Introduction

In this lab we explore some of the modelling options in gem5 and take a look at the design space concerning the vector extensions NEON and SVE. We then make a quantitative comparison of the two vector extensions.

Part 1 - Hello World with gem5

a) According to the homepage, GEM5 supports the following ISAs: Alpha, ARM, SPARC, MIPS, POWER, RISC-V and x86 ISA.

b) Standard output:

```
gem5 Simulator System.  http://gem5.org
gem5 is copyrighted software; use the --copyright option for details.

gem5 compiled Feb  7 2020 10:35:14
gem5 started Feb 25 2020 17:20:48
gem5 executing on remote11.chalmers.se, pid 24445
command line: ../../..//build/ARM/gem5.opt -d ./m5out/delet ./main.py

Global frequency set at 1000000000000 ticks per second
warn: DRAM device capacity (8192 Mbytes) does not match the address range assigned (512 Mbytes)
0: system.remote_gdb: listening for remote gdb on port 7000
Beginning simulation!
info: Entering event queue @ 0.  Starting simulation...
warn: CP14 unimplemented crn[14], opc1[7], crm[15], opc2[7]
Hello world!
Exiting @ tick 17348000 because exiting with last active thread context
```

c) CPU models are found here:
`https://www.gem5.org/documentation/general_docs/cpu_models/SimpleCPU`

The CPU model in the tutorial was the *TimingSimpleCPU*, the most simple timing-based model. It executes each instruction in a single clock cycle, except memory requests which flow through the memory system. It is derived from the *SimpleCPU* model, which also includes *AtomicSimpleCPU*.

- AtomicSimpleCPU: uses atomic memory accesses. It uses the latency estimates from the atomic accesses to estimate overall cache access time.
- TimingSimpleCPU: uses timing memory accesses. It stalls on cache accesses and waits for the memory system to respond prior to proceeding.
- O3CPU: OoO model
- TraceCPU: attached to O3CPU (oOo)
- MinorCPU: in-order CPU.
- DerivO3CPU: out-of-order CPU.

d) In GEM5, 1 *tick* corresponds to 1ps. The usual standard metric of clocks-per-cycle (CPI) therefore needs some adjustment, as a cycle is measured in a variable number of ticks depending on the frequency of the (simulated) processor. Once this has been accounted for, however, it becomes a useful point of comparison.

The atomicSimpleCPU treats all memory systems as ideal and thus, besides having the lowest CPI of all the models, its CPI is unchanging when the frequency is varied.

The O3CPU has almost half the CPI of the timingSimpleCPU, and for both the CPI is somewhat higher when doubling the frequency (presumably due to a higher number of cache misses on the L1 D-cache).

e) There are several different memory controllers available in the simulator representing DRAM technologies. The standard type used thus far has been the `DDR3_1600_8x8()` memory controller.

For this experiment we replace it with the `DDR4_2400_16x4()` memory controller.

We would generally expect the memory bandwidth to be significantly improved, though the observed differences in the output were marginal. Reasons for this could be the relatively low number of total memory accesses (experimentation using the O3 CPU gave 396 read requests), relatively low CPU clock speed not able to provide instructions with a high enough frequency to take advantage of the added bandwidth.

If a longer test program were used, it is likely that a larger difference would have been observed.

Part 2 - ARM SVE Simulation with gem5

a) Auto-vectorisation of Data Parallel Codes

The source code was compiled with and without SVE. Comparing them side-by-side, the vectorised version is apparent due to the presence of Scalable Vector Length (z) registers and Per-lane Predication (p) registers, as per [1].

b) Which code has better performance?

Version 1 will perform better than version 0 since the computation for the quasi-constant `_a` is performed outside the last for-loop, and therefore executed less often. V1 also skips the addition (accumulation) for the C matrix in the last line.

After examining the assembly output, we noticed that V1 only uses one floating point operation (mul) whereas V0 uses two (mul, add).

c) Differences

Both versions of the `matmul` program were simulated with a vector width of 2048 and compared. The output statistics reveal an order of magnitude difference in execution time, with version 1 being significantly faster: a speedup of ≈ 13.

Furthermore we could observe that the proportion of various types of instructions varies greatly between version 1 and version 0. Integer-ALU instructions make up 9% of all instructions in version 0 and 40% in version 1, SIMD floating point multiplications constitute 1.23% in version 0 and $\approx$ 10% in version 1, memory reads account for 82% of the instructions in version 0 and $\approx$ 15% in version 1 and memory writes make up 10% in version 1 and less than 1% in version 0.

d) Advantage of VLA registers

VLA registers do not have a fixed length, only a predefined maximum length to make vectors scalable, which leads to increased flexibility. Furthermore, the empty vector positions do not need to be filled, which usually saves one for-loop (*loop tail*).

In terms of our test code, namely matrix multiplication, the vector register length can be tailored to the row length of a matrix, thus potentially achieving significant throughput.

[1] `https://developer.arm.com/architectures/instruction-sets/simd-isas/sve/sve-programmers-guide/introduction-to-sve/single-page`

e) Per-lane Predicate Register

Predicate registers contain boolean conditions for instruction execution. Parallelism can be exploited by allowing both sides of a branch to execute, keeping one result and discarding the other based on the condition in the predicate register. This condition is in a sense precomputed as any arbitrary boolean expression can be used as a condition and the result stored in the p register.

Predicate *lanes* contain one predicate bit for each byte in a corresponding **z** register (e.g. 8 bits for a **z**-reg of 64 bits). Such *lanes* are either 'active' or 'inactive' depending on the value of the LSB.

Both versions of the program make use of per-lane predication registers (**p**redicate registers). This is clear due to the usage of p<X> registers. In these two cases, the p-registers are being used in conjunction with operations that use the z (vector) registers as well as in conditional statements (such as `whilelo`) that determine branching.

For version 0:

```
    .L5:
ld1w z3.s, p0/z, [x4, x1, lsl 2]
ld1w z2.s, p0/z, [x2, z4.s, sxtw 2]
incw x1
movprfx z0.s, p0/z, z2.s
fmul z0.s, p0/m, z0.s, z3.s
add x2, x2, x5
fadda s1, p1, s1, z0.s
whilelo p0.s, x1, x3
bne .L5
```

For version 1:

```
.L4:
ld1w z0.s, p0/z, [x4, x1, lsl 2]
fmul z0.s, z1.s, z0.s
st1w z0.s, p0, [x2, x1, lsl 2]
incw x1
whilelo p0.s, x1, x3
bne .L4
```

Part 3 - Vectorisation with Dependencies

In this section we refer to the two versions of the stencil program as V0 and V1.

a) ARMv8 output

In the assembly output for the two versions, only V1 produces code with SVE features. That is, no p or z registers are used in V0.

b) SVE Instructions

In V0, no vector instructions were generated. The reason for this is that V0 introduces data dependencies between subsequent iterations.

V0 implements A[i] = (A[i - 1] + A[i] + A[i + 1]). In a subsequent iteration, A[i-1] will depend on A[i] from the previous iteration.

V1 has no such dependency as it does not access "previous" values, only subsequent ones. Once a result is written back to memory it is not accessed again. Due to this, the entire vector can be parallelised.

c) Performance + Differences V0, V1

The two versions of `stencil` differ significantly in performance and how they are processed.

The ultimate measure of performance is, of course, execution time $T_{\text{exe}} = IC \cdot CPI \cdot T_c$. In all experiments comparing the different `stencil` modes T_c is kept constant, and can thus be omitted from the equation. We will not have the real execution time but all comparisons will be equally valid.

This difference in execution time is enormous: `stencil_1` performs over two orders of magnitude faster, or 49x faster. This can be seen in table 1.

Table 1: Differences in execution time between the difference versions of the stencil program.

	sim_insts	cpi	Texe
stencil_0	721443	3.44	2481271
stencil_1	22045	2.63	57925

Other significant differences include the mix and proportions of instructions in the programs. V0 almost exclusively uses scalar instructions (IntAlu, FAdd, FDiv etc), the number of which is two orders of magnitude higher than the corresponding vector instructions for V1. Predictably, the number of memory reads/writes is also proportionately higher.

V1 replaces all floating-point instructions with corresponding vector instructions, as can be seen in table 2. Predicate register instructions are also only present in V1. One last significant difference is the large number of memory accesses in V0 compared to V1. This is due to V0 reading and writing from/to individual memory addresses, whereas V1 reads and write vectors in contiguous memory addresses as a single instruction.

Table 2: Proportions of instructions in the two versions of the stencil program.

	Inst type								
	IntAlu	FAdd	FDiv	SimdAlu	SimdCmp	SimdFAdd	SimdFDiv	MemRd	MemWr
stencil_0	311785	204400	102200	8	8	0	0	103115	103292
stencil_1	6685	0	0	1608	1609	3200	1600	5615	2692

It is fairly easy to see how the instructions correspond to the actual source code.

```
arr_t A = (arr_t) malloc(1024 * sizeof(real_t));
stencil_1d(A, sz);

void stencil_1d(arr_t A, const int nx) {
    for(int timestep = 0; timestep < 100; ++timestep)
        for(int i = 1; i < nx - 1; ++i)
            A[i] = (A[i - 1] + A[i] + A[i + 1]) / 3.0;     // V0

        for(int i = 0; i < nx - 2; ++i)
            A[i] = (A[i] + A[i + 1] + A[i + 2]) / 3.0;     // V1
}
```

The kernel of the program uses 100*1024=102400 iterations and contains one floating-point division and two floating-point multiplications. This matches almost exactly with the number of floating-point and memory instructions in V0.

In V1, a vector width of 2048 bits is used. This is set at design-time (i.e. when configuring the CPU in the simulator). This corresponds to a width of 64 32-bit single-precision floating-point variables (`arr_t` uses floats). With this considered, it becomes obvious that 102400/64=1600, which accounts for all the floating-point instructions in all iterations.

d) Vector Efficiency

V1 of the `stencil` program was further examined due to its superior performance and heavy use of vector instructions.

All possible vector widths were explored, as well as the effect of changing the data type from single-precision to double-precision (i.e. 32-bit to 64-bit). The results are shown in fig. 1.

Since a given vector width can only fit half as many double-precision operands as single-precision ones, and the number of operands to be processed is fixed, we expect to get approximately half the performance (lower is better), and indeed this is the case.

As for the shapes of the curves, it is clear that it is asymptotic with diminishing returns for the width. The greatest gains in performance are made between 1-8, with performance starting to level off at this point.

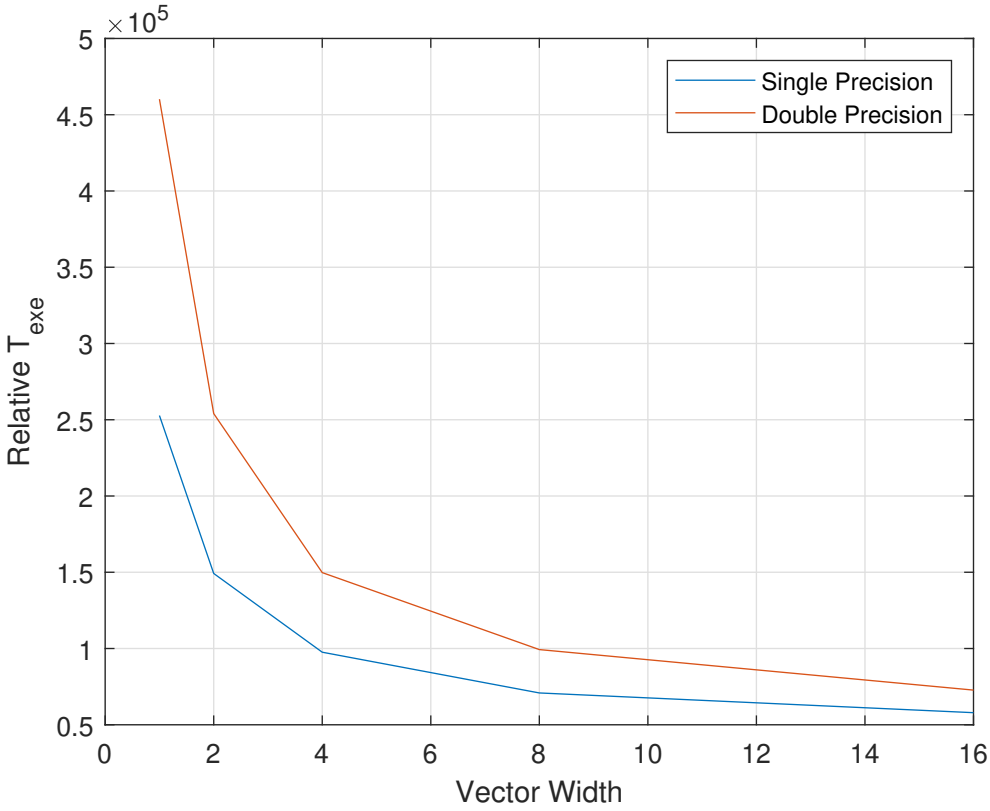

Figure 1: Relative performance of the `stencil` program for different data types and CPU vector widths. Widths are in terms of $128x$-bit.

e) NEON vs SVE

In this experiment we switch back to single-precision operands. Additionally, gem5 is run using the maximum vector width for SVE, namely 2048 bits. NEON registers have fixed width of 128 bits.

V1 of the stencil program was compiled using NEON and SVE ISAs, for different problem sizes (i.e. different matrix dimensions).

In the resulting assembly output, after having disabled SVE, we can see that the three main computations in the kernel of the program (two `fadd`s and an `fdiv`) are using the NEON ISA syntax, as the operand vectors are prefixed with a 'v'.

```
fadd    v0.4s, v0.4s, v2.4s
fadd    v0.4s, v0.4s, v1.4s
fdiv    v0.4s, v0.4s, v3.4s
```

In these commands, the 'f' prefix of the instruction indicates a float data type, 'vx' is a specific 128-bit NEON register and '4s' indicates that it contains four 32-bit words (i.e. floats). This is clear from `https://community.arm.com/developer/tools-software/oss-platforms/b/android-blog/posts/arm-neon-programming-quick-reference`.

On the other hand, there are now many instances of `fadd`s and `fdiv`s using scalar registers, though these may simply be for loop control.

If we plot the number of clock cycles taken to complete each problem size, we observe that SVE gives a relatively linear trend whereas NEON gives a quadratically increasing execution time. This is seen in fig. 2.

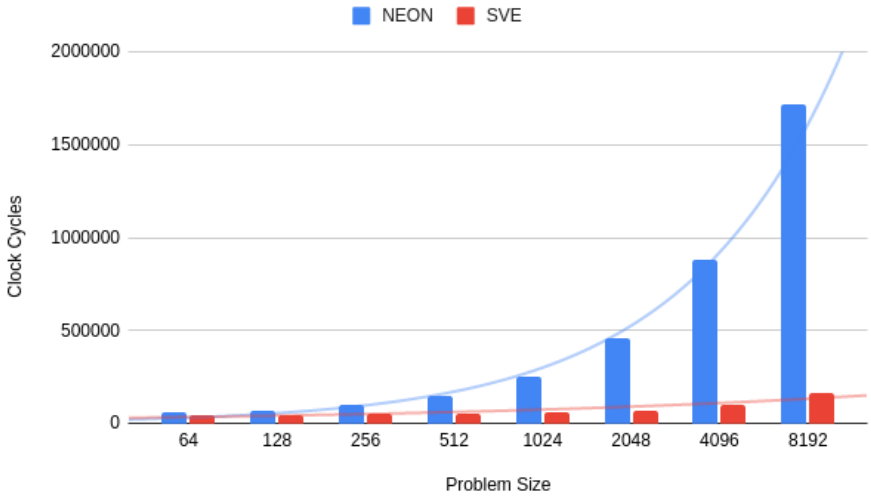

Figure 2: Clock cycles for each problem size and ISA.

The statistics generated after each run indicate that the NEON implementation typically results in the number of SIMD instructions, as well as memory accesses, differing by a constant factor from the SVE implementation depending on how much more data the SVE vector can hold. This is logical, since NEON uses vector registers that are fixed at a smaller width than SVE. Thus we see an approximate doubling in instruction count for every doubling of problem size. The execution time for SVE appears to be relatively constant until the problem size reaches the size of the vector register (2048 bits), at which point it also seems to grow quadratically.

In summary, SVE provides significant performance benefits over NEON when the problem size exceeds that of the NEON vector registers, even if the SVE vector registers are larger than what is required for a full stride.