Arm Scalable Vector Extension and application to Machine Learning

Dan Andrei Iliescu, Francesco Petrogalli

November 2017

Contents

Introduction

In this document we present code examples that show how to vectorize some of the core computational kernels that are part of machine learning system.

The examples presented in this document are written with the *Vector Length Agnostic (VLA)* approach introduced by the *Scalable Vector Extension (SVE)*.

In particular, the paper shows how VLA techniques can be used to efficiently vectorize *General Matrix Multiplication (GEMM)* and *low precision GEMM (GEMMlowp)* computational kernels.

1 The Scalable Vector Extension

SVE is a vector extension for AArch64 execution mode for the A64 instruction set of the Armv8 architecture. Unlike other SIMD architectures, SVE does not define the size of the vector registers, but constrains it to a range of possible values, from a minimum of 128 bits up to a maximum of 2048 in 128-bit wide units. Therefore, any CPU vendor can implement the extension by choosing the vector register size that better suits the workloads the CPU is targeting. The design of SVE guarantees that the same program can run on different implementations of the ISA without the need to recompile the code.

Most of the instructions of the extension also use predicate registers to mask the lanes for operating on partial vectors. The SVE instruction set also provides gather loads and scatter stores, plus truncating stores, and signed/unsigned extended loads.

The following documents describing the architecture extension are available:

- SVE architecture reference manual [1], which defines the instructions and the registers;
- *A Sneak Peak to SVE and VLA programming* [3], a whitepaper with assembly examples of loops vectorized with the SVE instructions;

- *The ARM Scalable Vector Extension* [4].

This document focuses on the interface at C/C++ level for SVE that is provided via the SVE Arm C Language Extensions (SVE ACLE) [2].

2 SVE ACLE

The SVE ACLE (or ACLE hereafter) is a set of functions and types that exposes the vectorization capabilities of SVE at C/C++ level.

They introduce a set of *size-less* types and *intrinsic functions* that a C/C++ compiler can directly convert into SVE assembly. The function-to-instruction mappings are not one to one, as some of the architectural details of the instruction set can be resolved by a compiler. For example, there is no need to expose at C/C++ level some of the addressing modes of the load and store instructions of SVE.

The ACLE introduces size-less data types in the form `sv[type]`, where `sv` stands for *Scalable Vector* and `type` can be any of the scalar types supported by the lanes of the SVE vectors. These size-less types cover SVE vectors consisting of of 8, 16, 32 and 64 bit lanes for signed and unsigned integral types, and 16, 32 and 64 bit lanes for floating point types:

- `sv[u]int[8|16|32|64]_t`;
- `svfloat[16|32|64]_t`.

An additional `svbool_t` type is defined to represent predicates for masking operations. The predicate type carries one bit for each byte in the data types.

SVE intrinsic functions

The naming convention of the intrinsic functions in the SVE ACLE is described in detail in section 4 of the SVE ACLE document [2].

Most of them are in the form: `svbase[_disambiguator][_type0][_type1]...[_predication]`.

For example, the name of the intrinsic `svadd_n_u16_ma`, with signature `svuint16_t svadd_n_u16_m(svbool_t pg, svuint16_t op1, uint16_t op1)`, describes a vector *addition* (`add`) of *unsigned 16-bit integer* (`u16`), where one of the arguments is a scalar (`_n`) and the predication mode is *merging* (`_m`).

Some of the functions, like loads and stores, have a different form for the names, with additional tokens that specify the addressing mode. For example, the function `svld1_gather_u32base_offset_s32`, with signature

`svint32_t svld1_gather_u32base_offset_s32(svbool_t pg, svuint32_t bases, int64_t offset)`

is a *gather load* (`ld1_gather`) of *signed 32-bit integer* (`_s32`) from a vector of *unsigned 32-bit integer* base addresses (`_u32base`) plus an *offset in bytes* (`_offset`).

The SVE ACLE are compatible with C++ overloading and C `_Generic` association, so that the names can be contracted removing those parts that can be derived from the arguments types. For example, the `svadd_n_u16_m` can be contracted to `svadd_m`, and `svld1_gather_u32base_offset_s32` can be contracted to `svld1_gather_offset`.

All the examples of this document use the short form. For simplicity, we also assume no aliasing, meaning that all the pointers passed as function parameters are to be considered as `restrict` pointers.

Example

A basic example that shows how to transform a scalar loop into VLA form using the SVE ACLE is shown in listing 2.1.

The vector version in `vla_add_arrays` of listing 2.1 works as follows.

Listing 2.1 VLA vectorization example using the SVE ACLE.

```
1  // Scalar version.
2  void add_arrays(double *dst, double *src, double c, const int N) {
3    for (int i = 0; i < N; i++)
4      dst[i] = src[i] + c;
5  }
6
7  // Vector version
8  void vla_add_arrays(double *dst, double *src, double c, const int N) {
9    svfloat64_t vc = svdup_f64(c);
10   for (int i = 0; i < N; i += svcntd()) {
11     svbool_t Pg = svwhilelt_b64(i, N);
12     svfloat64_t vsrc = svld1(Pg, &src[i]);
13     svfloat64_t vdst = svadd_x(Pg, vsrc, vc);
14     svst1(Pg, &dst[i], vdst);
15   }
16 }
```

First, the constant `c` is reproduced into all the lanes of a vector `vc`, with the `svdup_f64` function (line 9). Note that although we are using the short form of the ACLE, the `_f64` part in the name is required, because standard C scalar promotion does not allow the contraction of the name of those functions that process only scalar arguments (see section 4.2 of [2] for a detailed explanation).

Next, the header of the vector loop is issued (line 10). The number of lanes that one iteration of the vector loop can process is unknown at compile time. This means that the induction variable `i` needs to be incremented dynamically with the `svcntd()` function, which returns the number of 64-bit (double-word) lanes in an SVE vector type, or `VL.D` hereafter.

In the body of the loop, the predicate `Pg` is set with the `whilelt_b64` function (line 4). This function builds a predicate by testing the `i < N` inequality for all the values of the induction variable spanning the iteration of the vector loop and associating its result to the correspondent lane of the vector register. At iteration `i`, it computes `j < N` for `j=i, i+1, ..., i+VL.D-1`. The 64-bit lanes view of the predicate is specified by the `_b64` part of the intrinsic, which cannot be contracted into the intrinsic name because of C scalar promotion.

On a 256-bits implementation, the value of the predicate `Pg` would look as follows in the second iteration of the loop in the example where `N = 7`:

```
     MSB                                LSB
Pg = [00000000 00000001 00000001 00000001]
             7        6        5        4  64-bit lanes index 'i'
```

Using the predicate `Pg` effectively removes the need to deal with the remainder of the loop that would not fit in a full vector. The predicate values over the full loop iteration for the 256-bit example where `N = 7` are shown in figure 1.

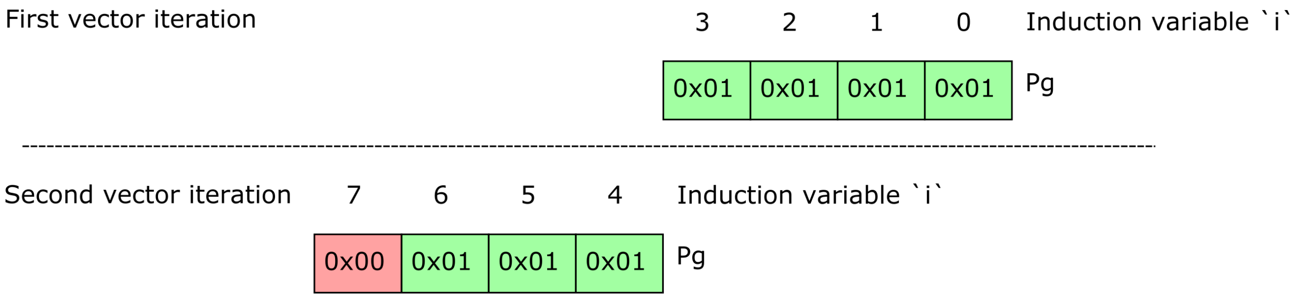

Figure 1: Predication.

3 ML algorithms with VLA SVE

Matrix multiplication

Matrix multiplication is one of the most widely used operations in machine learning. It is a binary operation that takes as input two matrices (A of dimensions (M, K) and B of dimensions (K, N)) and returns as output one matrix (C of dimensions (M, N)) in which every element C_{ij} is the dot product of row i of matrix A with row j of matrix B.

The dot product of two arrays, X and Y, of equal length, N, is computed by multiplying each element of one array with its corresponding element in the other array, and then summing up all the products.

$$Z = \sum_{n=1}^{N} X_n * Y_n$$

The dot product of two arrays can be implemented in C as in listing 3.1.

Listing 3.1 C implementation of the dot product of two arrays of the same size.

```c
double dot(double *X, double *Y, int N) {
  double Z = 0;
  for (int n = 0; n < N; n ++)
    Z += X[n] * Y[n];
}
```

To multiply two matrices, we perform the dot product for every row of matrix A with every column of matrix B.

$$\forall i \in [1 : M],\ \forall j \in [1 : N],\ C_{i,j} = \sum_{k=1}^{K} A_{i,k} * B_{k,j}$$

In this document we assume that matrices of size (S, T) are implemented as C arrays of length $S * T$ where element (s, t) is at memory offset $(s * T + t)$. Using this convention, matrix multiplication can be implemented as in listing 3.2.

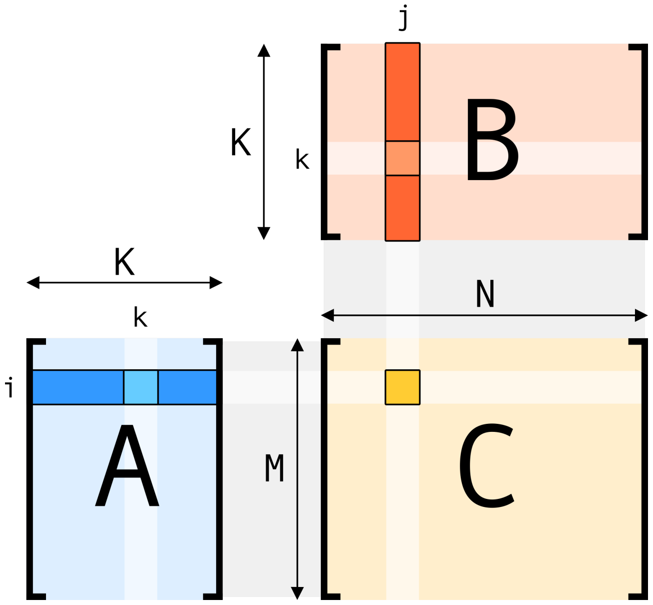

Figure 2: Matrix Multiplication. The blue row of A is multiplied with the red column of B to produce the yellow element of C.

Listing 3.2 C implementation of matrix multiplication.

```c
void mm(double *C, double *A, double *B, long M, long N, long K) {
  for (int i = 0; i < M; i ++)
    for (int j = 0; j < N; j ++) {
      C[i * N + j] = 0;
      for (int k = 0; k < K; k ++)
        C[i * N + j] += A[i * K + k] * B[k * N + j];
    }
}
```

Simple vectorization

Matrix multiplication has great potential for parallelization due to the repetitive nature of the operations involved and the layout of the data. For instance, each element of row i of the resulting matrix C is the dot product of row i of matrix A with the corresponding column of matrix B. We can therefore compute a batch of VL elements of row i of matrix C at once by computing the dot products of row i of matrix A with a batch of VL columns of matrix B figure 3.

For describing the algorithm of the examples in this and the following sections, we introduce the notation $A_{a,[b:c]}$ to represent the sub-matrix of a matrix A containing the elements of row a between columns b and c.

Let's considers a batch of VL columns of matrix B as a single-column matrix of size $(K, 1)$ whose elements are arrays of size VL. If we multiply each of these arrays of matrix B with the corresponding scalar element of matrix A, and then sum up the results, we get an array of size VL which is equivalent to the batch of VL elements of row i of the resulting matrix C.

$$C_{i,[j:j+VL]} = \sum_{k=1}^{K} A_{i,k} * B_{k,[j:j+VL]}$$

We achieve this algorithmically by carrying a rectangular *stencil*, $[j : j + VL]$, of size (K, VL) (where VL is the number of elements in an SVE register) in steps of VL over matrix B. The stencil starts at column j and sits over at most VL columns of matrix B. The columns of matrix B that are contained within the stencil represent the batch of VL columns that will be multiplied with row i of matrix A to result in its corresponding batch of elements of row i of matrix C figure 3.

So, for every row i in matrix A, and for every stencil $[j : j + VL]$ in matrix B, we record in a predicate how many columns of matrix B are actually contained within the stencil (in case N is not a multiple of VL and the current stencil goes over the border of the matrix), then we compute the dot product between row i and columns $[j : j + VL]$ by multiplying the corresponding elements and adding them up, and we store the result in $C_{i,[j:j+VL]}$.

The code shown in listing 3.3 presents a vectorized version of a half-precision (16-bit floating point) matrix multiplication.

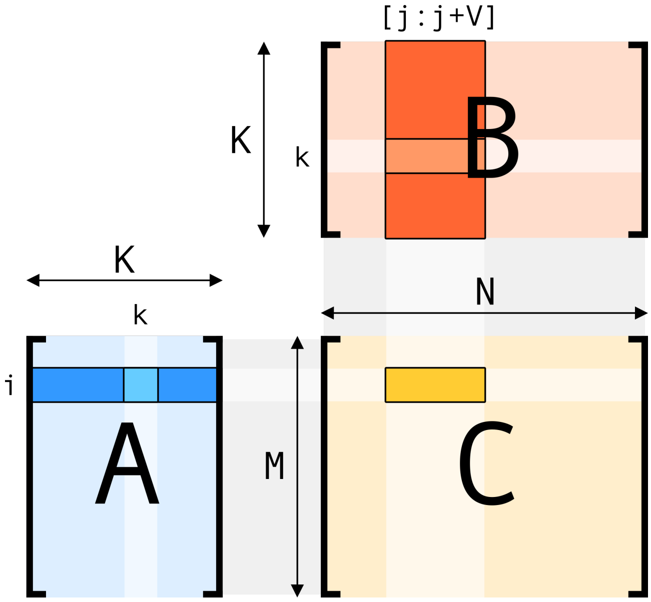

Figure 3: Vectorized Matrix Multiplication. The blue row of A is multiplied with the columns in the red stencil of B to produce the yellow vector of C.

Listing 3.3 Half-precision matrix multiplication with VLA vectorization.

```
 1  void hgemm_01(float16_t *C, float16_t const *A, float16_t const *B,
 2                const unsigned long M, const unsigned long K,
 3                const unsigned long N) {
 4    for (unsigned long i = 0; i < M; ++i)
 5      for (unsigned long j = 0; j < N; j += svcnth()) {
 6        svfloat16_t Acc = svdup_f16(0);
 7        const svbool_t pred_j = svwhilelt_b16(j, N);
 8        for (unsigned long k = 0; k < K; ++k) {
 9          const svfloat16_t A_i_k = svdup_f16(A[i * K + k]);
10          const svfloat16_t B_k_j = svld1_f16(pred_j, &B[k * N + j]);
11          Acc = svmla_x(pred_j, Acc, A_i_k, B_k_j);
12        }
13        svst1_f16(pred_j, &C[i * N + j], Acc);
14      }
15  }
```

Line 5: `svcnth()` is an SVE ACLE function that returns the value `VL.H`, representing the number of lanes of an SVE vector with 16-bit (H, from *half-word*) subdivision.

Line 7: Mark the elements of the predicate `pred_j` with a 2-bit subdivision as active if the corresponding element in the current stencil sits inside the matrix B, and as inactive if the element is outside the borders of the matrix. We will end up with `min(VL.H, N-j)` active elements. This is implemented with the SVE ACLE function `p = svwhilelt(a, b)` which activates the n-th element of p if and only if a+n `<=` b (see figure 4).

Line 11: Add to the accumulator `Acc`, that was set to zero on line 6, the dot product of an array of `VL.H` copies of element `A[i][k]` with the array `B[k][j:j+VL.H]`. The SVE ACLE function `a = svmla_x(pred, b, c, d)` assigns to a the value b+c*d. This accumulation operation is guarded by the predicate `pred_j`, which guarantees that the inactive elements are not processed.

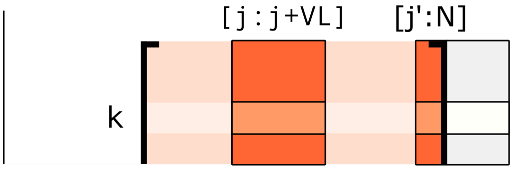

Figure 4: Stencil Overflow. The lanes of the stencil on the right that overflow the border of the matrix are not used for computation. Therefore, they are set inactive.

Unrolled vectorization

A further improvement (from the perspective of using more registers) to the computation of the matrix product implemented in [#lst:hgemm_01] is to use two consecutive stencils, `[j:j+VL.H-1]` and `[j+VL.H:j+2*VL.H-1]`, and, thus, compute `2*VL.H` elements of the matrix `C` at once (see listing 3.4, and figure 5 for a graphical representation of the unrolling of the stencils).

Listing 3.4 Half-precision unrolled matrix multiplication with VLA vectorization.

```
1  void hgemm_unrolled(float16_t *C, float16_t const *A, float16_t const *B,
2                             const unsigned long M, const unsigned long K,
3                             const unsigned long N) {
4    const svbool_t all_active = svptrue_b16();
5
6    for (unsigned long i = 0; i < M; ++i)
7      for (unsigned long j = 0; j < N; j += 2 * svcnth()) {
8        svfloat16_t Acc = svdup_f16(0);
9        svfloat16_t Acc_1 = svdup_f16(0);
10       const svbool_t pred_j_1 = svwhilelt_b16(j + svcnth(), N);
11       const svbool_t pred_j = svptest_first(all_active, pred_j_1)
12                                   ? all_active
13                                   : svwhilelt_b16(j, N);
14       for (unsigned long k = 0; k < K; ++k) {
15         const svfloat16_t A_i_k = svdup_f16(A[i * K + k]);
16         const svfloat16_t B_k_j = svld1_vnum_f16(pred_j, &B[k * N + j], 0);
17         const svfloat16_t B_k_j_1 = svld1_vnum_f16(pred_j_1, &B[k * N + j], 1);
18         Acc = svmla_x(pred_j, Acc, A_i_k, B_k_j);
19         Acc_1 = svmla_x(pred_j_1, Acc_1, A_i_k, B_k_j_1);
20       }
21       svst1_vnum_f16(pred_j, &C[i * N + j], 0, Acc);
22       svst1_vnum_f16(pred_j_1, &C[i * N + j], 1, Acc_1);
23     }
24 }
```

Line 4: The SVE ACLE function `svptrue_b16()` returns a vector predicate of all active lanes, with a 16-bit data subdivision.

Line 11: We can save unnecessary computation by not calling the `svwhilelt_b16()` function for the first stencil when we know that it fits completely into the matrix boundaries. We can assume this happens if the second stencil sits over at least one column of `B`. This can be verified by testing if the first lane of the predicate corresponding to the second stencil (`pred_j_1` in line 10) is active, by using the SVE ACLE function `svptest_first(a, b)`. This tests if the first element of predicate `b` is active.

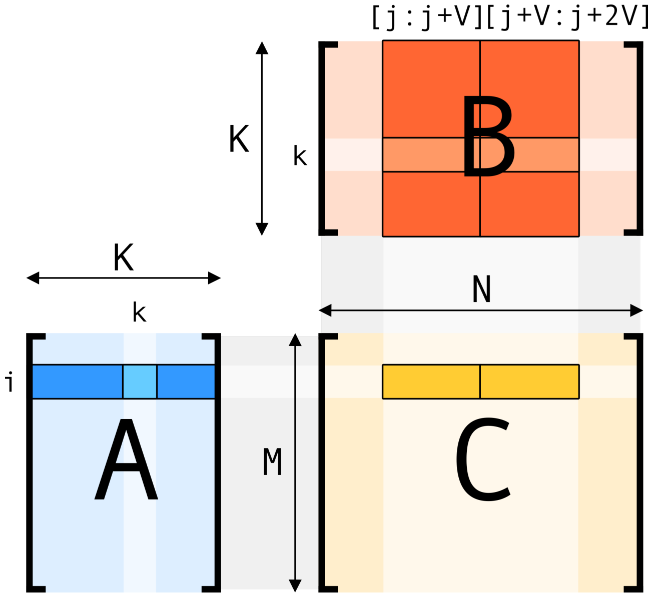

Figure 5: Unrolled Vectorized Matrix Multiplication.

Dot products

An array of matrices X of dimensions (A, B, C) is an array of A matrices with B rows and C columns. Element $X^a_{b,c}$ is the element of matrix a that sits on row b and column c.

A dot product of two arrays of matrices A of dimensions (W, M, K) and B of dimensions (W, K, N), is a matrix C of dimensions (M, N) where each element $C_{i,j}$ is the sum of the dot products of each row i from the matrices of array A with the column j from the corresponding matrix of array B.

$$\forall i \in [1:M], \quad \forall j \in [1:N], \quad C_{i,j} = \sum_{n=1}^{W} \sum_{k=1}^{K} A^n_{i,k} * B^n_{k,j}$$

Due to commutativity, we can switch the sums around and, thus, treat the result $C_{i,j}$ as the sum of the dot products of each element of row i from matrix A with its corresponding element of column j from matrix B, where the elements of matrices A and B are arrays of length W.

$$\sum_{n=1}^{W} \sum_{k=1}^{K} A^n_{i,k} * B^n_{k,j} = \sum_{k=1}^{K} \sum_{n=1}^{W} A^n_{i,k} * B^n_{k,j}$$

In machine learning, it is very common to process arrays of matrices of 8-bit unsigned data to obtain 32-bit results via such operations, without loss of information due to the accumulation of the product of 8-bit data into the 32-bit accumulator.

We can implement these matrices of arrays as normal matrices of 32-bit elements, but where each 8-bit lane of each element is an element in its own (see figure 6). We will refer to the 32-bit lanes as *quadruplets* of 8-bit elements. A C++ implementation of the GEMMlowp algorithm is shown in listing 3.5.

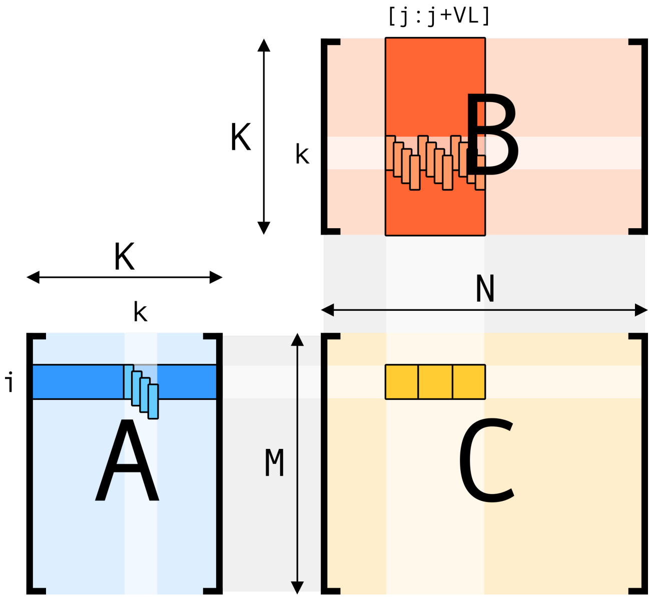

Figure 6: Low Precision

Listing 3.5 C++ implemetation of GEMMlowp

```
void gemmlowp_c(uint32_t *C, const uint32_t *A, const uint32_t *B, size_t M,
                size_t K, size_t N) {
  for (size_t i = 0; i < M; ++i)
    for (size_t j = 0; j < N; ++j)
      for (size_t k = 0; k < K; ++k) {
        uint8_t *dataA = (uint8_t *)&A[i * K + k];
        uint8_t *dataB = (uint8_t *)&B[k * N + j];
        uint32_t tmp = 0;
        for (size_t s = 0; s < 4; ++s, ++dataA, ++dataB)
          tmp += static_cast<uint32_t>(*dataA) * static_cast<uint32_t>(*dataB);

        C[i * N + j] += tmp;
      }
}
```

The GEMMlowp code in listing 3.5 can be vectorized by processing more than one multiplication from the dot product at a time. This implies loading VL elements from a column in matrix B, performing the multiplications and then summing up the results listing 3.6.

Listing 3.6 VLA version of GEMMlowp

```
1  void gemmlowp_acle(uint32_t *C, const uint32_t *A, const uint32_t *B, size_t M,
2                     size_t K, size_t N) {
3    const svint32_t index = svindex_s32(0, N);
4  
5    for (size_t i = 0; i < M; ++i)
6      for (size_t j = 0; j < N; ++j)
7        for (size_t k = 0; k < K; k += svcntw()) {
8          const svbool_t pred_k = svwhilelt_b32(k, K);
9          const svuint32_t vA = svld1_u32(pred_k, &A[i * K + k]);
10         const svuint32_t vB =
11             svld1_gather_s32index_u32(pred_k, &B[k * N + j], index);
12         svuint32_t tmp = svdot_u32(svdup_u32(0), svreinterpret_u8_u32(vA),
13                                    svreinterpret_u8_u32(vB));
14         C[i * N + j] += svaddv_u32(pred_k, tmp);
15       }
16 }
```

The code in listing 3.6 works as follow.

Line 3: Instruction `svindex_s32(0, N)` builds a vector register where each element is equal to its index times N.

Line 11: In order to load part of a column from B into a vector register, we need to load the first element from the address `B[k * N + j]`, followed by `VL-1` elements in the same column, which are stored N words apart from each other. This requires loading non-consecutive addresses, which can be done using `svld1_gather_s32index_u32()`.

Line 12: The `svdot_u32` operation performs the dot products of each quadruplets in its two 8-bit input registers, and stores the results in the correspondent 32-bit lane of the output vector register, without loss of precision.

Line 14: `svaddv_u32()` sums all the elements in the argument vector into a scalar.

In the next implementation in listing 3.7, we optimise the process (from the perspective of increasing register pressure) by loading four quadruplets from A at a time and computing the dot products with VL quadruplets from B.

Listing 3.7 GEMMlowp with higher register pressure

```
 1  void gemmlowp_acle_v3(uint32_t *C, const uint32_t *A, const uint32_t *B,
 2                        size_t M, size_t K, size_t N) {
 3    for (size_t i = 0; i < M; ++i)
 4      for (size_t j = 0; j < N; j += svcntw()) {
 5        const svbool_t pred_j = svwhilelt_b32(j, N);
 6        svuint32_t tmp = svdup_u32(0);
 7        size_t k = 0;
 8        if (K > 3)
 9          for (; k < K - 3; k += 4) {
10            svuint32_t vA = svld1rq(svptrue_b32(), &A[i * K + k]);
11            const svuint32_t vB0 = svld1_u32(pred_j, &B[k * N + j]);
12            const svuint32_t vB1 = svld1_u32(pred_j, &B[(k + 1) * N + j]);
13            const svuint32_t vB2 = svld1_u32(pred_j, &B[(k + 2) * N + j]);
14            const svuint32_t vB3 = svld1_u32(pred_j, &B[(k + 3) * N + j]);
15            tmp = svdot_lane_u32(tmp, svreinterpret_u8_u32(vB0),
16                                 svreinterpret_u8_u32(vA), 0);
17            tmp = svdot_lane_u32(tmp, svreinterpret_u8_u32(vB1),
18                                 svreinterpret_u8_u32(vA), 1);
19            tmp = svdot_lane_u32(tmp, svreinterpret_u8_u32(vB2),
20                                 svreinterpret_u8_u32(vA), 2);
21            tmp = svdot_lane_u32(tmp, svreinterpret_u8_u32(vB3),
22                                 svreinterpret_u8_u32(vA), 3);
23          }
24        // process tail
25        for (; k < K; k += 1) {
26          svuint32_t vA = svdup_u32(A[i * K + k]);
27          const svuint32_t vB = svld1_u32(pred_j, &B[k * N + j]);
28          tmp = svdot_u32(tmp, svreinterpret_u8_u32(vA), svreinterpret_u8_u32(vB));
29        }
30        svst1_u32(pred_j, &C[i * N + j], tmp);
31      }
32  }
```

The code in listing 3.7 works as follows.

Line 10: We want to load four 32-bit elements from A and repeat them in the register. The instruction svld1rq(), *load replicate*, fills every 128-bit subdivision of the vector register with the same four 32-bit elements from the memory address it is provided. A rendering of a vector loaded from address int *X is shown in listing 3.8.

Line 15: We want to compute the dot product of each of the four 32-bit quadruplets of 8-bit elements from A with every 32-bit quadruplet of 8-bit elements from B, and store the results as 32-bit numbers in a vector register. The instruction svdot_lane_u32() will compute the dot product of each 32-bit quadruplet of the second argument with the indicated quadruplet of the third argument, without loss of precision. The quadruplet of each 128-bit chunk of A is selected with an index from 0 to 3.

Listing 3.8 Example of load replicate from a pointer to 32-bit data.

```
                   ... |<----- 128 bits ---->|<----- 128 bits ---->|
Lane index: VL.S-1 ... |    7    6    5    4 |    3    2    1    0 |
Value:        X[3] ... | X[3] X[2] X[1] X[0] | X[3] X[2] X[1] X[0] |
```

4 Tools

The code snippets presented in this whitepaper can be compiled to SVE object files using *Arm Compiler for HPC*, which fully support the SVE ACLE. The intrinsics are recognised automatically when `-march=armv8+sve`.

```
$> armclang -march=armv8+sve -c filename.c -o objectfile.o
```

Arm Compiler also supports automatic vectorisation that targets SVE VLA vectorisation techniques when `-march=armv8+sve` is invoked in conjunction with the usual flag that enable the loop vectoriser, `-floop-vectorze`, which is enabled by default from the optimisation level `-O2`.

As part of our simulation tools, a user space simulator is also provided to emulate programs that include SVE instructions. The simulator, called *Arm Instruction Emulator (ArmIE)*, execute natively on AArch64 hardware and simulate only the SVE instructions.

ArmIE can execute SVE instruction for any size of the vector registers, from the minimal 128-bit to the maximum 2048-bit width. For example, the simulation of an SVE executable `program` for a 256-bit implementation of SVE is invoked as follows:

```
$> armie -msve-vector-bits=256 ./program
```

Both Arm Compiler for HPC and ArmIE are available on the Arm Developer[1] website.

[1] https://developer.arm.com/hpc

Acknowledgements

The authors would like to thank the following colleagues for their help and advice: Richard Sandiford, David Mansell, Geraint North, and Julie Gaskin.

Trademarks

The trademarks featured in this document are registered and/or unregistered trademarks of Arm Limited (or its subsidiaries) in the EU and/or elsewhere. All rights reserved. All other marks featured may be trademarks of their respective owners. For more information, visit Arm website[2].

References

[1] ARM limited, ed. *ARM Architecture Reference Manual Supplement - The Scalable Vector Extension (SVE), for ARMv8-A*. 2017. URL: https://developer.arm.com/products/architecture/a-profile/docs/arm-architecture-reference-manual-supplement-armv8-a.

[2] ARM limited, ed. *ARM C Language Extensions for SVE*. 2017. URL: https://developer.arm.com/docs/100987/latest/arm-c-language-extensions-for-sve.

[3] Francesco Petrogalli. *A sneak peek into SVE and VLA programming*. Nov. 2016. URL: https://developer.arm.com/hpc/a-sneak-peek-into-sve-and-vla-programming (visited on 10/14/2017).

[4] Nigel Stephens et al. “The ARM Scalable Vector Extension”. In: *IEEE Micro* 37.2 (Mar. 2017), pp. 26–39. DOI: 10.1109/mm.2017.35[3]. URL: https://doi.org/10.1109/mm.2017.35.

[2] http://www.arm.com/about/trademarks

[3] https://doi.org/10.1109/mm.2017.35