the amount of hardware (chip area) involved when each resource is changed. We plan to look at these issues in future work.

6. Conclusion

In this paper, we report the performance of application kernels using ARM SVE with multiple vector lengths to make use of the same amount of hardware resources efficiently. The ARM SVE instruction set is useful to explore the best vector length, since it offers "vector-length agnostic" programming to allow the same SVE code to run on hardware of any vector length without any modifications of the code. Based on our performance evaluation using our gem5 cycle-level simulator of ARM SVE, we have confirmed that when the performance of the program is limited by a bottleneck of a long chain of arithmetic operations or instruction issues, the performance can be improved by increasing the vector length, even if the computation unit throughput is almost the same. However, it was necessary to prepare a sufficient number of physical registers for performance improvement, and when the number of physical registers was too small, it was found that with such a program, the performance might actually be reduced. When the performance is limited by memory access bandwidth to cache and memory, the vector length does not affect the performance significantly.

Our future work will include evaluating more programs and identifying the characteristics of application kernels to determine the appropriate vector lengths for each application. We also plan to evaluate a wider range of parameters on the out-of-order resources used for these performance evaluations.

Acknowledgments

The authors would like to thank Nigel Stephens and his colleagues in ARM Ltd. for fruitful discussion about the SVE instructions and evaluation method.

This work is partially funded by MEXT's program for the Development and Improvement for the Next Generation Ultra High-Speed Computer System, under its Subsidies for Operating the Specific Advanced Large Research Facilities.

References

[1] A. Sodani, R. Gramunt, J. Corbal, H. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agwarwal and Y. Liu, *Knights Landing: Second-Generation Intel Xeon Phi Processor*, IEEE Micro, Vol.36, Issue 2, March/April 2016, pp.34–46.

[2] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu, A. Reid, A. Rico and P. Walker, *The ARM Scalable Vector Extension*, IEEE Micro, Vol.37, Issue 2, March/April 2017, pp.26–29.

[3] D. Brash and N. Stephens, *ARM: Scaling New Heights*, Proc. of Cool Chips 20, keynote 3, April, 2018

[4] https://developer.arm.com/products/architecture/a-profile/docs, *ARM Architecture Reference Manual Supplement - The Scalable Vector Extension (SVE), for ARMv8-A*.

[5] N. Binkert, B. Beckmann, G. Black, S.K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D.R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M.D. Hill and D.A. Wood. *The gem5 Simulator*, ACM SIGARCH Computer Architecture News, Vol.29, Issue 2, May 2011.

[6] http://gem5.org/, *The gem5 Simulator A modular platform for computer-system architecture research*.

[7] https://developer.arm.com/products/processors/cortex-a/cortex-a15, *Cortex-A15 Overview*.