CHALMERS
UNIVERSITY OF TECHNOLOGY

EDA284 Lab 2: Vector Instructions and ARMv8 SVE using gem5

Contacts: Mustafa Abduljabbar, Jeckson Souza and Miquel Pericàs,
Emails: musabdu, jeckson, miquelp@chalmers.se

February 28, 2020

The ARM Scalable Vector Extension (SVE) is an extension to the vector processing capability associated with the ARM AArch64 execution state to better address the compute requirements in domains such as high performance computing (HPC), data analytics, computer vision and machine learning. It is scalable across multiple implementations by introducing the VLA (Vector Length Agnostic) registers, thus allowing the CPU designers to choose the vector length suitable for the power, performance and area targets.

In this lab, you will be analyzing the effect of utilizing vector units and of triggering SVE instructions accompanied with In-order and Out-of-order CPU models. By the end of this lab, you are expected to

(a) analyze the effect of compiler's auto vectorization.

(b) explore different vector width capabilities.

(c) differentiate between vectorizable and non-vectorizable codes.

(d) demonstrate the impact of a few SVE features such as per-lane predication, gather-load/scatter-load and vector issue width as opposed to NEON.

Useful and Optional References:

(a) Introduction to ARMv8 SVE and GEM5 simulator
http://www.max-centre.eu/sites/default/files/MaX-Arm-webinar-Javier.pdf.

(b) Short introduction to SVE features
https://developer.arm.com/docs/101726/latest/explore-the-scalable-vector-extension-sve/what-is-the-scalable-vector-extension.

(c) SVE for ARMv8 reference manual
https://static.docs.arm.com/ddi0584/a/DDI0584A_a_SVE_supp_armv8A.pdf.

(d) Porting and optimizing HPC applications for ARMv8 SVE
https://www.dropbox.com/s/pirzbphvy6hicj7/porting_and_optimizing_hpc_applications_for_arm_sve.pdf?dl=0.

(e) For Machine Learning enthusiasts, using intrinsic to extract the potential of ARMVv8 SVE https://developer.arm.com/solutions/hpc/resources/hpc-white-papers/arm-scalable-vector-extensions-and-application-to-machine-learning.

Prerequisites:

(a) The pre-built (same as before) `gem5` https://www.dropbox.com/s/r7vuluhqjsq7twy/gem5.tar.gz?dl=0.

(b) The cross compiler should be available on the lab machines (`aarch64-none-linux-gnu-g++`, `aarch64-none-linux-gnu-gcc`). If you need to run on your laptop, the Aarch64v8 cross-compiler is available on https://www.dropbox.com/s/ps5sw1iho07tu7e/gcc-arm-9.2-2019.12-x86_64-aarch64.tar.xz?ld=0. After you extract the compiler, add the folder that contains the bin to your path. Execute `export PATH=${PATH}:PATH_TO_GCC_BIN`

(c) The codes and binaries, referred to by this lab, can be found at https://www.dropbox.com/s/s1dqd4j8n5sfvzn/lab2_codes.tar.gz?dl=0.

1 Auto-vectorization of Data Parallel Codes

Due to its inherent data parallelism, dense matrix multiplication is favorable to vectorization. The following code snippet contains a couple of basic working implementations for such kernels.

```
#include <stdlib.h>
typedef float real_t;
typedef real_t* arr_t;
#define VERSION 0 // pick version 0 or 1
#if VERSION == 0
arr_t matmul_basic(const arr_t A, const arr_t B, const int M, const int N, const int K) {
    arr_t C = (arr_t) malloc(M * N * sizeof(real_t));
    for( int m = 0; m < M; ++m ) {
        for( int n = 0; n < N; ++n ) {
            for( int k = 0; k < K; ++k ) {
                C[ m * M + n ] += A[ m * M + k ] * B[ k * K + n ];
            }
        }
    }
  return C;
}
#elif VERSION == 1
arr_t matmul_basic(const arr_t A, const arr_t B, const int M, const int N, const int K) {
    arr_t C = (arr_t) malloc(M * N * sizeof(real_t));
    for( int m = 0; m < M; ++m ) {
        for( int k = 0; k < K; ++k ) {
            real_t _a = A[m * M + k];
            for( int n = 0; n < N; ++n ) {
                C[ m * M + n ] += _a * B[ k * K + n ];
            }
        }
    }
  return C;
}
#endif
```

Tasks:

(a) Using the cross-compiler `aarch64-none-linux-gnu-gcc`, enable and compile version 0. Observe if SVE code is generated. To view the assembly output, you can use the following command:

```
aarch64-none-linux-gnu-gcc -O3 -march=armv8-a+sve -S -o assembly_ouput code.c
```

Did the code get vectorized with SVE? Why or why not?

(b) Compile the modified version 1, and comment on the output SVE code. Can you predict which code will have a better vector performance? Justify your prediction, then compare to the output vector code.

(c) Run both versions using `gem5`. Comment on the performance and document 2-3 differences using the stats output. Select vector width of 2048 bits. The following command can be used:

```
./build/ARM/gem5.opt -d PATH_TO_OUTPUT configs/example/se.py --param \
'system.cpu[:].isa[:].sve_vl_se = VECTOR_WIDTH_INDEX' --cpu-type=ex5_big \
--caches -c PATH_TO_BINARY -o 'ARGUMENTS' \
```

The `VECTOR_WIDTH_INDEX` is encoded as follows: 1 = 128bits, 2 = 256bits, 4 = 512bits, 8 = 1024bits, 16 = 2048bits.

(d) SVE introduces VLA (Vector Length Agonstic) registers. Given one of the versions above (e.g., version 1), comment on the advantage of using such registers.

(e) One of the SVE additions is the per-lane predication registers. Were they generated in either version of the codes above? Provide an assembly snippet with an explanation.

2 Vectorization with Dependencies

You are given the following two versions of 1D stencil computation, each applies a different definition for calculating the updated stencil point.

```
#include <stdlib.h>
typedef float real_t;
typedef real_t* arr_t;
typedef unsigned long long index_t;
typedef index_t* arr_index_t;
#define VERSION 0 // Select version
#define MAX_TIME 1000
#if VERSION == 0
void stencil_1d(arr_t A, const int nx) {
    for(int timestep = 0; timestep < MAX_TIME; ++timestep)
        for(int i = 1; i < nx - 1; ++i)
            A[i] = (A[i - 1] + A[i] + A[i + 1]) / 3.0;
}
#elif VERSION == 1
void stencil_1d(arr_t A, const int nx) {
    for(int timestep = 0; timestep < MAX_TIME; ++timestep)
        for(int i = 0; i < nx - 2; ++i)
            A[i] = (A[i] + A[i + 1] + A[i + 2]) / 3.0;
}
#endif
```

Tasks:

(a) Observe the ARMv8 output for both versions. What do you see?

(b) If SVE instructions were not generated in either version, explain the reason with a short numerical example (do not run any code).

(c) Run both versions using `gem5`. Comment on the performance and document 2-3 differences using the stats output.

(d) For the best version of the 2, run with all possible vector width capabilities and plot the vector scaling for single (float) and double precision. Write your comments on the vector efficiency.

(e) For the best version of the 2, disable SVE. NEON vector code should be generated. Run 2 examples (i.e., 2 problem sizes), compare to SVE versions and comment on the key differences in the ISA that led to such differences.