A design space exploration for optimal vector unit composition

Jack Jones, Simon McIntosh-Smith

Introduction

Most modern high performance processors include support for SIMD or vector instructions. Deciding the optimal width and number of these operations can be challenging. One way Arm supports vector instructions is through the Scalable Vector Extension (SVE). SVE supports the compilation and execution of vector-length agnostic code. Rather than specifying a set vector length, it is left as implementation-defined. This allows for improved flexibility between CPU designs. How CPU architects choose to implement support for SVE presents an interesting area of research. High performance processors such as Fujitsu's A64FX and AWS's Graviton3 implement support for SVE with vector lengths of 512-bits and 256-bits, respectively. The reasoning behind these design choices is not abundantly clear, which raises the question of whether they were optimal. Similar questions remain for future SVE-enabled cores: **what is the optimal vector width and number of vector units?**

To explore the SIMD/vector design space, we need an accurate, fast, and highly modifiable simulation environment. The Simulation Engine (SimEng), a simulation framework from the University of Bristol's HPC group, enables this kind of design space exploration research. SimEng provides fast, cycle-level simulation of out-of-order processors. The framework is highly configurable, enabling the modelling of existing and hypothetical high performance cores, including cores of extreme design. Within this study, we have utilised SimEng's support for SVE to explore various hardware implementations and act as a reference for non-architecture-specific vector unit compositions. By altering the implementation of SVE operations in theoretical and existing processor models, we explore how their choices have impacted the performance of selected, highly vectorisable, workloads.

Methodology

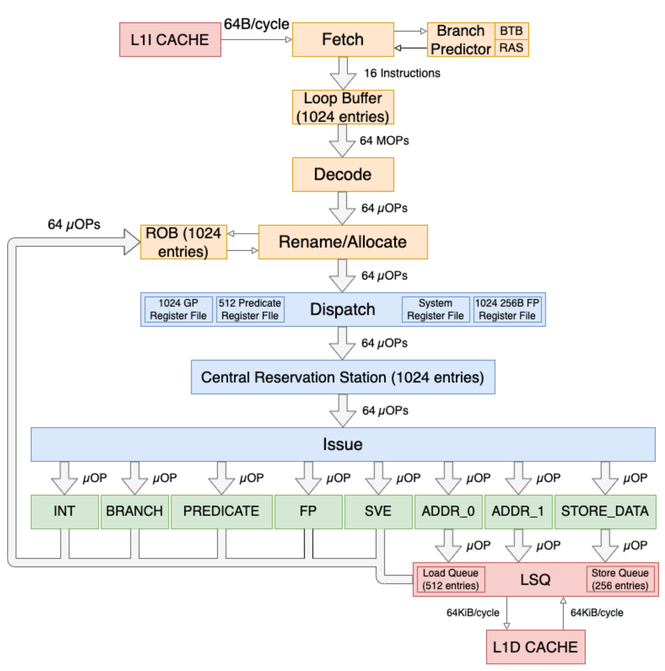

Figure 1: Hypothetical core model

To analyse the behaviour of various SVE implementations, we have constructed a theoretical core model. The primary parameterisation of the SVE implementations constructed within this study concerns the width and number of vector execution units. Therefore, this core was designed with the intent to stress the utilisation of these units. This targeted resource pressure allows for observations in emergent execution behaviour (e.g. the number of core cycles) to be more strongly associated with varying SVE implementations.

Figure 1 shows a diagram of the modelled core. The theoretical core boasts a very wide frontend with large numbers of Out-Of-Order resources to support it. The execution pipeline arrangement has been chosen to ensure minimal contention across varying instruction types. This arrangement allowed us to more feasibly calculate expected throughputs of workloads and better isolate the effects of various vector widths.

Two SVE execution pipeline arrangements were selected. Firstly, a single SVE execution pipeline that scales from 128-bits to 2048-bits in increments of 128-bits. Secondly, various decompositions of a 2048-bit vector width amongst multiple pipelines. For example, 2x1024, 4x512, or 16x128. The latency of SVE operations remains constant across all vector widths. To support this, we have assumed the number of vector lanes in each vector unit scales with the vector width. When modelling multiple SVE execution units, the number of address generation pipelines is scaled relative to the decomposition. This ensures the resource pressure within the core's execution engine remains on the SVE units.

To ensure changes to the implementation of the SVE execution pipelines were the sole influencer of execution behaviour, the memory subsystem was kept constant during the design space exploration. This was achieved by modelling an infinite L1 cache with a very wide, 64KiB datapath. Thus the memory subsystem should never be the limit on the flow of instructions across the chosen vector widths. However, as discussed in the future work section, we will explore the parameterisation of the memory subsystem in upcoming work.

DAXPY

The DAXPY function is a level 1 BLAS function of the form

$$Y := \alpha * X + Y$$

with X and Y being double precision vectors and α being a double precision scalar. Due to the relatively simple nature of the DAXPY function, its inner-loop body is small in SVE.

```
401e14 : ld1d {z1.d}, p0/z, [x1, x4, lsl #3]
401e18 : ld1d {z2.d}, p0/z, [x2, x4, lsl #3]
401e1c : fmla z2.d, p0/m, z0.d, z1.d
401e20 : st1d {z2.d}, p0, [x2, x4, lsl #3]
401e24 : incd x4
401e28 : whilelt p0.d, x4, x0
401e2c : b.mi 401e14
```

Taking the assembly generated for the Arm performance library's (ArmPL) DAXPY function, we can determine the maximum theoretical instruction throughput for the core being modelled. From this, we can calculate the expected number of cycles it would take to execute the DAXPY function for a given vector size. In the simplest case where there is a single SVE pipeline, we can expect each loop of DAXPY to complete every 1.5 cycles (actually 2 iterations every 3 cycles). The limiting factor per iteration is the `st1d` instruction which stores a resultant element of the DAXPY operation. Both the `ld1d` and `st1d` instructions contend for the use of the ADDR_0 and ADDR_1 address generation pipelines. The resultant access pattern of these two pipelines tends towards issuing two loads every third cycle and a load and store otherwise. Therefore, over a three-cycle period, two stores are retired yielding a rate of $\frac{3}{2}$ or 1.5 completions per cycle. The following graph showcases the expected and simulated cycle counts for the ArmPL DAXPY function.

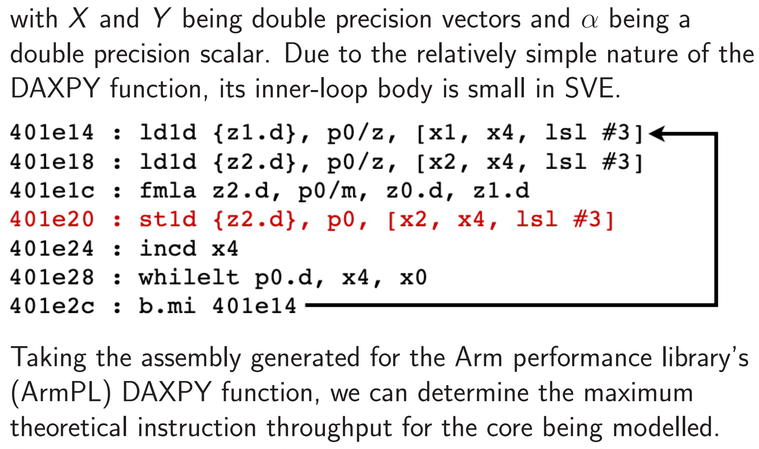

Figure 2: DAXPY simulated cycles for a single vector pipeline

Figure 2 shows the increase in the SVE Vector Length (VL) is proportional to the decrease in cycles taken to execute the workload. For a 2x increase in VL, we observe a 2x decrease in instructions required to execute the DAXPY function. Given the constant 1.5/cycle completion rate of the inner-loop body, we expect a proportional decrease between cycle counts and instruction counts. Changing the pipeline from one SVE unit to multiple pipelines, decomposing a 2048-bit VL yields the same behaviour.

DGEMM

The DGEMM function is a level 3 BLAS function of the form

$$C := \alpha * A * B + \beta * C$$

with A, B, and C being double precision matrices, and α and β being double precision scalars. The ArmPL assembly for DGEMM is too complex to easily calculate its expected cycle count. However, we have observed the inner loop is compute-bound. With a single SVE pipeline, we would expect similar behaviour to that seen in Figure 2.

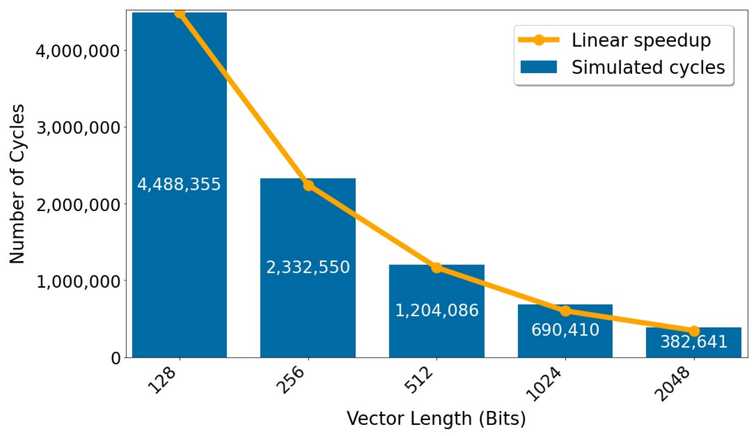

Figure 3: DGEMM simulated cycles for a single vector pipeline

We observe this DAXPY-like behaviour due to the increased depth of vectorisation as the VL increases; more iterations of the DGEMM inner loop are operated on per cycle as vectors widen. The sub-linear speedup shown is a result of the contention in register file resources. Instructions are stalled in the frontend of the pipeline forcing execution units to idle whilst existing instructions retire.

Figure 4: DGEMM simulated cycles for multiple vector pipelines

Figure 4 showcases the cycle count for the same workload but with the multiple pipeline arrangement. Although the aggregated vector width of all SVE pipelines remains constant, the execution cycle counts do not. We see the occurrences of backend pipeline stalls increase for the larger VLs. This increase results in execution units spending more cycles in an idle state. During these cycles, smaller VLs continue to feed their execution units and carry out up to 2048-bits of work across their multiple pipelines.

Arm Neoverse V1 model

Our hypothetical model is useful for stressing simulated vector units in isolation, while ignoring other micro-architecture features. However, the hypothetical model is not realistic. As a next step, we study a model based on real-world hardware, and alter its vector unit implementation to provide a more realistic parameter space. A suitable candidate is Arm's recent Neoverse V1 processor offering two 256-bit SVE pipelines.

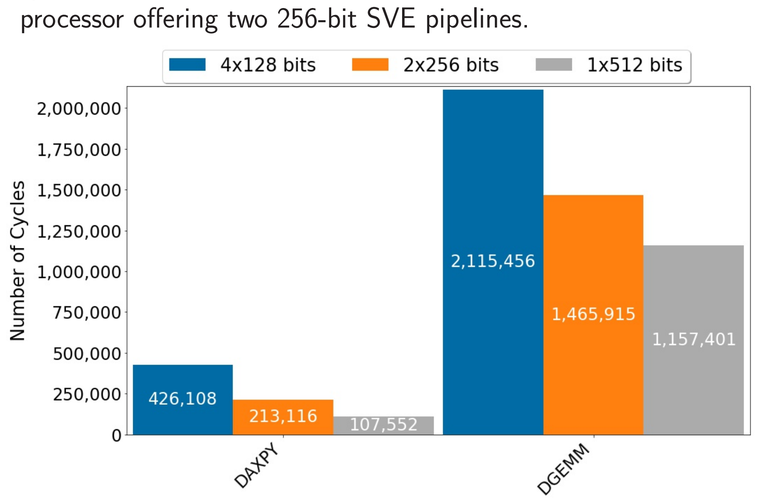

Figure 5: Simulated cycles across varying V1 models

Figure 5 shows the cycle count for the simulation of both DAXPY and DGEMM workloads under three V1 models. In addition to the original 2x256 SVE pipeline arrangement, modified V1 models with 4x128 and 1x512 vector unit compositions have been analysed. We have observed an improvement in the simulated cycle counts as the VL increases. During simulation, the V1 models exhibit diminishing utilisation of extra SVE pipelines. Whilst a single 512-bit pipeline is utilised fully, the 4th 128-bit pipeline is severely under-utilised for SVE operations. If the level of Instruction Level Parallelism could be improved, such issues may be resolved. Altering a compiler's cost model to better represent targeted hardware might achieve this.

Future Work

We plan to explore more of the design space and analyse more complex workloads that represent real-world examples.

The parameterisation of the memory subsystem has the potential to expose further limitations and advantages of varying vector widths and decompositions. The very wide models explored so far place a heavy reliance on memory accesses to feed the vector compute units. Mini-apps such as miniBUDE and TeaLeaf offer simplified variants of real-world workloads, focusing primarily on the innermost kernels. Carrying out similar studies with these mini-apps helps us to better translate the observations made within simulation to the real-world. They provide a method to understand the resource contentions seen in existing hardware. Using these mini-apps we will explore optimal vector unit configurations for future microarchitectures.